计算机研究与发展
Journal of Computer Research and Development
ISSN 1000-1239/CN 11-1777/TP
50(5): 1044-1054, 2013

一个程序验证器的设计和实现

张志天　李兆鹏　陈意云　刘　刚
(中国科学技术大学计算机科学技术学院　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)
(zpli@ustc.edu.cn)

An Automatic Program Verifier for PointerC: Design and Implementation

Zhang Zhitian, Li Zhaopeng, Chen Yiyun, and Liu Gang
(*School 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 Formal verification is a major method to improve the dependability of software. One of the hot research areas is automatic program verification based on logical inference. So far there is no product which can be used directly in the industries, and the root of this problem lies in the slow development and difficulties of automated theorem proving. Our approach is based on the observation that program analysis methods can be used in collecting global information to support program verification. We build shape graphs at each program point in the program analysis phase. A method is proposed, which uses regular Hoare logic rules to reason about assignment not dealing with pointers in a C-like programming language. Such rules will be applied after aliasing is eliminated using the information of shape graphs. The soundness of the logic system has been proved. Furthermore, an approach is presented to verify data constraints on mutable data structures without using user-defined predicates. A prototype of our program verifier has been implemented for the PointerC programming language. We have used it in the verification of programs manipulating recursive data structures, such as lists and trees, and programs dealing with one-dimension arrays.

Key words program verification; Hoare logic; shape graph logic; program analysis; separation logic

摘　要 形式验证是提高软件可信程度的重要方法,基于逻辑推理对程序性质进行严格的自动证明是当前的研究热点,但尚无可供工业界使用的产品,其根源在于自动定理证明方面的困难.介绍在通过程序分析建立起各程序点的形状图的基础上,如何利用形状图提供的信息来支持程序验证的方法.提出一种利用形状图信息来消除访问路径别名,使得指针程序中非指针部分的性质仍然可以用 Hoare 逻辑来进行验证的方法,并证明了该方法的可靠性.还提出一种在不使用自定义谓词的情况下,易变数据结构上数据性质的描述和验证方法.另外,介绍所设计并实现的基于上述方法的 PointerC 语言的程序验证器的原型.它不仅能自动验证操作易变数据结构程序的性质,也能自动验证使用一维数组的程序的性质.

关键词 程序验证;Hoare 逻辑;形状图逻辑;程序分析;分离逻辑

中图法分类号 TP301

收稿日期:2010-11-29;**修回日期**:2012-12-10
基金项目:国家自然科学基金项目(61170018,61003043)
通信作者:李兆鹏(zpli@ustc.edu.cn)

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

形式验证是提高软件可信程度的重要方法.粗略地说,软件的形式验证有两种途径.第1种途径是模型检测[1],它通过遍历系统所有状态空间,能够对有穷状态系统进行自动验证,并自动构造不满足验证性质的反例.这种方法在工业界比较流行,其优点是需要最小的用户交互,并可用于大规模复杂系统,近年来广泛用于清扫现有代码的错误上.第2种途径是逻辑推理,它采用程序逻辑,比较有影响的是Hoare逻辑[2],对程序的性质进行严格地推理,并且在推理过程中通常需要使用像Isabelle[3]或Coq[4]这样的定理证明器.在这种途径中,大部分研究围绕采用某种演算来产生验证条件,然后用某个定理证明器来证明验证条件,如Ynot[5],Spec#[6]和ESC/Java[7].有些研究依靠符号计算及其过程中的定理证明来避免验证条件生成步骤,如Smalllfoot[8]和jStar[9].还有的研究采用经严格证明的变换,从抽象规范逐步求精得到具体程序,如Perfect Developer[10].虽然这些工具已在实验室研发出来,但是尚无可供工业界使用的产品问世.究其原因,根源在于自动定理证明方面的困难.因为不管是访问路径的别名判断、循环不变式的推断、断言语言的表达能力和领域专用逻辑的设计等,最终都受到自动定理证明的能力的影响.

在研究自动定理证明技术的同时,也应该考虑怎样降低对自动定理证明器的能力的要求.例如,设计新的编程语言机制来提高合法程序的门槛,以排除部分有逻辑错误的程序.还有采用其他程序分析方法来收集程序信息,用这些信息来支持程序验证.这些都有可能减轻程序验证的负担.本文介绍在通过程序分析建立起各程序点形状图(shape graph)的基础上,如何利用形状图提供的信息来支持程序验证的方法,介绍基于该方法开发的一个程序验证器原型.

本文的主要贡献是提出一种利用形状图信息来消除访问路径别名,使得指针程序仍然可以用Hoare逻辑来进行验证的方法,并证明了该方法的可靠性.

Hoare逻辑的一个重要限制是程序中不同的名字代表不同的程序对象,即不允许出现别名.例如,在下面的三元式中:

$$\{Q\} *p = 5 \{p == q \wedge *p + *q == 10\},$$

访问路径$*p$和$*q$互为别名.若按Hoare逻辑的赋值公理,最弱前条件Q是$p==q \wedge *q==5$,而正确的最弱前条件是$p==q$.

对于访问路径的别名,一种针对性的解决办法是引入存储模型,把存储器M看成地址到被存储值的映射,并且有下面的存储器公理,其中E_1和E_3是地址表达式,E_2是值表达式:

$$select(update(M,E_1,E_2),E_3)=E_2,\ \text{if}\ E_1==E_3;$$

$$select(update(M,E_1,E_2),E_3)=select(M,E_3),\ \text{if}\ E_1!=E_3.$$

然后,将Hoare逻辑的赋值公理修改为

$$\{Q[update(\mu,E_1,E_2)/\mu]\} * E_1=E_2\{Q\},$$

其中,μ是存储器当前状态.利用该赋值公理和存储器公理,对先前的那个例子进行演算,可以得到最弱前条件$p==q$.这种办法的缺点是需要引入语义模型.

另一种解决办法是采用分离逻辑[11].分离逻辑通过把动态分配的各堆块的断言分离地表示,使得别名仅可能出现在同一个堆块的断言中.再通过对赋值语句的推理规则的特别设计,使得必须先消除堆断言中出现的别名,然后才可以对修改该堆块单元的赋值语句进行推理.使用分离逻辑的一个问题是,通常的自动定理证明器都不能证明带分离合取连接词(*,separating conjunction)的验证条件,必须为分离逻辑设计专用的自动定理证明工具.而使用这种自动定理器,规范的表达性受限于分离逻辑的一个可判定片段(fragment)[12].

在对操作易变数据结构的程序进行验证时,在各程序点形状图的支持下,本文方法仍然基于普通的Hoare逻辑,既不需要从语义角度对它扩展,也不需要引入新的逻辑连接词,既方便了程序员也减轻了自动定理证明器的负担.

其次,本文提出一种在不使用自定义谓词的情况下,易变数据结构上数据性质的描述和验证方法.主要是指断言中允许使用带上角标的访问路径(即$p(->next)^i$代表$p->next->\cdots->next$,其中$->next$重复i次),以便用带全称量词的断言描述链表的有序性等性质.在产生验证条件时,通过引入辅助函数来把带上角标的访问路径变换成可满足性模理论(satisfiability modulo theories)求解器Z3[13]能接受的形式.

最后,本文介绍我们所设计并实现的基于上述方法的 PointerC 语言的程序验证器的原型.它不仅能自动验证操作易变数据结构程序的性质,也能自动验证使用一维数组程序的性质.

1 PointerC 语言和形状图逻辑简介

本节介绍验证器面向的编程语言 PointerC 和验证器所依赖的形状图逻辑(shape graph logic).

1.1 PointerC 语言

PointerC 是一种强调指针类型并增加形状声明的类 C 小语言,语法的主要产生式如图 1 所示.在结构体声明中,通过指针域指向形状的声明来确定这种结构体用来构造什么形状的数据结构.这同时也限定了该结构体类型的指针所能指向的形状.这

program::=*struct_def_list var_dec_list fun_def_list*
struct_def_list::=*struct_def_list struct_def*|ε
struct_def::=typedef struct *id*{*field_dec_list*} *type_name*;
var_dec_list::=*var_dec_list var_dec*|ε
var_dec::=*type id_list*;
field_dec_list::=*field_dec_list field_dec*|*field_dec*
field_dec::=*field_type id*;
id_list::=*id_list*,*id*|*id*
fun_def_list::=*fun_def_list fun_def*|*fun_def*
fun_def::=*type id*(*param_list*) *body*|*type id*() *body*
param_list::=*param_type id*|*param_list*,*param_type id*
body::={*entry_assertion var_dec_list stmt_list exit_assertion*}
type::=*simple_type*|*element_type* [*number*]|*type_name**|void
type_name::=*id*
element_type::=*simple_type*
field_type::=*simple_type*|*type_name**:*shape*
param_type::=*simple_type*|*type_name**
simple_type::=bool|int
shape::=*LIST*|*DLIST*|*C_LIST*|*C_DLIST*|*TREE*|…
entry_assertion::=assertion *assertion*;|ε
exit_assertion::=assertion *assertion*;|ε
stmt_list::=*stmt_list stmt*|*stmt*
stmt::=*lval*=*exp*;|*lval*=*malloc*(*type_name*);|if(*exp*) *block*
|if(*exp*) *block* else *block*|while(*exp*) *loop_invariant block*
|*lval*=*id*(*exp_list*);|*id*(*exp_list*);|*lval*=*id*();|*id*();
|return;|return *exp*;|*free*(*lval*);
block::=*stmt*|{*stmt_list*}
loop_invariant::=loop_invariant *assertion*;|ε
exp::=*number*|*lval*|NULL|true|false
|−*exp*|!*exp*|*exp op exp*|(*exp*)
lval::=*id*|*lval*−>*id*|*id* [*exp*]

Fig. 1 Representative syntax of PointerC language.
图 1 PointerC 语言的语法(部分)

是按形状分析的需求所作的语言扩展,所允许的形状有单链表、循环单链表、双向链表、循环双向链表和二叉树等.在图 1 的文法中,有关表达式的部分产生式被略去,*assertion* 的语法在用到时再解释.

在 PointerC 中,指针类型的变量只能用于赋值、相等比较、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,并且没有指向指针类型的指针类型.*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.

1.2 形状图和形状图逻辑

程序验证之前需要先基于形状图逻辑对程序进行形状分析为每个程序点构建形状图,这些形状图构成了程序验证所需要的指针信息.在此通过举例来介绍形状图,形状图的严格定义请见文献[14].

图 2(a)是用形状图给出的单链表的归纳定义 *list*(*s*).圆节点称为声明节点,代表指针型的声明变量,唯一的出边上的标记是该变量的名字.框中没有标记的实线矩形称为结构节点,代表用 *malloc* 生成的结构体变量,其出边代表它的指针域,边上的标记是域名.框中有标记 $\mathcal{P}$ 的实线矩形称为谓词节点,框的下方是该谓词的名字,谓词节点没有出边.框中有标记 $\mathcal{N}$ 的虚线矩形称为 NULL 节点,表示指向它的边代表 NULL 指针.类似地,有标记的 $\mathcal{D}$ 虚线矩形称为悬空节点,表示指向它的边代表悬空指针.图 2(a)的谓词定义 *list*(*s*)表示单链表分成空表(*s* 是 NULL 指针)和非空(*s* 指向一个结构节点,该节

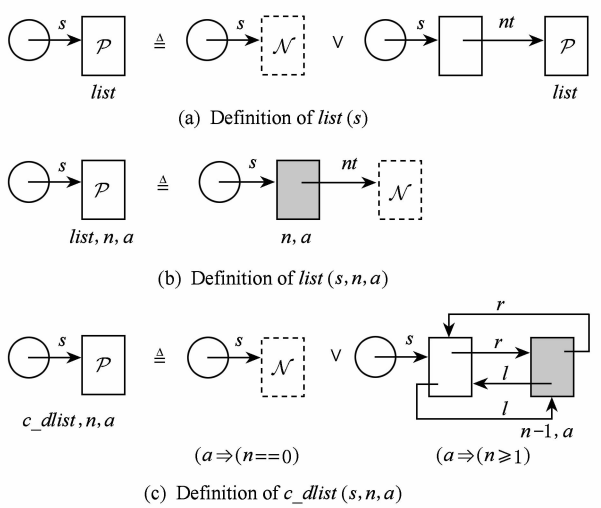


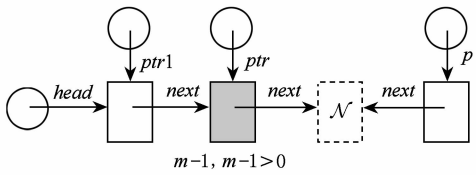
Fig. 2 Three definitions of linked list.
图 2 链表的几种定义

点再指向剩余节点构成的表)两种情况.

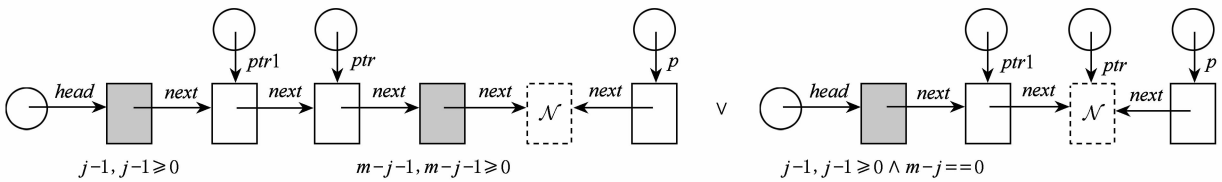
图 2(b)是单链表的一种非归纳定义 $list(s,n,a)$.灰色矩形节点称为浓缩节点(condensation node),是若干个(可以是 0 个)相邻的、属于同一数据结构的结构节点的概括表示.这些节点上和节点之间只有维系该数据结构的必要的边,外来的边只指向这组节点序列的边缘节点.浓缩节点的下侧可以有代表被浓缩节点个数的整型表达式 n 以及约束该表达式的断言 a.若没有这样的表达式和约束,则表示被浓缩节点的个数是某个自然数,但和程序中任何常量或声明变量都联系不起来.

谓词节点和浓缩节点下侧的长度或个数表达式是仅使用常量和声明变量的整型线性表达式,断言则是这类表达式的关系运算的逻辑合取式.

图 2(c)是循环双向链表的一种定义,两种情况分别表示空表和非空表.非空则至少有一个节点,后面有 $n-1$ 个节点.以图 3(a)和图 3(b)(分别为第 4 节例 1 的循环语句之前那个程序点的形状图和该循环的循环不变形状图)为例说明形状图和程序点指针等信息的联系.从图 3(a)可知,$head==ptr1$,$ptr==ptr1->next$,$head$ 指向的链表的长度是 m 并且 ptr 指向的浓缩节点代表 $m-1$ 个节点.



(a) Example one



(b) Example two

Fig. 3 Two examples of shape graph.

图 3 形状图的两个例子

形状图上的路径和它所代表的程序中的访问路径用同样的语法表示.只需要考虑从声明节点开始到达某个节点的路径,分成下面两种情况.

1) 路径的完全表示:在路径不穿越浓缩节点的情况下,由依次列出路径各边上的标记来表示该路径.若边上的标记依次为 $p,left,right$,则写成 $p->left->right$,即不是用逗号而是用 $->$ 连接边上的标记.

2) 路径的浓缩表示:若路径包括某浓缩节点的一条出边 $left$,并且该节点浓缩了 n 个节点,则需要让 $(->left)^n$ 而不是 $->left$ 出现在路径上,例如 $p(->left)^n->right$.若 $n=0$,则可以简化为 $p->right$.

这样,从声明节点开始到达某个节点的一条路径指称该路径最后那条边所代表的程序指针.

形状图是如下断言的图形表示:

1) 指向同一个结构节点或谓词节点的指针相等,指向浓缩节点展开后的同一个结构节点的指针相等,例如图 3(a)表示的断言中有 $head->next==ptr$;

2) 指向 NULL 节点(悬空节点)的指针都等于 NULL(是悬空指针);

3) 指向谓词节点的指针都满足相应的谓词.

因此,形状图可以作为程序断言,它是该图所能表达的指针相等、不相等和别名断言等的合取,包括其中谓词节点和浓缩节点下侧有关表长或被浓缩节点个数的整型数据断言.

形状图逻辑就是基于上面观点来设计的 Hoare 逻辑的一种扩展[14].程序规范的形式是 $\{G\wedge Q\}S\{G'\wedge Q'\}$,其中 G 是形状图,Q 是表达程序其他性质的符号断言,两部分的合取 $G\wedge Q$ 作为程序点完整的断言.本文程序验证器的第 1 步工作是,在无需程序员提供有关形状的函数前后条件和循环不变式的情况下,利用形状图逻辑对程序进行形状分析.由于从一个语句前的 G 推导该语句后的 G' 不受 Q 的影响,因此形状分析时,把程序规范简化为 $\{G\}S\{G'\}$,以此来使用形状图逻辑的推理规则,建立各程序点的形状图 G.在形状分析的过程中,还利用循环

不变式推断算法得出各循环的循环不变形状图[15].

在完成形状分析后,程序验证器进行程序其他性质 Q 的验证.在 $\{G\wedge Q\}S\{G'\wedge Q'\}$ 中,若 S 不是指针操作语句,则 G' 和 G 一样,但 Q' 可能不同于 Q.若 S 是指针操作语句(指针赋值、分配空间和释放空间等),则除了 G' 和 G 可能不同外,Q' 和 Q 可能也有一些细微的区别,这是本文下面关注的部分.

2 消除访问路径别名的程序验证方法

程序点数据结构构成的形状有多种可能时,则 G 表示为 $G_1\vee G_2\vee\cdots\vee G_n$,例如图 3(b)的循环不变形状图就是 $G_1\vee G_2$ 的形式.同样,Q 也可能是 $Q_1\vee Q_2\vee\cdots\vee Q_m$ 的析取形式,例如第 4 节例 1 的循环不变式就是 $Q_1\vee Q_2$ 的形式.完整的断言可以整理成析取范式(disjunctive normal form)$G_1\vee Q_1\vee G_2\vee Q_2\vee\cdots\vee G_k\vee Q_k$ 的形式.根据形状图逻辑,可以用析取范式的一种情况为例来讨论,写成 $G\wedge Q$,G 和 Q 分别为合取形式.

程序验证器基于形状图逻辑进行最强后条件演算并产生验证条件[16],验证条件由 $Z3$ 自动证明.

2.1 形状图和符号断言之间的联系

符号断言 Q 中允许出现指针是否等于 NULL 或两个指针是否相等的断言.例如在第 4 节例 1 的循环不变式中,这样的断言用来把其他断言分别联系到 ptr 是否等于 NULL 两种情况.即使函数前后条件和循环不变式中没有这样的断言,它们也会因为出现在条件语句或循环语句的布尔表达式中,而在最强后条件演算过程中被加到 Q 中,第 4 节例 1 也有这种情况.

Q 中指针等于 NULL 或两个指针相等的断言会因为和 G 中的信息重复而被吸收,或因有矛盾而使得 $G\wedge Q$ 为假.

Q 中访问路径的合法性依赖于 G.例如,在 Q 中若出现非指针型的访问路径 $p{->}\cdots{->}data$,则忽略 $->data$ 所剩下的前缀应该是 G 上到达某个结构节点的一条访问路径,若是到达悬空节点、NULL 节点或不存在这样的路径则都是非法的,若是到达谓词节点则视谓词节点展开后的情况决定.

Q 中的访问路径之间是否有别名,Q 中的访问路径和下一条语句 S 中的访问路径之间以及 S 中的访问路径之间是否有别名都依赖于 G,即利用 G 可以判断.

在指针操作语句中,在对指针 u 赋值时可能会影响符号断言:符号断言中若有以 u 或 u 为前缀的访问路径,则要用和 u 相等但不是别名的 u' 来代换 u.另一个影响符号断言的场合是,在 $free$ 语句之后应该删除涉及被释放节点上数据的原子断言.

G 中也会有符号断言,附加在浓缩节点上,用来限制它代表结构节点的个数.G 的符号断言和 Q 的符号断言不会有矛盾,但前者有时会给出更准确的信息,在第 4 节介绍例 1 时解释.

2.2 程序推理规则的扩展

在使用推理规则从语句 S 的前条件 $G\wedge Q$ 产生后条件 $G'\wedge Q'$ 时,要保证 Q 合法、Q 和 G 无重复和无矛盾.

先考虑 S 是指针操作语句.修改指针型数据的简单语句会引起指针值的变化,或者是存储堆块的增减,因而导致形状图的变化.根据 2.1 节的介绍可知,对 Q 的影响是访问路径的替换或者删除部分断言.先假定 Q 和 S 无别名,有别名的情况在考虑非指针操作语句时介绍.下面分各种语句给出规则.

1) 指针型赋值语句 $u=v$

若 u 既不是 NULL 指针也不是悬空指针,则按下面规则得到后断言:

$$\{G\wedge Q\}\ u=v\{G'\wedge Q[u'/u]\},$$

其中,G' 是由形状分析得到的形状图,$Q[u'/u]$ 表示 Q 中作为访问路径(包括作为前缀情况)的 u 及其别名用和它们相等且不互为别名的访问路径 u' 代换.

2) 对指针赋值的其他语句

分配空间语句 $u=malloc(t)$ 和函数调用语句 $ret=f(act)$ 有关 Q 的处理同上面赋值语句的规则一样.

3) 释放空间语句 $free(u)$

释放 u 指向的节点后,Q 中含 u 或 u 的别名的原子断言不应再存在,因此规则如下:

$$\{G\wedge Q\}\ free(u)\{G'\wedge Q'\},$$

其中,Q' 由把 Q 中含 u 或 u 的别名的原子断言都删除而得到.

很容易明白,若 Q 无别名,则这些语句的规则不会导致 Q' 出现别名,因为它们对 Q 作的小修改都不会引入别名.

再考虑非指针操作语句.只要前断言 Q 和语句 S 中无别名,则使用 Hoare 的赋值公理就是可靠的.若有别名则可以先用 G 的信息来消除别名(把互为别名的访问路径改成都用其中同一条访问路径),然后再用赋值公理.定义函数 $eliminate\_aliases$ 为 $(S',Q')=eliminate\_aliases(G,S,Q)$,它根据 G 消

除 S 和 Q 中的别名,得到 S'和 Q'.

我们把 Hoare 逻辑的赋值公理限定为无别名时才能使用,并增加下面的消除别名推理规则:

$$\frac{\{G \wedge Q''\}S'\{G \wedge Q'\}}{\{G \wedge Q\}S\{G \wedge Q'\}}$$

$$(S', Q'') = eliminate\_liases(G, S, Q),$$

就可以对含访问路径别名的程序进行推理.

对于修改指针型数据的语句,其前断言 Q 可能是程序员提供的,例如不排除循环不变式中 Q 存在别名,因此有时也需要这条规则.

复合、条件和循环语句的规则以及推论规则的形式和 Hoare 逻辑相应规则的形式一致.

在没有指针类型的情况下,使用 Hoare 逻辑的赋值公理从赋值语句的前断言 Q 得到后断言 Q'时,不用关心 Q 是否为 $Q_1 \vee Q_2$ 的形式.但是在有指针类型的情况下,$G_1 \vee G_2$ 代表相应程序点的形状图有两种可能,需要对它们分别考虑,因此需要增加一条分情况规则:

$$\frac{\{G_1 \wedge Q_1\}S\{G_1' \wedge Q_1'\}\{G_2 \wedge Q_2\}S\{G_2' \wedge Q_2'\}}{\{G_1 \wedge Q_1 \vee G_2 \wedge Q_2\}S\{G_1' \wedge Q_1' \vee G_2' \wedge Q_2'\}}.$$

先前提到的选取析取范式 $G_1 \wedge Q_1 \vee G_2 \wedge Q_2 \vee \cdots \vee G_n \wedge Q_n$ 的一种情况来讨论就是基于这条规则.

2.3 程序逻辑的可靠性证明

本节简要证明 2.2 节略加扩展的 Hoare 逻辑对 PointerC 语言的操作语义是可靠的(sound).

在有栈和堆的机器上,一个形状图再加上分配在栈上的整型和布尔型变量的图形表示就构成程序状态的一种图形表示.这种图形表示的程序状态区别于通常状态的主要地方是,每个指针的具体值不清楚,但它们之间相等与否是清楚的,因此可以理解为每个指针有一个抽象值.基于在这样抽象状态上的操作语义,文献[14]证明了下面两个重要性质:

1) 形状图断言的演算规则是可靠的;

2) 形状图逻辑中修改形状图的指针操作语句的程序推理规则是可靠的.

有了这两个性质,则 2.2 节的程序推理规则中,前后断言中的 G 和 G'是可靠的.

在没有指针类型的情况下,Hoare 逻辑的推理规则对通常的操作语义可靠已是众所周知的.因此下面仅非形式地证明几个涉及扩展的性质.

1) 对于非指针操作语句,Hoare 逻辑的赋值公理在没有别名的情况下使用是可靠的.

在无别名的情况下,赋值公理$\{Q[E/x]\}\ x=E\{Q\}$ 的赋值 $x=E$ 只改变 x 的值,不会改变 Q 中出现的任何其他变量的值,因此可靠性证明和无别名语言的 Hoare 逻辑的赋值公理的可靠性证明没有区别.

2) 对于指针操作语句,在 Q 中没有别名的情况下,这些规则对 Q 的修改都是可靠的.

以规则$\{G \wedge Q\}\ u=v\{G' \wedge Q[u'/u]\}$为例.若一条访问路径的前缀被替换,则替换前后的访问路径指称同一个数据单元.若是一条完整的访问路径被替换,就目前使用的断言语言来说,u 只可能作为参数出现在链表表长函数 $length$(u,结构体的指针型域名)中(见第 4 节例 1 的前断言),用相等的指针 u'来代换,则因它们指在同一个节点上而不改变 $length$ 的值.

3) 函数 $eliminate\_aliases$ 得到的 S'和 S 有同样的语义,Q'和 Q 也有同样的含义.类似 2),消除别名前后的访问路径指称同样的数据单元,因此语义和含义不变.

4) 消除别名的推理规则是可靠的.是 3)的推论.

5) 分情况规则的可靠性.在文献[14]中已经证明.

有了这几点则可以说所用的程序逻辑是可靠的.

3 易变数据结构上数据性质的验证方法

自动程序验证器的验证能力取决于所用的自动定理证明器的能力,供程序员描述函数前后断言和循环不变式用的断言语言也是据此来设计的.通用的自动定理证明器一般没有针对易变数据结构的性质证明的部分,本节介绍如何将所需证明的性质转化为 Z3 可以接受的形式.

易变数据结构上数据性质的描述,例如链表数据的有序性可以通过引入谓词来表示,也可以通过使用全称量词来表示,本文设计的验证器采用后者,例如,可以用下面的断言来描述 $head$ 所指向的长度为 m 的单链表的有序性:

$\forall i:1..m-1.(head(->next)^{i-1}->data \leqslant head(->next)^{i}->data)$.

在 PointerC 的布尔类型表达式的基础上,增加带上角标的左值表达式和预定义的用于各种链表的函数 $length$,并增加带量词的断言形式,就构成本文验证器提供的断言语言.为便于阅读,本文描述断言所用的逻辑符号并未遵从所设计的断言语言.

本文验证器采用最强后条件演算的方式来产生验证条件.通常在程序员提供断言的地方会产生验证条件,例如对程序员提供的每个循环不变式会产

生两个验证条件:循环入口点的断言蕴涵循环不变式和循环体最后一个语句之后的那个程序点的断言蕴涵循环不变式.

下面介绍把生成的验证条件提交给 Z3 时碰到的 3 个问题及其解决办法.

1) 把结构体的域名看成函数名,结构体的指针看成函数的变元,以此克服 Z3 不具备有关访问路径的知识带来的困难.

Bornat 在考虑指针程序的证明时,把数据堆看成由指针索引的一群对象[17],把对象看成由名字索引的一组成员.

在此采用类似的方式,把结构体的域名看成整数域上的一元函数名,把结构体的指针当成整型,作为函数的变元. 例如,第 4 节例 1 的结构体类型有两个域 $next$ 和 $data$,它们分别被看成函数. 验证条件中可能有的 $head->next$, $ptr->data$ 和 $ptr1->next->data$ 这样的访问路径分别被翻译成 $next(head)$, $data(ptr)$ 和 $data(next(ptr1))$. Z3 把 $next$ 和 $data$ 当成未解释函数[18],就是不知其定义的函数. 若有 $head==ptr1$,则从编程语言的语义知道 $head->next==ptr1->next$;在 Z3 中,若有 $head==ptr1$,则从函数的性质也会得出 $next(head)==next(ptr1)$.

2) 引入辅助的二元函数来克服带上角标的访问路径引起的困难.

断言中会出现像 $p(->next)^n->data$ 这样带上角标 n 并且 n 可以是线性表达式的访问路径,由于 n 的值一般不能静态确定,因此该访问路径不可能翻译成 $data(next(\cdots next(p)\cdots))$ 并且其中函数 $next$ 的应用次数确定的表达式.

可以把 $p(->next)^n$ 翻译成 $nextn(p,n)$,并引入一个二元函数 $nextn(p,n)$,它基于函数 $next$ 来定义(其中 $n\geqslant 0$):

$$nextn(p,n)=_{df}\begin{cases}p, & n=0;\\ next(p), & n=1;\\ next(nextn(p,n-1)), & n>1.\end{cases}$$

显然, $nextn(p,n)$ 表达的意思就是 $p(->next)^n$. 在访问路径的表示中, $p(->next)^{n+1}$ 就是 $p(->next)^n->next$;Z3 有了函数 $nextn$ 的定义后,很容易推出 $nextn(p,n+1)==next(nextn(p,n))$.

3) 将形状图上的信息用符号断言表示,以支持验证条件的证明.

符号断言形式的验证条件 $Q\Rightarrow Q'$ 的证明需要用到相应程序点的形状图 G 上的信息,因此验证条件应该是 $G\wedge Q\Rightarrow Q'$,但 G 上的信息必须符号化. 把 G 上的指针相等(包括指针等于 NULL)断言符号化,再加上浓缩节点下方的约束断言、链表长度断言(在验证条件中出现链表长度断言时需要),形成符号断言 P,因此实际的验证条件是 $P\wedge Q\Rightarrow Q'$. 若 G 是空图(如不涉及指针类型的程序),则验证条件直接就是 $Q\Rightarrow Q'$.

在把指针相等信息符号化时,为避免冗余,给出 G 上相邻两个静态声明指针变量之间的关系,剩下的相等关系可以推导. 例如对于图 3(b)的第 1 种情况,指针之间的相等关系转换成的符号断言如下:

$$ptr1==head(->next)^{j-1}\wedge ptr==ptr1->next\wedge ptr1(->next)^{m-j-1}==\text{NULL},$$

其中,第 3 个子断言不用给出,因为经过一个浓缩节点再指向 NULL 节点的信息对验证条件的证明通常没有用处.

对于双向链表,相邻两个节点之间指向对方的指针形成一个等式,若域名分别是 $right$ 和 $left$,则有 $p->right->left==p$ 或 $p->left->right==p$. 那么,代表 n 个节点的浓缩节点之间就有很多个等式,好在一般来说,当这 n 个节点在 G 上被浓缩,验证条件就不会涉及其中的指针相等关系,所以在将 G 符号化时不用给出这些相等关系.

4 系统原型

基于形状图逻辑和程序分析的支持,我们实现了 PointerC 语言的一个程序验证器[19],它能够验证不涉及易变数据结构的程序,也能验证涉及各种链表和二叉树的构建、插入和删除等的程序. 除了验证形状外,还能验证节点上数据的一些性质.

该验证器无需程序员提供有关数据结构形状的函数前后条件和循环不变式,但要求提供非指针型数据的函数前后条件和循环不变式.

4.1 系统流程及验证条件的生成

该验证器分成下面几个模块,按所列次序顺序执行.

1) 普通编译器的前端

对源程序进行词法分析、语法分析和静态语义检查后生成抽象语法树.

2) 形状分析

遍历抽象语法树,根据形状声明和形状图逻辑来生成各程序点的形状图. 其中在遇到循环语句时,

需要对它进行多次遍历,以推断循环不变状态图[14].

若整个程序不涉及指针类型,则在每个程序点生成的是空图.

3) 验证条件的生成

遍历抽象语法树,根据程序员提供的函数前后条件和循环不变式,按最强后条件演算方式为各函数生成验证条件.

若仅验证易变数据结构的形状,程序员不用提供任何断言,则程序验证在上一步就结束.

下面以 4.2 节例 1 为例,说明在本阶段如何将分先后得到的 G 和 Q 进行对应. 在该例中,$ptr==$NULL 和 $ptr!=$NULL 直接出现在程序员提供的循环不变式的两种情况($Q_1 \vee Q_2$)中,以表示 Q_1 和 Q_2 分别联系到 ptr 是否等于 NULL. 该循环的循环不变形状图如图 3(b)($G_1 \vee G_2$)所示,ptr 在这两个图上分别表现为等于和不等于 NULL. 因此,当完整的断言($G_1 \vee G_2$)$\wedge$($Q_1 \vee Q_2$)展开成析取范式时,在析取范式的 4 种情况中,上述 ptr 断言不是因和形状图有矛盾而导致该情况被删除,就是因一致而被吸收. 这样,在它们被吸收的两种情况中,符号断言和与之相适应的形状图就联系在一起.

4) 验证条件的证明

将生成的各验证条件 $G \wedge Q \Rightarrow Q'$,按照第 3 节所介绍的方法翻译成 $P \wedge Q \Rightarrow Q'$ 的形式,逐个交给 Z3 进行证明. 由于 Z3 是可满足性的证明工具,因此实际是从 Z3 证明 $\neg(P \wedge Q \Rightarrow Q')$ 不可满足来得出 $P \wedge Q \Rightarrow Q'$ 成立.

在 2.1 节提到 G 中也会有符号断言,G 和 Q 的符号断言一般不会有矛盾,但 G 有时会给出更准确的信息. 在例 1 中,$j \leqslant m$ 不能加入循环不变式,因为仅 j 在循环中增 1. 因此通过最强后条件演算在循环出口点只能得出 $j \geqslant 1$,而从 G(这时是 $G_1 \vee G_2$)可以得到 $j < m$ 和 $j == m$ 两种情况,因为它们分别出现在 G_1 和 G_2 的某个浓缩节点附带的约束断言中.

用该系统原型已经验证过的简单程序分成 3 类:

1) 只涉及数据结构形状的程序:单链表、循环单链表、双向链表、循环双向链表和二叉树的创建、插入和删除等函数.

2) 不涉及指针类型数据的程序:冒泡排序、快速排序等程序.

3) 两者都涉及的程序:有序单链表的插入程序等.

4.2 两个例子

例 1. 本例是在有序单链表中插入一个节点,插入节点后链表保持有序,返回结果链表的指针,代码和断言如图 4 所示,其中作为链表节点的结构体类型的定义是 typedef struct $listnode\{Node*: LIST\ next$;int $data$;$\}Node$. 该程序不仅涉及形状,而且需关注各节点数据之间的大小关系.

```
Node* insert(Node* head,int data){
  assertion m==length(head,next) ∧
    ∀ i:1..m−1.(head(−>next)^{i−1} −>data≤head(−>next)^i
      −>data)
  Node* ptr;Node* ptr1;Node* ret;Node* p;
  int j;
  p=malloc(Node*);p−>data=data;p−>next=NULL;
  if(head==NULL){
    ret=p;
  } else if(p−>data≤head−>data){
    p−>next=head;ret=p;
  } else {
    ptr1=head;ptr=head−>next;j=1;
    while((ptr!=NULL) &&. (ptr−>data< p−>data))
      loop_assertion
        ptr!=NULL ∧ ∀ i:1..j−1.(head(−>next)^{i−1} −>
        data
        ≤head(−>next)^i −>data) ∧ ptr1−>data≤ptr−>data
        ∧ ptr1−>data< p−>data ∧ j≥1 ∧ ∀ i:1..m−j−1.
        (ptr(−>next)^{i−1} −>data≤ptr(−>next)^i −>data)
        ∨ ptr==NULL ∧ ∀ i:1..j−1.(head(−>next)^{i−1} −>
          data≤
        head(−>next)^i −>data) ∧ ptr1−>data< p−>data
        ∧ j≥1 /* loop invariant */{
      ptr1=ptr;ptr=ptr−>next;j=j+1;
    }
    p−>next=ptr1−>next;ptr1−>next=p;ret=head;
  }
  return ret;
    assertion length(ret,next)==m+1 ∧
      ∀ i:1..m.(ret(−>next)^{i−1} −>data≤ret(−>next)^i −>
        data)
}
```

Fig. 4 Program and assertions of function for inserting a node into an ordered singly-linked list.

图 4 有序单链表的节点插入函数的代码和断言

链表节点的有序性可以用带全称量词的断言来刻画,参看图 4 的函数前后断言,其中 m 是一个逻辑变量或者叫辅助变量. 用全称量词表示链表的有序性时需要有链表的长度信息,断言语言有一个预定义的函数 $length$,用于各种链表. $length(head, next)$是指针 $head$ 顺着 $next$ 域的表长.

通过最强后条件演算,循环前的那个程序点的断言 Q 是:$m==length(head) \wedge j==1 \wedge head->data< p->data \wedge \forall i:1..m-1.\ head(->$

$next)^{i-1}->data \leqslant head(->next)^{i}->data$.

在该程序点产生的验证条件是 Q 蕴涵循环不变式,该验证条件的证明必须在从形状图(如图 3(a)所示)获得 $ptr1==head$, $ptr==ptr1->next$ 和 $m-1\geqslant 0$(该表这时至少有 1 个节点),还有 $ptr==NULL \vee ptr!=NULL$ 之后才能完成. 其他验证条件的证明也需要形状图的支持.

例 2. 本例是快速排序的完整程序和断言,如图 5 所示. 该程序虽不涉及指针型数据,但使用了一维数组类型. 下标表达式的使用也会引起别名,例如在 $i=j$ 时, $a[i]$ 和 $a[j]$ 就互为别名. 下标变量的别名问题比较容易解决,本系统采用的办法是每次对数组 a 的一个下标变量赋值就引入一个大小一样的虚拟新数组 a'(即它仅在验证中使用). 例如,若数组 a 大小是 n,在赋值 $a[j]=e(0\leqslant j\leqslant n-1)$ 之后,该赋值的后断言包括其前断言以及 $\forall i:0..j-1.a'[i]==a[i] \wedge a'[j]==e \wedge \forall i:j+1..n-1.a'[i]==a[i]$.

```
int [10] arr;
int low,up;
int partition(int low1,int up1){
  assertion 0==low∧up==9∧low≤low1
        ∧low1≤up1∧up1≤up
  int k,m,j,temp,ret;
  k=arr[up1];m=low1-1;j=low1;
  while(j!=up1){
    loop_invariant 0==low∧up==9∧low≤low1∧
      low1≤up1∧up1≤up∧low1-1≤m∧
      m≤up1-1∧low1≤j∧j≤up1∧
      ∀i:low1..m.arr[i]≤k∧∀i:m+1..j-1.arr[i]>k∧
      arr[up1]==k
    if(arr[j]≤k){
      m=m+1;temp=arr[m];arr[m]=arr[j];arr[j]=temp;
    }
    j=j+1;
  }
  temp=arr[up1];arr[up1]=arr[m+1];arr[m+1]=temp;
  ret=m+1;return ret;
  assertion 0==low∧up==9∧low≤low1∧low1≤up1∧
    up1≤up∧low1≤ret∧ret≤up1∧
    ∀i:low1..ret-1.arr[i]≤k∧arr[ret]==k∧
    ∀i:ret+1..up1.arr[i]>k
}
void quick_sort(int low2,int up2){
  assertion 0==low∧up==9∧low≤low2∧up2≤up
  int i;
  if(low2<up2){
    i=partition(low2,up2);
    quick_sort(low2,i-1);quick_sort(i+1,up2);
  }
  return;
  assertion 0==low∧up==9∧low≤low2∧
    low2≤up2∧up2≤up∧∀i:low2..up2-1.a[i]≤a[i+1]
}
main(){
  assertion true
  low=0;up=9;quick_sork(low,up);
  assertion 0==low∧up==9∧∀i:low..up-1.a[i]≤a[i+1]
}
```

Fig. 5 Program and assertions of the Quicksort function.

图 5 快速排序完整的程序和断言

该方法使得生成验证条件的演算比较简单,但增加了自动定理证明器的负担. 本例虽然没有给数组置初值,但它不影响对该程序的验证.

另一个特点是本例包括了递归函数的验证,以体现所实现的系统已经有较好的适应能力.

5 相关工作比较

程序验证技术的各种应用体现出它们对程序性质的保证程度,高端是证明程序满足它的功能规范,低端是仅仅发现一些可能的程序缺陷. 通常所说的程序验证器处于高端,它需要自动定理证明器的支持.

自 Reynolds 提出分离逻辑[11]后,指针程序验证的研究进入一个活跃阶段. Smallfoot[8]是一个典型的使用分离逻辑进行符号执行的程序验证工具,其他还有 Verifast[20]等. 我们在先前的研究中也提出一种指针逻辑[21]来用于指针程序的验证. 除了引言已经指出的本文克服访问路径别名方法与分离逻辑方法相比的优点外,本文方法的另一个优点是,不需要程序员提供有关数据结构形状的函数前后条件和循环不变式. 无论是用分离逻辑还是指针逻辑,让普通程序员来写这些断言是一件十分困难的事情.

另外,分离逻辑用分离合取连接词来分隔不同堆块的断言的方式,使得一般需要引入归纳定义的谓词来描述数据结构的节点之间的关系. 而与归纳定义相适应的是用递归方式而不是循环方式来编程,例如文献[22]给出的研究程序验证的源语言连循环语句也没有. 对于各种链表的插入和删除等函数,程序员可能更习惯用循环来编程,本文给出的带上角标的访问路径以及使用全称量词的方式,便于描述各种链表的节点之间的数据关系.

分离逻辑的优势是其适用性较强,它不需要像本文这样禁止编程语言使用指针算术运算和取地址运算等. 分离逻辑技术的众多研究使得它在提高自

动化程度[23]、终止性证明[24]、细粒度的并发[25]、基于锁的并发性[26]和用于 Java 语言[27]等方面有很大拓展.除了用于源程序的验证以外,本文方法在其他方面的应用有待进一步研究.

6 总 结

本文提出一种利用形状图信息来消除访问路径别名,使得指针程序仍然可以用 Hoare 逻辑来进行验证的方法,并利用可满足性模理论求解器 Z3 的支持,开发了一个 PointerC 语言的程序验证器原型,展示了该方法的可行性.

本文方法的局限是程序员只能使用有限的几种形状来编程.下一步考虑怎样以现有的几种形状为基础,构造出各种所需的形状,例如嵌套的形状(例如双向链表的每个节点指向一个单链表)、带有附加链的形状(把形状上满足某个性质的节点链接起来)、增加附加指针的形状(例如在二叉树的每个节点上增加指针形成左孩子右兄弟树).

我们将继续寻找减轻自动定理证明器负担的方法.目前正在实现的是:提供程序员在断言中使用自定义谓词以及表达不同自定义谓词之间关系的机制.仍以易变数据结构为例,允许程序员在断言中使用自定义谓词,除了可以比较简洁地表示节点之间的数据关系外,联系不同谓词的关系等式是相关验证条件的证明依据和提示.

参 考 文 献

[1] Lin Huimin, Zhang Wenhui. Model checking: Theory, methods and applications [J]. Acta Electronica Sinica, 2002, 30(12A): 1907-1912 (in Chinese)
(林惠民,张文辉. 模型检测:理论、方法与应用[J]. 电子学报, 2002, 30(12A): 1907-1912)

[2] Hoare C A R. An axiomatic basis for computer programming [J]. Communications of the ACM, 1969, 12(10): 576-585

[3] Paulson L C. Isabelle: A Generic Theorem Prover [M]. Berlin: Springer, 1994: 1-300

[4] The Coq Development Team. The Coq proof assistant reference manual, Version 8.2 [OL]. [2009-08-01]. http://coq.inria.fr

[5] Aleksandar N, Morrisett G, Shinnar A, et al. Ynot: Dependent types for imperative programs [C] //Proc of the 13th ACM SIGPLAN Int Conf on Functional Programming (ICFP'08). New York: ACM, 2008: 229-240

[6] Barnett M, Rustan M, Leino M, et al. The spec # programming system: An overview [G] //LNCS 3362: Proc of Int Workshop Construction and Analysis of Safe, Secure, and Interoperable Smart Devices (CASSIS 2004). Berlin: Springer, 2004: 49-69

[7] Flanagan C, Leino K R M, Lillibridge M, et al. Extended static checking for Java [C] //Proc of the Conf on Programming Language Design and Implementation (PLDI'02). New York: ACM, 2002: 234-245

[8] Berdine J, Calcagno C, O'Hearn P W. Smallfoot: Modular automatic assertion checking with separation logic [G] //LNCS 4111: Proc of Formal Methods for Components and Objects(FMCO 2005). Berlin: Springer, 2005: 115-137

[9] Distefano D, Parkinson M J. jStar: Towards practical verification for Java [C] //Proc of ACM SIGPLAN Int Conf on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2008). New York: ACM, 2008: 213-226

[10] Carter G, Monahan R, Morris J M. Software refinement with perfect developer [C] //Proc of the 3rd IEEE Int Conf on Software Engineering and Formal Methods(SEFM 2005). Piscataway, NJ: IEEE, 2005: 363-372

[11] Reynolds J C. Separation logic: A logic for shared mutable data structures [C] //Proc of the 17th Annual IEEE Symp on Logic in Computer Science (LICS 2002). Piscataway, NJ: IEEE, 2002: 55-74

[12] Berdine J, Calcagno C, O'Hearn P W. A decidable fragment of separation logic [G] //LNCS 3328: Proc of the 24th Int Conf on Foundations of Software Technology and Theoretical Computer Science(FSTTCS 2004). Berlin: Springer, 2004: 97-109

[13] Moura L D, Bjørner N. Z3: An efficient SMT solver [G] //LNCS 4963: Proc of Conf on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2008). Berlin: Springer, 2008

[14] Li Zhaopeng, Zhang Yu, Chen Yiyun. Shape graph logic and a shape system [OL]. [2012-12-05]. http://ssg.ustcsz.edu.cn/SGL

[15] Liu Gang, Chen Yiyun, Zhang Zhitian. Automatic inference of loop invariant shape graphs [J]. Chinese Journal of Electronic Technology, 2011(8): 4-6 (in Chinese)
(刘刚,陈意云,张志天. 循环不变形状图的自动推断[J]. 电子技术, 2011(8): 4-6)

[16] Necula G. Proof-carrying code [C] //Proc of the 24th ACM Symp on Principles of Programming Languages (POPL'97). New York: ACM, 1997: 106-119

[17] Bornat R. Proving pointer programs in Hoare logic [G] //LNCS 1837: Proc of the 5th Int Conf on Mathematics of Program Construction(MPC 2000). Berlin: Springer, 2000: 102-126

[18] Kroening D, Strichman O. Decision Procedures: An Algorithmic Point of View [M]. Berlin: Springer, 2008

[19] Zhang Zhitian, Li Zhaopeng, Chen Yiyun, et al. The verifier prototype system of PointerC based on the shape graph logic and shape system [OL]. [2012-12-05]. http://ssg.ustcsz.edu.cn/content/shape-graph-logic

[20] Jacobs B, Piessens F. The VeriFast program verifier. TR CW-520 [R]. Flanders, Belgium: Department of Computer Science, Katholieke Universiteit Leuven, 2008

[21] Chen Yiyun, Li Zhaopeng, Wang Zhifang, et al. A pointer logic for safety verification of pointer programs [J]. Journal of Software, 2010, 21(3): 124-137 (in Chinese)
(陈意云, 李兆鹏, 王志芳, 等. 一种用于指针程序验证的指针逻辑[J]. 软件学报, 2010, 21(3): 124-137)

[22] Chin W, David C, Nguyen H H, et al. Automated verification of shape, size and bag properties [C] //Proc of the 12th IEEE Int Conf on Engineering of Complex Computer Systems (ICECCS 2007). Piscataway, NJ: IEEE, 2007: 307-320

[23] Yang H, Lee O, Berdine J, et al. Scalable shape analysis for systems code [C] //Proc of the 20th Int Conf on Computer Aided Verification (CAV 2008). New York: ACM, 2008: 385-398

[24] Brotherston J, Bornat R, Calcagno C. Cyclic proofs of program termination in separation logic [C] //Proc of the ACM SIGPLAN—SIGACT Symp on Principles of Programming Languages (POPL 2008). New York: ACM, 2008: 739-782

[25] Calcagno C, Parkinson M J, Vafeiadis V. Modular safety checking for fine-grained concurrency [G] //LNCS 4634: Proc of the 14th Int Static Analysis Symp (SAS 2007). Berlin: Springer, 2007: 233-248

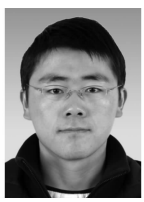
[26] Gotsman A, Berdine J, Cook B, et al. Local reasoning for storable locks and threads [G] //LNCS 4807: Proc of the 5th ASIAN Symp on Programming Languages and Systems (APLAS 2007). Berlin: Springer, 2007: 19-37

[27] Haack C, Hurlin C. Separation logic contracts for a Java-like language with fork/join [G] //LNCS 5140: Proc of the 12th Int Conf on Algebraic Methodology and Software Technology (AMAST 2008). Berlin: Springer, 2008: 199-215

Zhang Zhitian, born in 1987. Received her MSc degree in computer science from the University of Science and Technology of China, Hefei, China, in 2011. Her main research interests include program verification, shape graph logic, etc.

Li Zhaopeng, born in 1978. Received his PhD degree in computer science from the University of Science and Technology of China, Hefei, China, in 2008. Postdoctoral researcher in the school of Computer Science and Technology, the University of Science and Technology of China. Member of China Computer Federation. His main research interests include programming language, program verification, certifying compiler, and automated theorem proving.

Chen Yiyun, born in 1946. Received his MSc degree from the East-China Institute of Computer Technology in 1982. Professor and PhD supervisor in the School of Computer Science and Technology, the University of Science and Technology of China. His main research interests include applications of logic (including formal semantics and type theory), techniques for designing and implementing programming languages and software safety and security.



Liu Gang, born in 1986. Received his MSc degree in computer science from the University of Science and Technology of China, Hefei, China, in 2011. His main research interests include program verification, shape graph logic, and loop invariant inference.