ISSN 1000-9825, CODEN RUXUEW
Journal of Software, Vol.20, No.8, August 2009, pp.2037−2050
doi: 10.3724/SP.J.1001.2009.00572

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel/Fax: +86-10-62562563

用于指针逻辑的自动定理证明器*

王振明[1+], 陈意云[1], 王志芳[2]

[1](中国科学技术大学 计算机科学技术系,安徽 合肥 230026)

[2](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)

Automated Theorem Prover for Pointer Logic

WANG Zhen-Ming[1+], CHEN Yi-Yun[1], WANG Zhi-Fang[2]

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

[2](Software Security Laboratory, Suzhou Institute for Advanced Study, University of Science and Technology of China, Suzhou 215123, China)

+ Corresponding author: E-mail: zmwang4@mail.ustc.edu.cn, http://ssg.ustcsz.edu.cn/

Wang ZM, Chen YY, Wang ZF. Automated theorem prover for pointer logic. *Journal of Software*, 2009,20(8): 2037−2050. http://www.jos.org.cn/1000-9825/572.htm

Abstract: This paper presents a technique for designing theorem prover which mainly based on transformation and substitution for Pointer Logic. The technique realized as a tool called APL is implemented. The APL theorem prover is fully automated with which proofs can be recorded and checked efficiently. The tool is tested on pointer programs mainly about singly-linked lists, doubly-linked lists and binary trees.

Key words: pointer program; pointer logic; verification condition; automated theorem prover; proof checker

摘 要: 提出了一种为指针逻辑设计定理证明器的新技术,该项技术主要是基于变换和替代,已在 APL 的工具中得以实现.APL 自动定理证明器是完全自动的,且其产生的证明可以被有效地记录和检验.已使用关于单链表、双链表和二叉树的指针程序测试了该自动定理证明器.

关键词: 指针程序;指针逻辑;验证条件;自动定理证明器;证明检查器

中图法分类号: TP301 **文献标识码**: A

1 Introduction

With increased complexity of missions, the need for high quality and safety-critical software has increased dramatically over the past few years. Formal methods have been used in the development of many safety-critical systems in the form of formal specification and formal proof of correctness. A wide variety of different logics have been developed for formal methods. Building theorem provers for each of these logics is a massive challenge.

Among all the existing popular theorem provers, PVS[1] is a proprietary system for developing formal specifications, of which the PVS proof checker is one component. Isabelle[2,3] is a generic theorem prover, allowing

* Supported by the National Natural Science Foundation of China under Grant Nos.60673126, 90718026 (国家自然科学基金)
Received 2008-05-25; Revised 2008-08-28; Accepted 2008-10-09

different logics to be defined by users. Nuprl[4] is an interactive proof system for constructive mathematics based on Martin-Löf's Type Theory. Coq[5] is a proof assistant for the Calculus of Inductive Constructions and Coq uses the natural deduction proof style. ACL2[6] is a heavily automated, first order theorem prover specifically designed to support the specification and development of computing systems. As is well known, this area is characterized by its wide variety of proof methods, forms of automated deduction and applications.

This paper concerns with automated theorem proving for Pointer Logic[7], which is an extension of Hoare logic and essentially is a pointer analysis tool. Pointer Logic is designed for PointerC[8] which is a C-like programming language. In Pointer Logic, pointers are classified into effective pointers (those point to objects) and ineffective pointers which contain null pointers and dangling pointers. Pointer Logic collects pointer information including whether a pointer is null, dangling or effective, the equality between effective pointers in a forward manner. And the collected information can then be used to prove that the program satisfies the requests of user-defined safety policies such as type safety and memory safety. To meet the safety requirements of software, some undecidable pointer operations have been restrained in PointerC. Thus we can obtain accurate pointer analysis instead of an imprecise one.

In our earlier work, we designed a certifying compiler called PLCC[9] for PointerC. In our original implementation, not all the verification conditions (VCs) generated by the verification condition generator (VCG) could be proved automatically by the theorem prover embedded in the compiler and those VCs need to be proved interactively by programmers using the proof-assistant tool—Coq. Consequently it made the tool difficult to use by one who is not an expert in Coq. Besides, the early version of PLCC can handle only a few simple programs written in PointerC. This is because the programming language constructs such as pointers, structures and unions are not directly supported by the existing provers, and are often encoded imprecisely by using axioms and uninterrupted functions[10]. To solve those problems, we have developed a new powerful automated theorem prover called APL for the PLCC system.

Our paper makes the following contributions:

1. We present a new technique for designing automated theorem prover which mainly proves proof obligations manipulating pointers;
2. A fully automated theorem prover using this technique has been implemented for Pointer Logic;
3. Machine checkable proofs could be generated, recorded and checked efficiently by this tool;
4. The realization of the automated theorem prover APL makes our earlier certifying compiler PLCC more powerful now.

In this paper, we introduce Pointer Logic, and the design and implementation of our automated theorem prover. The rest of the paper is organized as follows. Section 2 describes Pointer Logic. In Section 3 we give an overview of the whole theorem prover and introduce the verification condition. We describe the verification condition checker, which checks if the verification condition to be proved violates the rules of Pointer Logic in Section 4. In Section 5, we explain how the proofs are generated and recorded. Proof checking is described in Section 6. Section 7 shows the experimental results. Section 8 compares our work with related work and Section 9 concludes the paper.

2 Pointer Logic

Pointer Logic is an extension of Hoare logic and it essentially is a pointer analysis tool. The basic idea of Pointer Logic is to represent memory states by means of sets of pointers. Pointer Logic consists of an assertion language, a set of axioms and inference rules. The interested reader can find a detailed description of Pointer Logic in Refs.[7,9].

Our certifying compiler PLCC supports a source language called PointerC which equipped with both a type system and a logic system. PointerC is a C-like imperative language, which excludes pointer arithmetic and the address-of operation. These restrictions are based on the premise of not affecting the functionality of PointerC and this makes checking more pointer programs statically possible. We use the logic system to help reason the side conditions in the typing rules and then support value-sensitive static checking. In the following, we will give a brief introduction to this logic system called Pointer Logic. Due to the limitation of space, we will not describe the axioms and inference rules of Pointer Logic in the following subsections.

2.1 Conventions and notations

In Pointer Logic, we represent states by means of sets of pointers (or access paths, which are introduced later) and we classify pointers into three kinds: effective pointers (those point to dynamically allocated objects), null pointers and dangling pointers. At any program point, we use Π to denote set of **effective pointers**; use N to denote the set of null pointers; use D to denote the set of dangling pointers. The null pointers and dangling pointers are also called **ineffective pointers**. N and D are used for the purpose of detecting possible memory errors such as null dereference or using an ineffective pointer as actual parameter of function free. The elements of set N and set D are pointers while the elements of Π are sets of pointers when N, D, Π are not empty. We use S_n to denote the elements of Π, the suffix n represents the dimension of Π. Note that the order of S_n in Π does not matter. For example, if $\Pi=\{\{p,q\},\{m,n\}\}$, we let $S_1=\{p,q\}$, $S_2=\{m,n\}$, we call p, q, m, n all effective pointers, and p, q are equal pointers (but not aliases), m, n are equal pointers (but not aliases). According to Point Logic, p and q in S_1 should not be equal to any other pointers in S_2 and vice versa. Next, we introduce the concepts of equality and aliasing of pointers. We say that two pointers are **equal** if their r-values are equal. We say that two pointers are **aliases** if their l-values are equal.

In our Logic, a heap is represented by a directed graph. Each dynamically allocated object is a vertex in a graph. The **access paths** maintain the topological structure by connecting vertices in the graph. Access paths are a special kind of strings that satisfies certain syntactical requirements. Thus we introduce the notation of **prefix**. For example, $p\rightarrow next$ is a prefix of $p\rightarrow next\rightarrow next$, and p is a prefix of $p\rightarrow next\rightarrow next$. Pointer Logic concerns pointer **aliasing** which occurs when two or more access paths refer the same storage location at the same program point. Different access paths are assumed to bound to different storage locations, unless it can be proved that they are bound to the same location (those bound to the same location are aliases) therefore, equality information of effective pointers is needed to deduce the access paths that are bound to the same location.

2.2 Assertion language

In the following figure we show the syntax of the assertion language, which is used to annotate the source program in PointerC. We omitted the definition of boolexp. Actually, it is a Boolean expression, and it can be value TRUE, FALSE, or conjunction, disjunction, negation, and so on. Here ε means null. The definition of Π, N and D is just the same form as we described above. And *lval* can be considered as the pointer or the access path we mentioned in Section 2.1. Let us take $p(\rightarrow next)^3$ as an example to explain the meaning of $lval(\rightarrow id)^{\exp}$. $p(\rightarrow next)^3$ is the abbreviation for $p\rightarrow next\rightarrow next\rightarrow next$. And id is a string over the alphabet [0..9a..zA..Z]. If the effective pointer set or the null pointer set or the dangling pointer set is empty, then $\{\ldots\}_{\Pi}$ or $\{\ldots\}_N$ or $\{\ldots\}_D$ will not appear in the assertion.

$$
\begin{array}{lll}
ass & ::= & bool\,\mathrm{exp} \mid \Psi \mid \neg ass \mid ass \wedge ass \mid (ass) \mid ass \vee ass \mid ass \Rightarrow ass \\
 & & \mid id(lval) \mid \exists id : dom.ass \mid \forall id : dom.ass \\
dom & ::= & \varepsilon \mid \mathrm{exp}..\mathrm{exp} \\
\Psi & ::= & \Pi \wedge N \wedge D \mid \Pi \wedge N \mid \Pi \wedge D \mid N \wedge D \mid \Pi \mid N \mid D \mid ... \\
\Pi & ::= & \{ss\}_{\Pi} \\
N & ::= & \{ls\}_{N} \\
D & ::= & \{ls\}_{D} \\
ss & ::= & ss, set \mid set \\
set & ::= & \{ll\} \\
ll & ::= & ll, lval \mid lval \\
lval & ::= & id \mid lval \rightarrow id \mid lval(\rightarrow id)^{\exp} \\
\mathrm{exp} & ::= & num \mid NULL \mid \mathrm{exp} + \mathrm{exp} \mid \mathrm{exp} - \mathrm{exp} \mid \mathrm{exp} * \mathrm{exp} \mid ... \\
num & ::= & 0 \mid 1 \mid 2 \mid ...
\end{array}
$$

Pointer Logic is designed to be fit to collect pointer information in a forward manner. This information includes whether a pointer is null, dangling or effective, equality relation between effective pointers. The information collected is recorded in sets. From all of the above, one may find that Pointer Logic is a little different from all the other existing logics devised to prove pointer programs, although itself is an extension of Hoare logic. Considering the fact that it is not quantifier free, the pointer information is stored in set and it needs to take many pointer aliasing problems into account, it is a challenging problem to design an automated theorem prover for Pointer Logic.

3 Overview of the APL Theorem Prover

The basic interface that an ATP provides takes as input a formula and returns a Boolean (“true”, “false”) answer. In addition to this basic interface, ATP may generate proofs witnessing the validity of input formulas, this basic capability is essential to techniques such as Proof-Carrying Code (PCC)[11], where the ATP is an untrusted and potentially complicated program and the proof generated by the ATP can be checked efficiently by a simple program. And our implementation is not an exception.

Figure 1 gives an overview of the overall system architecture.

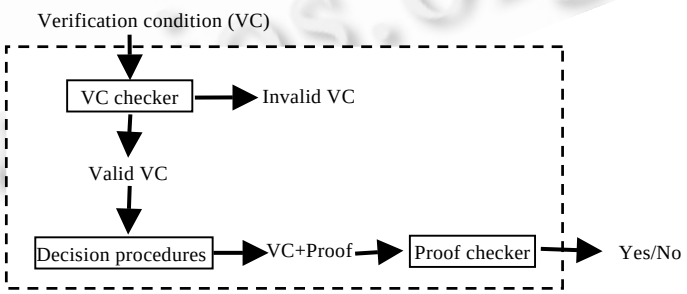


Fig.1 Overall structure of the APL theorem prover

The system takes as input the verification condition generated by a verification condition generator embedded in the front end of the PLCC compiler. The VC is first propagated to the VC checker, which checks if the VC satisfies the rules of Pointer Logic, if not, the invalid VC will not be handled further more by the APL system. The valid one is put to the decision procedures which will generate its corresponding proofs if existed. Then the VC and

its proofs are propagated to the proof checker, which will check if the proofs are right or not. The correctness of our system doesn't depend on the correctness of the theorem prover. Instead, we only need trust the proof checker. The individual components of APL will be described in some details in the subsequent sections.

3.1 Verification condition

In this section we describe the verification condition and introduce an example. First, the form of VC will be demonstrated, and then we present a VC example to help the reader to understand what the goals to be proved look like. The VC has the form as follows:

$$prop \rightarrow prop \tag{e 2.1.0}$$

Each prop above is composed of eight parts, they are: *Quant*, *ΠΠ*, *N*, *D*, *Pred*, *Darr*, *Notsure*, *Env*. The contents of each part are: *Quant* contains the quantifiers, *ΠΠ* contains the effective pointers, *N* contains the null pointers, *D* contains the dangling pointers, *Pred* contains the recursively defined predicates that appear in the annotations, *Darr* contains the dynamic array information, *Notsure* contains the pointers that are represented in the form of forall *k*: *i*..*j*(*P*), where *P* could be predicates or pointers, which have *k* at the exponential position. For example, *Notsure* may look like the form in (e 2.1.1) or (e 2.1.2). And the last part of prop is *Env*, which contains the integer linear arithmetic equalities and inequalities. For instance, the *Env* part may have the form shown in (e 2.1.3).

$$\text{forall } i[0...n].(Tree(p \rightarrow lchild(\rightarrow rchild)^i \rightarrow lchild)) \tag{e 2.1.1}$$

$$\text{forall } k[1...n].(P_i\{\{p(\rightarrow rchild)^k, p(\rightarrow r)^{1+k} \rightarrow lchild\}\}) \tag{e 2.1.2}$$

$$i \geq 1 \ \&\& \ i < n \ \&\& \ j == 0 \tag{e 2.1.3}$$

As the above, the goal we are going to prove will be the form of the following:

$$\begin{aligned} Goal \triangleq \{&Quant_1, \Pi\Pi_1, N_1, D_1, Pred_1, Darr_1, Notsure_1, Env_1\} \rightarrow \\ &\{Quant_2, \Pi\Pi_2, N_2, D_2, Pred_2, Darr_2, Notsure_2, Env_2\} \end{aligned} \tag{e 2.1.4}$$

And

$$Quant \triangleq exists\ n:\ int$$

$$\Pi\Pi \triangleq [] \text{ or } [[p_{11}, p_{12}, ..., p_{1i}], ..., [p_{m1}, p_{m2}, ..., p_{mj}]]$$

$$N \triangleq [] \text{ or } [p_{11}, p_{12}, ..., p_{1i}]$$

$$D \triangleq [] \text{ or } [p_{11}, p_{12}, ..., p_{1i}]$$

$$Pred \triangleq [] \text{ or } [(id_1, [p_{11}, p_{12}, ..., p_{1i}]), ..., (id_n, [p_{m1}, p_{m2}, ..., p_{mj}])]$$

$$Darr \triangleq [] \text{ or } [(p_{11}, p_{12}, \exp_1), ..., (p_{m1}, p_{m2}, \exp_m)]$$

$$Notsure \triangleq [] \text{ or } [(p_1, \exp_{11}, \exp_{12}, \exp_{13}), ..., (p_m, \exp_{m1}, \exp_{m2}, \exp_{m3})]$$

$$Env \triangleq integer\ linear\ arithmetic\ expression$$

In order to give the reader an overall impression of what the VCs look like, a practical example of VC that appears in the program list_rotate.c which is about the rotation operation on lists is given in Fig.2.

As shown in the Fig.2, the first annotation between /*and*/ defines an auxiliary predicate describing a singly-linked list. The second annotation between /*@ and*/ is the pre-condition of the whole function, and the last annotation between /*@ and*/ is the post-condition of the whole function. For lack of space we omit the assertions inserted between the source codes. In this paper, we only consider the proof goals constructed from the verification condition between the /*THE VERIFICATION CONDITION IS: and*/, and do not care about how the verification condition are generated by the verification condition generator (VCG). There are 12 proof goals for the verification condition in the above example. Fig.3 shows some of these proof goals.

We can find some facts about the proof obligations in Fig.3 to be proved: first, the goal that is going to be proved is not quantifier-free as shown in (p 2.2.1) and lots of former experiences indicate that this kind of proof

obligations is not easy to be proved automatically; second, not all the eight parts mentioned above will be used in the VC as shown in (p 2.2.2) and (p 2.2.3). Obviously, which part will be used depends on the program going to be proved; third, the existence of exponent that indicates how many the next exist as shown in (p 2.2.1) makes the VC not easy to prove. Due to all the facts mentioned above and the characteristics of Pointer Logic, the design and implementation of an automated theorem prover for Pointer Logic is different from that are used for the existing provers.

```
Struct list
{
   int data;
   Struct list*next;
};
/*list(Struct list*p)=
N{p}
||
P_i{{p}} && list(p→next)*/
/*@exists n: int.(n≥1 && P{{x}} && forall k[1...n−1].(P_i{{x(→next)^k}})) && P_i{{x(→next)^n}} && N{x(→next)^{1+n}})
||
P_i{{x,p}} && N{→next}
||
N{x,p}*/
Struct list*list_rotate(Struct list*x,Struct list*p)
{if (x!=null)
{p→next=x;
   x=x→next;
   p=p→next;
   p→next=null
return x;
/*THE VERIFICATION CONDITION IS:
P_i{{p,x→next,{x}} && N{p→next}
||
exists n: int.(P_i{{p,x next)^n},{x next)^{n−1}},{x}} && N{p→next} && forall k[2...n−1].(P_i{{x next)^{k−1}}}) && n≥1 && 2≤n)
||
P_i{{x,p}} && N{x→next}
||
N{x,p}
⇒
(exists n: int.(n≥1 && P_i{{x}} && forall k[1...n−1].(P_i{{x(→next)^k}})) && P_i{{p,x(→next)^n}} && N{x(→next)^{1+n}})
||
P_i{{x,p}} && N{x→next}
||
N{x,p}*/
}
/*@exists n: int.(n≥1 && P_i{{x}} && forall k[1...n−1].(P_i{{x(→next)^k}})) && P_i{{p,x(→next)^n}} && N{x(→next)^{1+n}})
||
P_i{{x,p}} && N{x→next}
||
N{x,p}*/
```

Fig.2 An example of VC

$$P_i\{\{p, x \to next\}, \{x\}\} \;\&\&\; N\{p \to next\} \to \quad \text{(p 2.2.1)}$$
$$exists\ n:\ \text{int}.(n \geq 1 \;\&\&\; P_i\{\{x\}\} \;\&\&\; \text{forall } k[1...n-1].(P_i\{\{x(\to next)^k\}\})$$
$$\&\&\; P_i\{\{p, x(\to next)^n\}\} \;\&\&\; N\{x(\to next)^{1+n}\})$$
$$P_i\{\{p, x \to next\}, \{x\}\} \;\&\&\; N\{p \to next\} \to \quad \text{(p 2.2.2)}$$
$$P_i\{\{x, p\}\} \;\&\&\; N\{x \to next\}$$
$$P_i\{\{p, x \to next\}, \{x\}\} \;\&\&\; N\{p \to next\} \to \quad \text{(p 2.2.3)}$$
$$N\{x, p\}$$

Fig.3 Examples of proof obligation

4 VC Checker

There is a good reason for designing such a checker: The verification condition generator generates a large number of VCs for the program being verified, and not all of those VCs are valid according to the properties of Pointer Logic. So there is no need to prove the invalid ones. The role of the VC checker is to check the VCs to be proved if they obey the rules of Pointer Logic before they are propagated to the main theorem proving decision procedures. Some of the rules used by the checker are listed as follows:

These rules are directly derived from Pointer Logic. The meaning of these rules will be explained in the rest of this paragraph. The p in the rules (1) to (14) represents a pointer. The suffix 1, 2 in Π, N and D indicate the corresponding premise or conclusion respectively. S represents a set which consists of effective pointers that are equal. Pointers in set S should be different from each other, and different sets in Π should not have intersection. In Ref.[7], the reader will find the details of Π, N, D and S. The meaning of rule (1) is that if p is in set Π_1 and N_1 at the same time, then we get false. Rule (2) says that if p is in set Π_1 and D_1 at the same time, then we get false. The meaning of rule (3) and rule (4) is similar to that of rule (1) and rule (2). The meaning of rule (14) is that if p and q are in the same set S, and p, q are equal to each other, then we get false. Let us take rule (3) as an example, which indicates that if pointer p in the set N of the premise and it also in the set D of the premise, then we get false. This is because according to Pointer Logic, p should not be a null pointer or a dangling pointer simultaneously.

The advantages of designing this VC checker will be shown by our experiments which will be introduced in Section 6. Let us take the program *list_zip.c* as an example, 14 VCs that violate the rules defined in Fig.4 could be filtered out. The run-time cost of the VC checker itself is small, so it greatly saves the time cost of the whole proving process.

$$\frac{p\in\Pi_1, p\in N_1}{\text{false}}\ (1)\qquad \frac{p\in\Pi_1, p\in D_1}{\text{false}}\ (2)\qquad \frac{p\in N_1, p\in D_1}{\text{false}}\ (3)\qquad \frac{p\in\Pi_2, p\in N_2}{\text{false}}\ (4)\ldots$$

$$\frac{\Pi=[S_1,\ldots,S_i,\ldots,S_j,\ldots,S_n]\ \ S=[p_1,p_2,\ldots,p_l]\ \ p\in S_i\ \ q\in S_i\ \ p\equiv q}{\text{false}}\ (14)$$

Fig.4 Rules used by the VC checker

5 Proof Generation

In this section, we present a new technique for designing theorem prover which mainly based on transformation and substitution for Pointer Logic. Taking the proof obligations to be proved into account, we adopt the method of compositional verification. The original proof goal is divided into small sub-goals, and then each of the new sub-goals is considered. If all of the sub-goals could be proved, then the original goal has a proof. We use all the proofs for the sub-goals to construct the proof of the original goal. In the rest of the section we will give some details about how the goal is splitted into sub-goals and how the sub-goals could be proved by using transformation and substitution.

The original goal usually has the form of (e 2.1.4) as described in Section 3.1, and the sub-goals for it are shown in Fig.5.

The reason for breaking the original Goal into these 7 sub-goals above is that, according to Pointer Logic, the information of pointers in the premise and conclusion of the proof goal should be equal if the goal could be proved. This is a little different from the traditional rules. In traditional rules, if the premise contains all the information of the conclusion, then the premise implies the conclusion. The fact we require strictly that the information in the premise and conclusion should be equal lies on the essence of Pointer Logic and the presentation of the program states (sets of pointers). This restriction is required only in proving the original goal, but not the sub-goals of it. The

equality of the information of pointers will be checked by the proof checker. $Goal_1$ to $Goal_6$ are about pointers and $Goal_7$ is about integer linear arithmetic. The Quant part is treated specially by the decision procedures we have implemented and does not appear as a sub-goal in Fig.5.

$$Goal_1 \triangleq \{Quant_1, \Pi_1, N_1, D_1, Pred_1, Darr_1, Notsure_1, Env_1\} \rightarrow \{\Pi_2\}$$
$$Goal_2 \triangleq \{Quant_1, \Pi_1, N_1, D_1, Pred_1, Darr_1, Notsure_1, Env_1\} \rightarrow \{N_2\}$$
$$Goal_3 \triangleq \{Quant_1, \Pi_1, N_1, D_1, Pred_1, Darr_1, Notsure_1, Env_1\} \rightarrow \{D_2\}$$
$$Goal_4 \triangleq \{Quant_1, \Pi_1, N_1, D_1, Pred_1, Darr_1, Notsure_1, Env_1\} \rightarrow \{Pred_2\}$$
$$Goal_5 \triangleq \{Quant_1, \Pi_1, N_1, D_1, Pred_1, Darr_1, Notsure_1, Env_1\} \rightarrow \{Darr_2\}$$
$$Goal_6 \triangleq \{Quant_1, \Pi_1, N_1, D_1, Pred_1, Darr_1, Notsure_1, Env_1\} \rightarrow \{Notsure_2\}$$
$$Goal_7 \triangleq \{Quant_1, \Pi_1, N_1, D_1, Pred_1, Darr_1, Notsure_1, Env_1\} \rightarrow \{Quant_2, Env_2\}$$

Fig.5 The sub-goals for the original goal

$$\frac{\begin{array}{c}VC \triangleq (\{Quant_1, \Pi_1, N_1, D_1, Pred_1, Darr_1, Notsure_1, Env_1\},\\ \{Quant_2, \Pi_2, N_2, D_2, Pred_2, Darr_2, Notsure_2, Env_2\})\\ \Pi_2 \subseteq \Pi_1\end{array}}{\begin{array}{c}VC' \triangleq (\{Quant_1, \Pi_1 - \Pi_2, N_1, D_1, Pred_1, Darr_1, Notsure_1, Env_1\}\\ \{Quant_2, N_2, D_2, Pred_2, Darr_2, Notsure_2, Env_2\})\end{array}} \quad (R1)$$

$$\frac{\begin{array}{c}VC \triangleq (\{Quant_1, \Pi_1, N_1, D_1, Pred_1, Darr_1, Notsure_1, Env_1\},\\ \{Quant_2, \Pi_2, N_2, D_2, Pred_2, Darr_2, Notsure_2, Env_2\})\\ Env_1 \rightarrow Env_2\end{array}}{\begin{array}{c}VC' \triangleq (\{Quant_1, \Pi_1, N_1, D_1, Pred_1, Darr_1, Notsure_1\},\\ \{Quant_2, \Pi_2, N_2, D_2, Pred_2, Darr_2, Notsure_2\})\end{array}} \quad (R2)$$

$$\frac{VC_1 \triangleq \{Quant_1, \Pi_1, N_1, D_1, Pred_1, Darr_1, Notsure_1, Env_1\} \ \{p,q\} \in \Pi_1 \ lval_p \ in \ VC_1}{[q/p]lval_p \ in \ VC_1} \quad (R3)$$

$$\frac{\text{forall } k: \ m..n\exp \in Notsure \ m < n}{Notsure' = Notsure - \{\text{forall } k: \ m..n\exp\}} \quad (R4)$$

$$\frac{\text{forall } k: \ m..n\exp \in Notsure \ \ \text{forall } k: \ n+1..l\exp \in Notsure}{Notsure' = Notsure + \{\text{forall } k: \ m..l\exp\} - \{\text{forall } k: \ m..n\exp, \text{forall } k: \ n+1..l\exp\}} \quad (R5)$$

In this paragraph, we will give an introduction to some of the rules used by the decision procedures. These rules are shown in the above figure. Rule $R1$ means that if Π_2 is a subset of Π_1, then the original VC can be transformed to VC', and $\Pi_1-\Pi_2$ means Π_1 minus Π_2, as we know, Π_1 and Π_2 are sets. Rule $R2$ says that if Env_1 implies Env_2 then we can get VC' from VC. Rule $R3$ describes the substitution of prefix of *lval*. The $lval_p$ represents all the *lvals* that have prefix p in VC_1, if $\{p,q\}$ is an element of Π_1, then we can substitute all the appearance of p in $lval_p$ with q. Rule $R4$ and $R5$ are used to help proving $Goal_6$ as shown in Fig.5. Rule $R4$ says that if the upper bound n is small than the lower bound m for an element of *Notsure*, then we can delete this element from *Notsure* and get *Notsure'*. Rule $R5$ combines two elements of *Notsure* into one to get *Notsure'*, by adding the new one to set *Notsure* and deleting the original two from *Notsure*. There are also kinds of rules designed for predicates, such as list, dlist and tree. Due to spaces limitations, we will not describe all of those rules used by our decision procedures.

Let us see an example. When proving the above goal (p 4.1.1) in Fig.6, we first break this goal into two sub-goals (p 4.1.2) and (p 4.1.3). Now we need to prove two sub-goals: It is easy to prove (p 4.1.2), because the Π part of the conclusion is a subset of the Π part of the premise. When proving (p 4.1.3), predicate *tree*(*res*→*rchild*→*lchild*) is handled first, we try to find out if *tree*(*res*→*rchild*→*lchild*) is in the *Pred* part of the premise, the testing result is that it is not in the *Pred* set of the premise, then we find pointers those are equal to

pointer *res*→*rchild* from the *Π* part of the conclusion. We find that *p* and *res*→*rchild* are in the same set, thus we substitute *res*→*rchild* with *p*, afterwards we check if *tree*(*p*→*lchild*) in the *Pred* part of the premise, if the answer is yes, then we prove the predicate *tree*(*res*→*rchild*→*lchild*). Next, we prove predicate *tree*(*res*→*rchild*→*rchild*), the proof process of this predicate is similar to the previous one, and we can prove it too. Lastly, we prove predicate *tree*(*res*→*lchild*), we find it in the *Pred* part of the premise. So we prove this predicate too. Finally, all of the predicates in the conclusion can be proved. So (p 4.1.3) can be proved. And the proof of each predicate consists of the proof of (p 4.1.3). If any of the predicates in the conclusion can not be proved, then (p 4.1.3) can not be proved either.

$$P_i\{\{res \to rchild, p\},\{res\}\} \;\&\&\; N\{p \to lchild\} \;\&\&\; tree(p \to rchild) \;\&\&\; tree(res \to lchild)$$
$$\to \qquad (p\ 4.1.1)$$
$$P_i\{\{res \to rchild, p\},\{res\}\} \;\&\&\; tree(res \to rchild \to lchild) \;\&\&\; tree(res \to rchild \to rchild) \;\&\&\; tree(res \to lchild)$$

$$P_i\{\{res \to rchild, p\},\{res\}\} \;\&\&\; N\{p \to lchild\} \;\&\&\; tree(p \to rchild) \;\&\&\; tree(res \to lchild)$$
$$\to \qquad (p\ 4.1.2)$$
$$P_i\{\{res \to rchild, p\},\{res\}\}$$

$$P_i\{\{res \to rchild, p\},\{res\}\} \;\&\&\; N\{p \to lchild\} \;\&\&\; tree(p \to rchild) \;\&\&\; tree(res \to lchild)$$
$$\to \qquad (p\ 4.1.3)$$
$$tree(res \to rchild \to lchild) \;\&\&\; tree(res \to rchild \to rchild) \;\&\&\; tree(res \to lchild)$$

Fig.6 An example for predicate

$$P_i\{\{p, x \to next\},\{x\}\} \;\&\&\; N\{p \to next\}$$
$$\to \qquad (\text{new p } 2.2.1)$$
$$(1 \geq 1 \;\&\&\; P_i\{\{p, x(\to next)^1\},\{x\}\} \;\&\&\; \text{forall } k[1..\sim 1+1].(P_i\{\{x(\to next)^k\}\}) \;\&\&\; N\{x(\to next)^{1+1}\})$$

As another example, let us consider the proof obligations in Fig.3. For (p 2.2.1), all appearances of n in the conclusion part are instantiated to integer 1 according to our proof strategies. Then we only need to prove the resulting proof goal: (new p 2.2.1). Now we break (new p 2.2.1) into 4 sub-goals, the reader will find out that each of these sub-goals is easy to prove. As to (p 2.2.2), there is no proof for it, and (p 2.2.3) will be filtered out by the VC checker.

5.1 Proof recording

Necula strongly advocated that theorem provers ought to generate easily checkable proofs in his paper[12]. Just as he suggested, in our work some effort has been spent in trying to produce human readable results, which allows the user to examine the generated proofs. Furthermore, our proof file can be handled easily by machine. Our implementation is inspired by the work of Wong[13−15] and the work of Geoffrey Norman Watson[16].

The type proof defined in SML for recording proofs in Fig.7 will help the reader to catch on how the proofs are recorded. Usually, a proof has 11 fields. Field name records the name of the proof. Field *f*1, *f*2 record the information of *N*, *D* and Darr. Field *f*3, *f*4 and *f*5 record the predicate information of *Pred*. Field *f*6, *f*7 record the information of *Π*. Field *f*8, *f*9 and *f*10 record the information of *Notsure*. As to $Goal_7$ that we mentioned in Fig.5, which involves the *Env* part, we only record the name of the proof of it. To a user, recording proof is a feature which can be enabled or disabled. The printing of proof details can be enabled or disabled too. When the printing of the proof detail is disabled, the printout only contains the name of proof for each sub-goal. The right part of Fig.7 gives the proof of (new p2.2.1) mentioned in Section 5.

The process of recording proof and generating proof files can be divided into three stages:

1. Recording proofs of the sub-goals;
2. Generating a whole proof by combining all the above proofs in stage 1;

3. Outputting the proof generated in stage 2 to a text file or the terminal.

```
type proof ={name: string,
             f1: lval list,
             f2: lval list,
             f3: (id *(lval list)) list,
             f4: (id *(lval list)) list,
             f5: (id *(lval list)) list,
             f6: lval list list,
             f7: lval list list,
             f8: (lval *exp*exp) list,
             f9: (lval *exp*exp*exp) list,
             f10: (lval *exp*exp*exp) list}
```

```
| ************************************************************
| The proof is:
| ************************************************************
| pai_in                                  pred_null
| f1: f2: f3: f4: f5:                     f1: f2: f3: f4: f5: f6: f7: f8: f9: f10:
| f6:{x},{p, x(→ next)∧(1)}
| f7: f8: f9: f10:                        darr_null
|                                         f1: f2: f3: f4: f5: f6: f7: f8: f9: f10:
| en_trans_equal
| f1: x(→ next)∧(2)                       notsure_compare_null
| f2: x(→ next)∧((1)+(1))                 f1: f2: f3: f4: f5: f6: f7:
| f3: f4: f5: f6: f7: f8: f9: f10:        f8: forall k :[1..(1)−(1)].(P_i{{x(→ next)∧(k)}})
|                                         f9: f10:
| di_null
| f1: f2: f3: f4: f5:                     env_true
| f6: f7: f8: f9: f10:                    f1: f2: f3: f4: f5: f6: f7: f8: f9: f10:
```

Fig.7 Definition of type proof and proof script for (new p 2.2.1)

6 Proof Checking

In Ref.[15], the authors design a proof checker for three main reasons: first, the mechanically generated formal proofs are usually very long; second, the theorem provers are usually very complex so that it is very difficult to verify their correctness; third, the programs that a user develops while doing the proof are very often too complicated and do not have simple mapping to the sequence of inferences performed by the system. Our intention of designing such a checker is different. The fundamental reason is that taking the characteristics of Pointer Logic into account, designing a theorem prover without a proof checker will be difficult to guarantee the soundness of the system, because the implication between the premise and conclusion of the proof goal is based on the equality of the information of pointers. A proof checker will be suitable to deal with this situation. At the beginning we are clear about it, and this makes our implementation of theorem prover is not as complex as the other existing ones, we let the proof checker take the task of ensuring the soundness of the theorem prover. In our implementation, a proof goal may have a proof generated by the theorem prover, but the proof must be checked by the proof checker.

To a user, the checker is an ML function which does not read a proof file directly, it takes the verification condition and its corresponding proof as inputs and reports back with either a success which means the proofs are correct or a failure which means the opposite. The checking process is done in a loop. The body of the loop is a case analysis on the values of a string type, actually, it is the name field mentioned above in the definition of type proof of Section 5.1. We evaluate the name filed of all proofs, and then attempt to match it to one of the patterns in a given order. When a successful match occurs, a corresponding expression performing deletion is evaluated. Let's take the pattern *en_eq_trans_in* as an example, when this pattern matches, the information in the field of *f*1 will be deleted from the *N* field in the premise, the information in the field of *f*2 will be deleted from the *N* field in the conclusion. And this pattern says that after doing substitution to the *N* field of the conclusion by using equalities in the *Env* field of the conclusion, the resulting *N* field of the conclusion is a subset of the *N* field of the premise. There are 55 patterns for the checker now. Each pattern handles some of the fields from *f*1 to *f*10. It may happen that the case patterns do not cover all cases, that is, there can be values that do not match any of the patterns. In that case, SML reports a warning stating that the matching is incomplete at compile-time. And at runtime, if an

unmatched expression occurs, an exception is raised.

For example, the checking process of the proof script for (new p 2.2.1) in Fig.7 is as follows: first, pattern "*pai_in*" is matched, and the corresponding pointer information stored in *f*6 is deleted from the Π fields of the premise and conclusion of (new p 2.2.1), then pattern "*en_trans_equal*" is matched, and the information stored in *f*1 and *f*2 is deleted from the *N* fields of the premise and conclusion of (new p 2.2.1) respectively. Here, we need to do some transformations to the pointer $p \rightarrow next$ in the set *N* of the premise of (new p 2.2.1). Because p and $x \rightarrow next$ are equal according to Pointer Logic, we substitute *p* of pointer $p \rightarrow next$ with $x \rightarrow next$ and get $x \rightarrow next \rightarrow next$, which is same to $x(\rightarrow next)^2$. For the patterns "*di_null*", "*pred_null*" and "*darr_null*", nothing is done. When "*notsure_compare_null*" is matched, the information stored in field *f*8 is deleted from the *Notsure* part of the conclusion of (new p 2.2.1). At last, the pattern "*env*_true" is matched, and the *Env* parts of the premise and conclusion of (new p 2.2.1) are empty set. Thus, after matching all patterns in the proof and deleting corresponding information from the premise and conclusion of the verification (new p 2.2.1), we can easily find that all parts of the premise and conclusion are empty. This means that the pointer information in the premise is equal to that in the conclusion, that is, the proof is correct.

7 Performance

We have implemented our prover APL in SML and embedded the APL theorem prover in the certifying compiler PLCC which takes the annotated program in PointerC as input. Then the VCs generated by the VCG component of PLCC are propagated to the APL. A proof file is generated for each source file. Table 1 lists the name of program for testing, the number of VC for each program, the proof file size and the run time. The first 8 programs dealing with operations such as create, reverse, destroy, insert, delete on singly-linked list. The 9th and 10th test cases are about insertion and deletion operations on doubly-linked list, which indicate that cyclic data structures could be handled by the APL prover. We also tried our prover with a number of examples about left-rotate, right-rotate, delete, traverse operations on trees as shown in the last four lines. Although none of these examples is large enough to provide hard data on how much more effective a user of our tool can be than he would be without it, they served to test the robustness of our implementation and allow us to present some preliminary figures.

Table 1 Performance of the APL prover

Program name	Number of VC	Proof size (KB)	Time (ms)
list_create.c	7	2	78
list_reverse.c	14	2	281
list_get_element.c	7	3	109
list_destroy.c	9	1	94
list_concat.c	12	4	344
list_insert.c	10	5	156
list_delete.c	10	5	93
list_zip.c	79	2	125
dlist_insert.c	10	6	219
dlist_delete.c	8	5	94
l_rotate.c	2	1	31
r_rotate.c	2	1	47
tree_delete.c	24	5	203
tree_traversal.c	4	1	47

All timings were carried out on a HP 500 with 1.73 GHz processor running Windows XP. The results were promising as the figures in the table showed, the proof file size was small and the run time for each program was less than 1 second. We were able to prove all of the proof obligations of the programs listed in Table 1 with single APL invocations. Compared to an earlier manual attempt, the efficiency and power of the PLCC system have been

greatly improved by embedding the APL theorem prover. We hope to explore this source of examples further so as to be able to present more concrete results.

Let us take the operation delete on singly-linked list (*list_delete.c*) as an example. PALE[17] spent 1.36 seconds compared to 0.093 seconds for APL, and PALE didn't generate a proof file while APL did. The experiments show that our prover does work in practice for non-trivial data structures, and with time and space requirement which are as good as or better than those related approaches with similar goals[17,18].

8 Related Work

The approach of verifying programs with pointers by using theorem provers has been described by a number of authors. Usually the programs are written in C or a sub-set of C programming language. For lack of space we only mention the most relevant ones. Caduceus[19] takes the advantage of Why[20] and various existing provers (e.g. PVS[1], Coq[5] and Simplify[21]) can be used to prove the verification condition. But Caduceus can only handle type-safe C programs and the annotated C program must be first translated to an intermediate language. In our implementation we do not need to do any translation, the interaction takes place at the level of source code. ASTRÉE[22] is a program analyzer which can analyze C programs with pointers, structures and arrays, and it excludes union types and dynamic memory allocation. It's bounded to a specific prover and able to fully prove automatically the absence of runtime errors in real-life large programs. But producing a correctness proof for complex software will take a few hours and use a lot of memory. ACE[23] uses a theorem proving tool called Stanford Temporal Prover (STeP)[24] and adopts the static assertion checking technique to verify MISRA-C program. ACE uses the technique of compositional verification, which helps in proving higher level properties by splitting the task of verification into small, manageable program slices. We employ this technique in our implementation too. When we are proving the initial goal, the goal will be broken into some small sub-goals first, then we prove each of the sub-goals, if all of the sub-goals have a proof, then we could get a proof for the initial goal. The size and complexity of the sub-goals is small, thus the properties of sub-goals can be obtained and proved easily. Another recent work KeY-C[25,26] based on KeY[27] is a tool for deductive verification of programs written in a subset of C. KeY is an interactive theorem proving environment and KeY-C use the calculus of C Dynamic Logic which is based on first-order logic extended with a type system. Compared with our approach, systems that employ interactive proof assistants have the advantage that even difficult proofs can be discharged with enough manual effort, but our point of view is that only systems that use fully automated theorem prover take chance of widespread acceptance. And we used the calculus of Pointer Logic which is an extension of Hoare logic. The features of Pointer Logic make it strong in manipulating pointer related information, such as reasoning with pointer aliasing.

9 Conclusion

The work presented in this paper is targeted towards proving functional correctness of sequential program code and adopts the automated theorem-proving approach to formal verification. We proposed a method for designing automated theorem prover handling proof obligations with pointers and a prototype implementation for Pointer Logic. We have demonstrated that several small but non-trivial programs in our C-like programming language can be verified efficiently by using a special theorem prover called APL, and reduced the problem to validity of formulas in Pointer Logic.

In a near future, we plan to add support for arrays and floating-point arithmetic and improve the Cooper's algorithm used currently to make the APL system more usable and improve its efficiency and its expressiveness.

Acknowledgement We benefited from fruitful discussions with the members of the PLCC project. We thank Dr. Chunxiao Lin, Dr. Long Li, Dr. Zhaopeng Li and Dr. Baojian Hua for their very helpful comments on drafts of this article.

References:

[1] Owre S, Rushby JM, Shankar N. PVS: A prototype verification system. In: Proc. of the 11th Int'l Conf. on Automated Deduction (CADE). Saratoga: Deepak Kapur, 1992. 748−752.

[2] Paulson LC, Nipkow T. Isabelle: A generic theorem prover. Berlin: Springer-Verlag, LNCS 828, 1994.

[3] Paulson LC. The Isabelle reference manual. Technical Report 283, University of Cambridge Computer Laboratory, 1997.

[4] Allen SF, Constable RL, Eaton R, Kreitz C, Lorigo L. The Nuprl open logical environment. In: Proc. of the 17th Int'l Conf. on Automated Deduction. LNAI 1831, Springer-Verlag, 2000. 170−176.

[5] The Coq Development Team. The Coq proof assistant reference manual. The Coq release v7.1, 2001.

[6] Kaufmann M, Moore JS. Design goals for ACL2. In: Proc. of the 3rd Int'l School and Symp. on Formal Techniques in Real Time and Fault Tolerant Systems. Kiel: Published by Christian-Albrechts-Universitat, 1994. 92−117.

[7] 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.

[8] Hua BJ, Ge L. The definition of PointerC programming language. http://ssg.ustcsz.edu.cn/lss/doc/index.html

[9] 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 Joint IEEE/IFIP Symp. on Theoretical Aspects of Software Engineering (TASE 2007). 2007. 127−138.

[10] Cook B, Kroening D, Sharygina N. Cogent: Accurate theorem proving for program verification. In: Proc. of the Conf. on Computer Aided Verification (CAV 2005). 2005.

[11] Necula G. Proof-Carrying code. In: Proc. of the 24th Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL'97). 1997.

[12] Necula G, Lee P. Proof generation in the Touchstone theorem prover. In: McAllester D, ed. Proc. of the CADE-17. Springer-Verlag, 2000.

[13] Wong W. Recording HOL proofs. Technical Report 306, University of Cambridge Computer Laboratory, 1993.

[14] Wong W. A proof checker for HOL. Technical Report 389, University of Cambridge Computer Laboratory, 1996.

[15] Wong W. Recording and checking HOL proofs. In: Schubert ET, Windley PJ, Alves-Foss J, eds. Proc. of the Higher-Order Logic Theorem Proving and Its Applications. Lecture Notes in Computer Science, Springer-Verlag, 1995,971:353−368.

[16] Geoffrey Norman Watson. Proof representation in theorem provers. Technical Report 98-13, The University of Queensland, 1998.

[17] Möller A, Schwartzbach MI. The pointer assertion logic engine. In: Proc. of the 8th PLDI. 2001. 221−231.

[18] Jensen JL, Jørgensen ME, Klarlund N, Schwartzbach MI. Automatic verification of pointer programs using monadic second-order logic. In: Proc. of the Programming Language Design and Implementation (PLDI'97). 1997.

[19] Filliâtre JC, Marché C. Multi-Prover verification of C programs. In: Davies J, Schulte W, Barnett M, eds. Proc. of the Formal Methods and Software Engineering, 6th Int'l Conf. on Formal Engineering Methods (ICFEM 2004). Seattle: Springer-Verlag, 2004, 3308:15−29.

[20] The why verification tool. http://why.lri.fr/

[21] Detlefs D, Nelson G, Saxe JB. Simplify: A theorem prover for program checking. Journal of the ACM, 2005.

[22] Blanchet B, Cousot P, Cousot R, Feret J, Mauborgne L, Monniaux D, Rival X. The ASTRÉE analyzer. In: Proc. of the European Symp. on Programming (ESOP 2005). Edinburgh, 2005,3444:21−30.

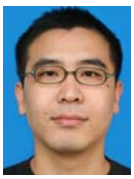
[23] Sharma B, Dhodapkar SD, Ramesh S. Assertion checking environment (ace) for formal verification of C programs. In: Proc. of the 21st Int'l Conf. on Computer Safety, Reliability and Security (SAFECOMP 2002). London: Springer-Verlag. 2002. 284−295.

[24] Bjorner N, *et al*. The Stanford Temporal Prover User's Manual. Stanford University, 1998. 284−286.

[25] Mürk O, Larsson D, Hähnle R. A dynamic logic for deductive verification of C programs with KeY-C. C/C++ Verification Workshop at Integrated Formal Methods (IFM). Technical Report ICIS-R07015, Radboud University, 2007. 43−58.

[26] Mürk O, Larsson D, Hähnle R. KeY-C: A tool for verification of C programs. In: Proc. of the Int'l Conf. on Automated Deduction (CADE). Bremen: Springer-Verlag, 2007,4603:385−390.

[27] Beckert B, Hähnle R, Schmitt P. Verification of Object-Oriented Software: The Key Approach. Springer-Verlag, 2006.



WANG Zhen-Ming was born in 1981. He is a Ph.D. candidate at the Department of Computer Science and Technology, USTC. His current research areas are programming language theory, program verification, automated theorem proving theory and application.

CHEN Yi-Yun was born in 1946. He is a professor and doctoral supervisor at the Department of Computer Science and Technology, USTC and a CCF senior member. His research areas are design and implementation of programming language, software security and program verification.

WANG Zhi-Fang was born in 1982. He is a Ph.D. candidate at the Department of Computer Science and Technology, USTC. His current research areas are software safety and security, logic and formal semantics and certifying compiler.