

An Extension to Pointer Logic for Verification

Zhifang Wang Yiyun Chen Zhenming Wang Wei Wang Bo Tian

*Department of Computer Science and Technology, University of Science and Technology of China
Hefei, 230026, China*

*Suzhou Institute for Advanced Study, University of Science and Technology of China
Suzhou, 215123, China*

{zfwang7, zmwang4, wqsh}@mail.ustc.edu.cn yiyun@ustc.edu.cn tianbo@ustc.edu

Abstract

The safety of pointer programs is an important issue in high-assurance software design, and their verification remains a major challenge. Pointer Logic has been proposed to verify basic safety properties of pointer programs in our previous work, but still lacks support for a wide range of pointer programs. In this work, we present an extension to Pointer Logic by: 1) introducing modular reasoning to scale better on programs involving function calls; 2) allowing pointer arithmetic to take more advantage of pointers in programming. Moreover, to demonstrate that Pointer Logic is a useful approach to verification, we implement a tool – plcc to automatically verify a range of non-trivial programs, including basic operations on singly-linked lists, trees, circular doubly-linked list, dynamic arrays etc..

1. Introduction

Nowadays software is widely used in various areas, and potential errors such as dangling dereferences and memory leaks might lead to serious damages. As a result, high-assurance software is highly desirable. On the other hand, software verification promises to overcome the current software crisis of bug-ridden software. Pointers (references) are widely used in modern programming languages for its efficiency and flexibility, but they are also the source of errors because of aliasing.

In recent years, program verification through formal reasoning to predict the effect that a program produces when run on a physical machine, has made great progress in software engineering. PCC (Proof Carrying Code) [16] pioneered by Necula has given light into the verification paradigm which can be applied in actual software development. In order to utilize the PCC paradigm for verification, various approaches *e.g.*, [15, 1, 23, 22] are proposed. However, many of these approaches based on formal reasoning need

a theory to describe the semantics of the programming language, and introduce concepts such as heaps, stacks *etc.* explicitly that are not part of the syntax. Although useful in reasoning about program properties, they complicate the reasoning significantly. On the contrary, Pointer Logic explores the possibility to directly handle the program safety properties of a given state through sets of access paths (pointers) instead of introducing stacks or heaps explicitly – the reasoning is all syntactic.

Pointer Logic has been reported on in two earlier works [6, 7]. However, existing Pointer Logic rules for function calls associate the entire heap with each invocation, and the function body might be verified more than once, thus it is difficult to scale the current approach to more programs. We believe that modular treatment of the heap will allow the implementation to scale better on larger pieces of code. Moreover, it lacks support for pointer arithmetic, which hinders wide usage of Pointer Logic for program verification. In this work, we take a leap forward in addressing the aforementioned issues by introducing modular reasoning and extending Pointer Logic with pointer arithmetic. We consider the case of a C-like imperative language–PointerC, which includes pointer arithmetic but excludes the address-of operation. Although in [6, 7] a previous version of plcc(v1.5) was released, it can only verify several simple programs and verification conditions generated need proving manually in COQ [8]. Therefore, we redesigned plcc(Current version is v2.0) to verify a range of pointer programs without any help from interactive theorem provers – it’s fully automated. As long as the programs are annotated with loop invariants, pre- and post-conditions, the annotations are then automatically checked for validity by calculating the strongest post-conditions according to Pointer Logic rules and verifying a number of induced implications internally.

Our contributions can be summarized as follows:

1. We introduce modular reasoning in Pointer Logic by splitting the program states via reachability of objects,

which enables us to verify more programs involving function calls and helps efficient implementation of Pointer Logic rules;

2. We extend Pointer Logic with limited pointer arithmetic to verify the properties such as absence of array-out-of-bound errors, absence of null pointers involved in pointer arithmetic *etc.*, besides the common guarantees of memory safety;
3. To demonstrate that Pointer Logic is a useful approach to verification, we implemented plcc(v2.0) to automatically verify a range of non-trivial pointer programs, including basic operations on lists, trees, circular doubly-linked lists, dynamic arrays. A bright feature of plcc(v2.0) is that it can generate human readable proof outlines when the verification is successful.

The rest of the paper is organized as follows. Section 2 defines the PointerC language. Section 3 introduces Pointer Logic briefly. Section 4 presents the modular reasoning in Pointer Logic. Section 5 extends Pointer Logic with pointer arithmetic. Section 6 shows the experimental results. Section 7 presents the related work and Section 8 concludes our work.

2. The PointerC language

PointerC is a C-like imperative language which excludes the address-of operation (&). The syntax is given in Figure 1. The notation \bar{z} denotes a sequence of z 's. Note that we

<i>Program</i>	p	::=	$\overline{stdec} \overline{fndec}$
<i>Struct. Decl.</i>	$stdec$::=	struct id { \overline{vdecl} };
<i>Func. Decl.</i>	$fndec$::=	t id ($\overline{t id}$) { $\overline{vdecl} stms$ }
<i>Var. Decl.</i>	$vdecl$::=	$t id$;
<i>Type</i>	t	::=	bool int struct id * $t []$ *
<i>Statements</i>	$stms$::=	$stm stms$ ϵ
<i>Statement</i>	stm	::=	$l=e$; $l=\mathbf{malloc}(\mathit{struct id})$;
			$l=\mathbf{calloc}(t, e)$; $\mathbf{free}(e)$;
			if (be) { $stms$ } else { $stms$ }
			while (be) { $stms$ }
			$id(\bar{e})$; $l=id(\bar{e})$;
<i>Expression</i>	e	::=	n NULL l $-e$ $e+e$...
<i>Bool Exp.</i>	be	::=	true false $e==e$...
			$be \wedge be$ $be \vee be$ $\neg be$
<i>Left Val.</i>	l	::=	id $l \rightarrow id$ $l(\rightarrow id)^e$
<i>Number</i>	n	::=	0 1 2 ...

(Note: $l(\rightarrow id)^e$ is only allowed to appear in assertions)

Figure 1. Definition of the PointerC language.

extend the type with $t [] *$ to support dynamic array declarations and add a system call `calloc`. Currently we only allow pointer arithmetic on dynamic arrays. Every successful

call of `calloc(t, n)` will return a dynamic array of type $t [] *$ with length n . Also `malloc` and `free` functions are defined as system calls and meet the basic requirements of safe programs. For example, every call to `malloc` always successfully returns and the objects allocated are not overlapped in the heap. The type system is deferred to [21].

3. The Pointer Logic

In this section, we briefly introduce Pointer Logic. The details are deferred to [7]. The basic idea is to represent memory states by means of sets of pointers. Every dynamically allocated object is denoted by a *valid pointer set* π in which the r-values of the pointers are the same while l-values are distinct. A pointer is *valid* if and only if it points to a dynamically allocated object. A set of π sets representing all the objects in the heap is denoted by Π . Besides, there are two special sets: an \mathcal{N} set for null pointer set and a \mathcal{D} set for dangling pointer set which are used for the purpose of catching memory errors such as null pointer dereferences and dangling pointer dereferences *etc.*. As the pointers in the same π set, the pointers in \mathcal{N} or \mathcal{D} are also with distinct l-values. Note the representation captures the heap structure exactly, for every object is described by a set π and the pointers in these sets maintain the relationships between objects.

For example, in Figure 2, the representation of Π is

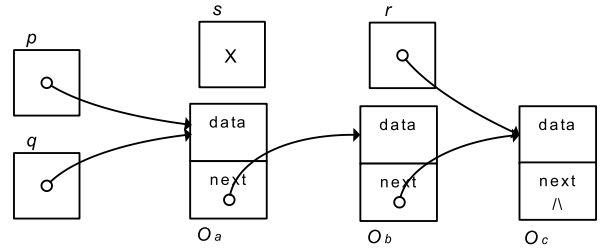


Figure 2. An example of Π , \mathcal{N} , and \mathcal{D} .

$\{\{p, q\}, \{p \rightarrow next\}, \{p \rightarrow next \rightarrow next, r\}\}$, \mathcal{N} is $\{r \rightarrow next\}$, and \mathcal{D} is $\{s\}$ (The cross in s represents meaningless locations and “ \wedge ” means null). In Π , $q \rightarrow next$ does not show up in set $\{p \rightarrow next\}$ because $q \rightarrow next$ and $p \rightarrow next$ have the same l-value. In Pointer Logic, $\{p, q\}$ denotes object O_a , $\{p \rightarrow next\}$ denotes object O_b , while $\{p \rightarrow next \rightarrow next, r\}$ denotes object O_c . As a result, the Π , \mathcal{N} , and \mathcal{D} sets capture the topological structure of the heap exactly, the information of which is useful for program verification.

3.1. The assertion language

In this section, we define the assertion language used for program annotation. The syntax is given in Figure 3. Not

<i>Assertion</i>	$\mathcal{A} ::= be \mid (\mathcal{A}) \mid \mathcal{A} \wedge \mathcal{A} \mid \mathcal{A} \vee \mathcal{A}$ $\mid \mathcal{A} \Rightarrow \mathcal{A} \mid id(l) \mid \Psi \mid \neg \mathcal{A}$ $\mid \exists id : d. \mathcal{A} \mid \forall id : d. \mathcal{A}$
<i>Domain</i>	$d ::= \mathbf{N} \mid e..e$
<i>Pointer Set</i>	$\Psi ::= \Pi \mid \mathcal{N} \mid \mathcal{D} \mid \mathcal{C}$
<i>Valid Ptr. Sets</i>	$\Pi ::= \Pi \wedge \{ls\} \mid \{ls\}$
<i>Null Ptr. Set</i>	$\mathcal{N} ::= \{ls\}_N$
<i>Dangling Ptr. Set</i>	$\mathcal{D} ::= \{ls\}_D$
<i>Dynamic Arr. Sets</i>	$\mathcal{C} ::= \mathcal{C} \wedge \langle l, l, e \rangle \mid \langle l, l, e \rangle$
<i>Lval Set</i>	$ls ::= ls, l \mid l$

Figure 3. The assertion language.

all the sentences derived by this grammar can be assertions here: (i) Pointer sets can be operands of conjunction and disjunction operators, but can not be operands of the negation operator; (ii) The *l-value* in a set must be pointers, and an *l-value* with type *int* or *bool* should not appear in the set. Here we add \mathcal{C} to support pointer arithmetic reasoning which will be discussed in section 5.

Since the assertion designed is to describe memory states, the conjunction of pointers sets is different from that in propositional logic. When an assertion contains pointer sets, contraction ($A \Rightarrow A \wedge A$), weakening ($A \wedge B \Rightarrow A$) and the distributive law ($(A \wedge B) \wedge C \Rightarrow (A \wedge C) \wedge (B \wedge C)$) are not sound.

The assertion language can be used to describe widely used data structures such as singly-linked lists, circular doubly-linked lists, trees *etc.*. For example, we may use a recursive definition for singly-linked lists:

$$list(p) \triangleq \{p\}_{\mathcal{N}} \vee (\{p\} \wedge list(p \rightarrow next))$$

in which the node of the list is defined as:

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

The definition $list(p)$ says that a list might be empty, or might consist of a head pointed to by p and a sub-list pointed to by $p \rightarrow next$. Also we might use a descriptive definition:

$$list(p) \triangleq \exists n : \mathbf{N}. (\forall k : 0..n-1 \{p(-\rightarrow next)^k\}) \wedge \{p(-\rightarrow next)^n\}_{\mathcal{N}}$$

which means that a singly-linked list is comprised of n elements. When $n = 0$, it just denotes an empty list. The advantage of this definition is that we can obtain accurate alias relationship because integer coefficients in symbolic access paths denote positions in data structures. In our current implementation, both representations are well supported.

4. Modular reasoning

In this section, we introduce the frame rule and the rule for function call. Existing Pointer Logic rules (see [7]) for

function calls associate the entire heap with each invocation, and the function body might be verified more than once. However, a program often accesses part of the heap instead of the whole heap. When we annotate a function, its precondition only describes part of the heap which may be modified by the function body. Thus modular treatment of heaps for function call is a necessity. Furthermore, modular reasoning helps efficient implementation of Pointer Logic rules.

4.1. Objects and reachability

To make the reasoning more uniform, we introduce objects and reachability in this section.

In Pointer Logic every dynamically allocated object O is represented by a set of pointers pointing to O . Here we extend the concept about objects by assuming that every pointer in \mathcal{N} or \mathcal{D} points to different *virtual objects*. This makes sense because the following rules hold in Pointer Logic:

$$\{p_1, p_2, \dots, p_n\}_N = \{p_1\}_N \wedge \{p_2\}_N \wedge \dots \wedge \{p_n\}_N \quad (1)$$

$$\{p_1, p_2, \dots, p_n\}_D = \{p_1\}_D \wedge \{p_2\}_D \wedge \dots \wedge \{p_n\}_D \quad (2)$$

Thus we can imagine that they are pointing to some virtual objects in the heap.

DEFINITION 4.1 (REACHABILITY) *An object, represented by a set S , either a π set, or $\{x\}_N$, or $\{x\}_D$, is reachable from pointer p , if and only if there exists pointer $q \in S$ such that $\exists r. r \in closure(q) \wedge prefix(p, r)$. The relationship of reachability is denoted by $Reach(p, S)$.*

The two functions *closure* and *prefix* are defined in [7], in which $closure(p)$ returns all the pointers that are with the same *l-value* as p , and $prefix(p, r)$ decides whether pointer p is a prefix of r . Based on definition 4.1, we can define a function to describe objects reachable from a set of pointers. Let $S = \{p_1, \dots, p_n\}$ be a set of pointers, we define:

$$R(S) \triangleq \bigcup_{i=1}^n \Pi_{p_i} \wedge \bigcup_{i=1}^n \mathcal{N}_{p_i} \wedge \bigcup_{i=1}^n \mathcal{D}_{p_i} \quad (3)$$

where $\Pi_{p_i} \triangleq \{\pi \mid \pi \in \Pi \wedge Reach(p_i, \pi)\}$; $\mathcal{N}_{p_i} \triangleq \{q \mid q \in \mathcal{N} \wedge \exists r. r \in closure(q) \wedge prefix(p_i, r)\}$; $\mathcal{D}_{p_i} \triangleq \{q \mid q \in \mathcal{D} \wedge \exists r. r \in closure(q) \wedge prefix(p_i, r)\}$.

4.2. Inference rules

First, we concentrate on the assignment rules from [7]. Existing rules for a destructive update have to perform a substitution operation on all the sets of Π , \mathcal{N} and \mathcal{D} . Now that we have the R function, we only have to update part of

sets. For an assignment $p = q$, let $R(\{p, q\}) = \Pi_{\{p, q\}} \wedge \mathcal{N}_{\{p, q\}} \wedge \mathcal{D}_{\{p, q\}}$ be the objects reachable from p and q , and $\Psi_{left} = (\Pi - \Pi_{\{p, q\}}) \wedge (\mathcal{N} - \mathcal{N}_{\{p, q\}}) \wedge (\mathcal{D} - \mathcal{D}_{\{p, q\}})$ be the unreachable. Assume after assignment, $R(\{p, q\})$ turns into $R'(\{p, q\})$ which could be obtained through the functions in [7]. Thus the rule for assignment would be:

$$\{R(\{p, q\}) \wedge \Psi_{left}\} p = q \{R'(\{p, q\}) \wedge \Psi_{left}\} \quad (4)$$

In this way efficient implementation of the assignment rules can be obtained because only the reachable part of the heap is modified.

Similarly, the frame rule is as follows:

$$\frac{\{R(S_1)\} Stms \{R(S_1)'\}}{\{R(S_1) \wedge R(S_2)\} Stms \{R(S_1)' \wedge R(S_2)\}} \quad (5)$$

where S_1 and S_2 are two sets of root pointer variables and $R(S_i) = \Pi_{S_i} \wedge \mathcal{N}_{S_i} \wedge \mathcal{D}_{S_i}$ ($i = 1, 2$) satisfying $S_1 \cap S_2 = \emptyset \wedge \Pi_{S_1} \cap \Pi_{S_2} = \emptyset$.

The rule for function call bears a similar form. We assume the formal parameters of function f are x_1, \dots, x_n , the actual parameters are y_1, \dots, y_n , and its precondition and postcondition are f_{pre} and f_{post} which are parameterized by f 's formal parameters. Note that the reachable objects from the actual parameters of a function are possibly influenced by the function body. As a result, we only have to split the memory states to obtain the local heaps the functions might update:

$$\frac{R(\{y_1, \dots, y_n\}) \Rightarrow f_{pre}[y_1, \dots, y_n/x_1, \dots, x_n]}{\{\Pi \wedge \mathcal{N} \wedge \mathcal{D}\} f(y_1, \dots, y_n) \wedge \Psi_{left}} \quad (6)$$

In rule (6), Ψ_{left} is analogous to the one in rule (4) and $f_{pre}[y_1, \dots, y_n/x_1, \dots, x_n]$ instantiates the formal parameters in function f , which indicates that the function has a local view that only includes objects reachable from its actual parameters – unreachable remain unchanged. Hence when a function is called, it operates on part of the heap. This rule resembles the frame rule in the sense that the effect of a function call on a larger heap can be obtained from its effect on a subheap.

Due to space limitations, we omit the rule for $ret = f(y_1, \dots, y_n)$.

5. Pointer Logic with pointer arithmetic

The advantage of pointer arithmetic is its flexibility and convenience for the programming, such as accessing array elements using the pointer itself as a loop control variable. Moreover, highly efficient assembly code can be obtained

by using pointer arithmetic. However, flexible pointer arithmetic might bring array-out-of-bound error when dereferencing a pointer pointing to some element of an array. There are cases when null pointers or dangling pointers are involved in pointer arithmetic, while it is difficult to detect via type checking. The goal of extending Pointer Logic with pointer arithmetic is to verify pointer program properties related to the correct use of dynamically allocated memory such as absence of null pointers or dangling pointers involved in pointer arithmetic, absence of array-out-of-bound errors, besides the common guarantees of absence of null dereferences, memory leaks *etc.*. Here, we study the reasoning in Pointer Logic by allowing limited pointer arithmetic as a probe.

Since Pointer Logic collects pointer information which is about the pointers pointing to heap blocks, we only allow pointer arithmetic on pointers pointing to one-dimensional dynamic arrays which are allocated by $\text{calloc}(T, n)$. Function calloc is defined as a system call where T is the type of elements and n is the length of the array. An example of using calloc and pointer arithmetic is shown in Figure 4.

```
T[] *q;    T *p,*r;
q=calloc(T,10); p=q+5; r=p+3;
```

Figure 4. An example for pointer arithmetic.

In Pointer Logic, when $p \neq q$ (we denote by \equiv *syntactic equivalence*), we use a triple $\langle p, q, \text{offset} \rangle$ to denote pointers pointing to a dynamic array, where p is the *inner pointer* with type T^* , q is the *base pointer* with type $T[]^*$, and offset indicates the position in the dynamic array, *i.e.*, $p = q + \text{offset} * \text{sizeof}(T)$. In this case, the triple expresses the fact that pointer p points to position offset in a dynamic array q . When $p \equiv q$, the triple $\langle p, q, \text{offset} \rangle$ just means the length of the dynamic array p is offset . Note that the triple we define records the information about pointers. For example, in Figure 4, after an assignment $p = q + 5$, the triple for p is $\langle p, q, 5 \rangle$; after another assignment $r = p + 3$ the triple for p is the same, while the triple for r is $\langle r, q, 8 \rangle$.

In order to simplify the reasoning, we require that all pointers pointing to dynamic arrays be root variables, and that the elements of a dynamic array be of a simple type without pointer fields, otherwise the form of the pointer won't be uniform and therefore will greatly complicate the reasoning. However, these constraints can be relieved by introducing more general and uniform representations of pointers and rewriting more universal Pointer Logic rules.

5.1. Axioms and inference rules

In this section, we introduce the axioms and rules involving pointer arithmetic. In accordance with the rules in [7], we try to make the minimum changes by adding a new kind

of set \mathcal{C} (See Figure 3) which is made up of a set of triples describing pointers pointing to dynamic arrays. When two pointers point to non-dynamic-array structures, the rules are the same as in [7] since the program won't modify any elements in \mathcal{C} . As a result, the rules in [7] can be treated as a special case when \mathcal{C} is empty or \mathcal{C} stays the same. In essence, \mathcal{C} is a loose form describing pointers – every base pointer in \mathcal{C} is valid, every inner pointer is either valid or dangling, the state of which depends on the value *offset*. If *offset* is within the scope of the corresponding array, then the pointer is valid, else it is dangling. Hence triples classify themselves, and there is no need to classify the triples in \mathcal{C} into more sets just as Π, \mathcal{N} and \mathcal{D} sets do. Note that inner pointers won't point to elsewhere, *i.e.*, they won't appear in Π, \mathcal{N} or \mathcal{D} . However, the pointers with dynamic array type must appear in Π, \mathcal{N} or \mathcal{D} , because a dynamic array as a whole can be viewed as an object in the heap.

To ease notation in the extended rules, we introduce some helper functions and shortcuts shown in Figure 5. Notation $x.eq$ defined in [7] returns the equivalent set

$$\begin{aligned}
\mathcal{C} \setminus p &\triangleq \{ \langle p', q'', e \rangle \mid \langle p', q', e \rangle \in \mathcal{C} \wedge \exists q''. q'' \in q'.eq \wedge \\
&\quad (\forall p''. p'' \in \text{closure}(p) \Rightarrow \neg \text{prefix}(p'', q'')) \} \\
\mathcal{C} - p &\triangleq \{ \langle p', q', e \rangle \mid \langle p', q', e \rangle \in \mathcal{C} \wedge p' \notin \text{closure}(p) \} \\
\mathcal{C} + \langle p, q, n \rangle &\triangleq \mathcal{C} \cup \{ \langle p, q, n \rangle \} \\
\mathcal{C} - \{ p_1, \dots, p_n \} &\triangleq (\dots (\mathcal{C} - p_1) \dots - p_n) \\
p <: \mathcal{C} &\triangleq \exists \langle p', q', e \rangle \in \mathcal{C} : (p' \in p.eq \vee q' \in p.eq) \\
p <: \mathcal{R} &\triangleq \text{closure}(p) \cap \mathcal{R} \neq \emptyset \\
\mathcal{R} - p &\triangleq \{ q \mid q \in \mathcal{R} \wedge q \notin \text{closure}(p) \} \\
\mathcal{R} + p &\triangleq \mathcal{R} \cup \{ p \} \\
p <: \Pi &\triangleq p.eq \in \Pi \\
\Pi + p &\triangleq \Pi \cup \{ \{ p \} \} \\
\text{base}(p) &\triangleq q \text{ where } \exists e : \langle p, q, e \rangle \in \mathcal{C} \\
\text{index}(p) &\triangleq e \text{ where } \exists q : \langle p, q, e \rangle \in \mathcal{C}
\end{aligned}$$

Figure 5. Shortcuts and helper functions.

pointer x is in. $\mathcal{C} \setminus p$ deals with substitutions for every base pointer q' which is prefixed by an alias of p , with another pointer q'' . $\mathcal{C} - p$ just removes the triple describing pointer p . $p <: \mathcal{C}$ means that p is either an inner pointer or a base pointer. Meta variable \mathcal{R} denotes a set of pointers.

Because of the constraints we mentioned above, the rules are simpler than we may have thought. The most important of all, all the rules in [7] can be reused. However, the introduction of \mathcal{C} requires more rules to deal with dynamic array assignment. We only highlight the differences.

1) Pointer p and q are pointing to non-dynamic-array structures:

$$\frac{p \not<: \mathcal{C} \wedge q \not<: \mathcal{C}}{\{\Pi \wedge \mathcal{N} \wedge \mathcal{D} \wedge \mathcal{C}\} p=q \{ \Pi' \wedge \mathcal{N}' \wedge \mathcal{D}' \wedge \mathcal{C} \}} \quad (7)$$

In rule (7) the assignment has no effect on \mathcal{C} and the $\Pi' \wedge \mathcal{N}' \wedge \mathcal{D}'$ in postcondition is obtained through the corresponding rules $\{\Pi \wedge \mathcal{N} \wedge \mathcal{D}\} p=q \{ \Pi' \wedge \mathcal{N}' \wedge \mathcal{D}' \}$ in [7]. As a result, we inherit the rules from [7] with little modification, and most memory errors can be caught by using the rules in [7]. In the following rules, we assume that $\Pi' \wedge \mathcal{N}' \wedge \mathcal{D}'$ be obtained this way.

2) Pointer p and q are of dynamic array type:

$$\frac{(p <: \mathcal{N} \vee p <: \mathcal{D}) \wedge q <: \Pi \wedge q <: \mathcal{C}}{\{\Pi \wedge \mathcal{N} \wedge \mathcal{D} \wedge \mathcal{C}\} p=q \{ \Pi' \wedge \mathcal{N}' \wedge \mathcal{D}' \wedge \mathcal{C} \}} \quad (8)$$

$$\frac{((p <: \Pi \wedge q <: \Pi \wedge q <: \mathcal{C}) \vee (p <: \Pi \wedge p <: \mathcal{C} \wedge q <: \mathcal{N})) \wedge \text{no_leak}(p)}{\{\Pi \wedge \mathcal{N} \wedge \mathcal{D} \wedge \mathcal{C}\} p=q \{ \Pi' \wedge \mathcal{N}' \wedge \mathcal{D}' \wedge \mathcal{C} \setminus p \}} \quad (9)$$

Rule (8) and (9) are analogous to rule (7) in that the modification of the Π, \mathcal{N} , and \mathcal{D} part is obtained through the rules in [7]. The *no_leak* function defined in [7] ensures no memory leaks can occur.

3) Pointer q is of dynamic array type, and p is of its elements' type:

$$\frac{p \not<: \Pi \wedge p <: \mathcal{C} \wedge q <: \Pi \wedge q <: \mathcal{C}}{\{\Pi \wedge \mathcal{N} \wedge \mathcal{D} \wedge \mathcal{C}\} p=q+e \{ \Pi \wedge \mathcal{N} \wedge \mathcal{D} \wedge (\mathcal{C} - p + \langle p, q, e \rangle) \}} \quad (10)$$

$$\frac{(p <: \mathcal{N} \vee p <: \mathcal{D} \vee p <: \Pi \wedge p \not<: \mathcal{C}) \wedge q <: \Pi \wedge q <: \mathcal{C} \wedge \{ \Pi \wedge \mathcal{N} \wedge \mathcal{D} \} p=\text{NULL} \{ \Pi' \wedge \mathcal{N}' \wedge \mathcal{D}' \}}{\{\Pi \wedge \mathcal{N} \wedge \mathcal{D} \wedge \mathcal{C}\} p=q+e \{ \Pi' \wedge \mathcal{N}' - p \wedge \mathcal{D}' \wedge (\mathcal{C} + \langle p, q, e \rangle) \}} \quad (11)$$

The intermediate assertion $\Pi' \wedge \mathcal{N}' \wedge \mathcal{D}'$ is obtained through the corresponding rule in [7].

4) Pointer p and q are both of dynamic-array-element type for assignment $p=q+e$:

$$\frac{p \not<: \Pi \wedge p <: \mathcal{C} \wedge q <: \mathcal{C} \wedge q \not<: \Pi}{\{\Pi \wedge \mathcal{N} \wedge \mathcal{D} \wedge \mathcal{C}\} p=q+e \{ \Pi \wedge \mathcal{N} \wedge \mathcal{D} \wedge (\mathcal{C} - p + \langle p, \text{base}(q), \text{index}(q) + e \rangle) \}} \quad (12)$$

$$\frac{(p <: \mathcal{N} \vee p <: \mathcal{D} \vee p <: \Pi \wedge p \not<: \mathcal{C}) \wedge q <: \mathcal{C} \wedge q \not<: \Pi \wedge \{ \Pi \wedge \mathcal{N} \wedge \mathcal{D} \} p=\text{NULL} \{ \Pi' \wedge \mathcal{N}' \wedge \mathcal{D}' \}}{\{\Pi \wedge \mathcal{N} \wedge \mathcal{D} \wedge \mathcal{C}\} p=q+e \{ \Pi' \wedge \mathcal{N}' - p \wedge \mathcal{D}' \wedge (\mathcal{C} + \langle p, \text{base}(q), \text{index}(q) + e \rangle) \}} \quad (13)$$

5) Pointer p is of dynamic-array-element type:

$$\frac{p <: \mathcal{C}}{\{\Pi \wedge \mathcal{N} \wedge \mathcal{D} \wedge \mathcal{C}\} p=\text{NULL} \{ \Pi \wedge \mathcal{N} + p \wedge \mathcal{D} \wedge (\mathcal{C} - p) \}} \quad (14)$$

6) The above rules for pointer assignment only concern about the modification of pointer sets. Often, the precondition is Ψ accompanied by assertion Q and some l -values in Q may contain pointer prefixes. Here we just give the corresponding rule for rule (12) and (13), and the rest are omitted for space limitations.

$$\{\Psi \wedge Q\} p=q + e \{\Psi' \wedge Q(r \leftarrow r[p'/p])\} \quad (15)$$

where

$$p' \triangleq \text{if } base(p) \equiv base(q) \text{ then } q + (index(p) - index(q))$$

$$\text{else if } p \equiv q \text{ then } p - e \text{ else } base(p) + index(p)$$

and Ψ' is obtained through the corresponding rule (12) or (13). Note that $Q(r \leftarrow r[p'/p])$ means that for every pointer r in Q , if r has a prefix that has the same l -value as p , then we replace the prefix with p' .

7) The rules for `calloc` and `free` are given below:

$$\frac{\{\Pi \wedge \mathcal{N} \wedge \mathcal{D}\} p = \text{NULL} \{\Pi' \wedge \mathcal{N}' \wedge \mathcal{D}'\}}{\{\Pi \wedge \mathcal{N} \wedge \mathcal{D} \wedge \mathcal{C}\} p = \text{calloc}(T, n)} \quad (16)$$

$$\{(\Pi' + p) \wedge (\mathcal{N}' - p) \wedge \mathcal{D}' \wedge (\mathcal{C} \setminus p + \langle p, p, n \rangle)\}$$

$$\frac{p <: \Pi \wedge p <: \mathcal{C} \wedge \{\Pi \wedge \mathcal{N} \wedge \mathcal{D}\} \text{free}(p) \{\Pi' \wedge \mathcal{N}' \wedge \mathcal{D}'\}}{\{\Pi \wedge \mathcal{N} \wedge \mathcal{D} \wedge \mathcal{C}\} \text{free}(p) \{\Pi' \wedge \mathcal{N}' \wedge \mathcal{D}' \cup \{p_1, \dots, p_n\} \wedge (\mathcal{C} - \{p, p_1, \dots, p_n\})\}} \quad (17)$$

Note that in rule (16), pointer p must appear in Π , \mathcal{N} or \mathcal{D} , because p must be of dynamic array type. Thus it is safe to use the rules in [7] to obtain an intermediate state $\Pi' \wedge \mathcal{N}' \wedge \mathcal{D}'$. Adding $\langle p, p, n \rangle$ to \mathcal{C} just describes the length of the dynamic array p points to is n , which is to be used for array-out-of-bound check. The rule for `free` is based on the following assumptions: (i) the dynamic array is pointed to by p_1, \dots, p_n ; (ii) p is of dynamic array type; (iii) p_1, \dots, p_n are of its elements' type.

5.2. Case study

In this section, we show how Pointer Logic extended with pointer arithmetic can be used in verification by illustrating an example with its proof outline shown in Figure 6. Note that dereferencing only occurs in *if* statement, and we guarantee that the dereferencing is safe – where p points to is within the length of the array. From $\langle p, a, i \rangle$ we know p points to the i th position in the array, and $i \geq 0 \wedge n \geq 1 \wedge i < n$ ensures that $i \geq 0 \wedge i < n$ holds.

6. Experimental evaluation

Our approach has been efficiently implemented in a tool called `plcc`, and now its newest version is 2.0. Compared to

Test case	Code size (Byte)	Proof outline size (KB)	Time(s)
listReverse	405	10.9	0.062
listCreate	494	12.6	0.110
listDestroy	389	6.03	0.031
listGetElement	839	17.6	0.281
listInsertNth	891	33.6	0.547
listDeleteNth	843	30.5	0.656
listZip	1,051	70.9	0.203
listRotate	564	12.5	0.125
listConcat	584	17.9	0.141
dlistInsertNth	1,085	59.1	2.390
dlistDeleteNth	1,061	47.1	2.812
treeRightRotate	531	8.55	0.031
treeLeftRotate	531	8.55	0.031
treeDeleteNode	1,502	56.6	0.406
treeTraverse	325	4.28	0.031
treeCopy	493	13.3	0.047
treeDispose	316	5.85	0.016
maxMinValue	489	15.9	0.062
printDArray	366	10.0	0.047

Table 1. Experimental results.

the previous version of `plcc`(v1.5) (see [6, 7]), `plcc`(v2.0) has the following advantages: (i) `plcc`(v2.0) is fully automated, while the verification conditions generated by `plcc`(v1.5) need proving manually in COQ [8]; (ii) With support for modular reasoning and pointer arithmetic, `plcc`(v2.0) can verify a range of programs while `plcc`(v1.5) can only handle several simple programs; (iii) `plcc`(v2.0) automatically generates a human readable proof outline file for each program when the verification is successful. The proof outline file is of great use for us to debug programs. `plcc`(v2.0) not only verifies the basic safety properties of correct programs, but also detects errors such as possible memory leaks, null dereferences, dangling pointer dereferences, trying to free null pointers or dangling pointers. Thus `plcc`(v2.0) can be used as a tool for statically checking errors.

We have assessed our calculus on a number of pointer programs, as shown in Table 1. In all examples verified by `plcc`, we are ensured that no memory errors could occur. All measurements were performed on a machine using a 1.73GHz Intel PentiumM CPU with 1GB memory. We have verified commonly used operations on singly-linked lists, circular doubly-linked lists, trees, dynamic arrays. The verification time of each program is less than 10 seconds, which is insignificant compared to the time required to design an algorithm and annotate the code. The source code and all the examples' proof outline files are available in reference [21].

```

int max, min;
 $\{\{a\} \wedge \langle a, a, n \rangle \wedge n \geq 1\}$  /* — precondition */
void max_min_value(int*[] a, int n) {
  int *p, *end;
   $\{\{a\} \wedge \{p, end\}_D \wedge \langle a, a, n \rangle \wedge n \geq 1\}$ 
  p = a + 0; end = a + n;
   $\{\{a\} \wedge \langle p, a, 0 \rangle \wedge \langle end, a, n \rangle \wedge \langle a, a, n \rangle \wedge n \geq 1\}$ 
  /* — loop invariant */
   $\{\exists i : N. \{a\} \wedge \langle p, a, i \rangle \wedge \langle end, a, n \rangle \wedge \langle a, a, n \rangle \wedge i \leq n \wedge i \geq 0 \wedge n \geq 1\}$ 
  while (p < end) {
     $\{\exists i : N. \{a\} \wedge \langle p, a, i \rangle \wedge \langle end, a, n \rangle \wedge \langle a, a, n \rangle \wedge i \geq 0 \wedge n \geq 1 \wedge i < n\}$ 
    if (*p > max) { max = *p; } else if (*p < min) { min = *p; }
     $\{\exists L_0 : int. \exists i : N. \{a\} \wedge \langle p, a, i \rangle \wedge \langle end, a, n \rangle \wedge \langle a, a, n \rangle \wedge i \geq 0 \wedge n \geq 1 \wedge i < n \wedge$ 
     $(*p > L_0 \wedge max = *p \vee *p \leq max \wedge *p < L_0 \wedge min = *p \vee *p > min \wedge max \geq *p)\}$ 
    p = p + 1;
     $\{\exists L_0 : int. \exists i : N. \{a\} \wedge \langle p, a, i+1 \rangle \wedge \langle end, a, n \rangle \wedge \langle a, a, n \rangle \wedge i \geq 0 \wedge n \geq 1 \wedge i < n \wedge$ 
     $(*(p-1) > L_0 \wedge max = *(p-1) \vee *(p-1) \leq max \wedge *(p-1) < L_0 \wedge min = *(p-1) \vee *(p-1) > min \wedge max \geq *(p-1))\}$ 
  }
   $\{\{a\} \wedge \langle p, a, n \rangle \wedge \langle end, a, n \rangle \wedge \langle a, a, n \rangle \wedge n \geq 1\}$ 
  return;
   $\{\{a\} \wedge \langle a, a, n \rangle \wedge n \geq 1\}$ 
}
 $\{\{a\} \wedge \langle a, a, n \rangle \wedge n \geq 1\}$  /* — postcondition */

```

Figure 6. Maximum and minimum value of a dynamic array.

7. Related work

Programming with pointers is difficult and risky. This has motivated a huge body of work concerned with analyzing pointer programs and verifying their properties. The most common way is to use Hoare logic to reason about pointer programs, *e.g.*, [4, 13]. However, it's difficult for the traditional Hoare approach to be automated. Besides, there are various extensions to Hoare logic for pointer verification, among which the most novel idea is separation logic. Separation logic [10, 17] is an extension of Hoare logic to reason about pointer structures through separating conjunction of domain disjointness. Recent work towards automation includes [2, 11]. In [2], an experimental tool called SMALLFOOT is implemented based on symbolic execution with separation logic [3]. It supports annotations for concurrency which is out of the scope of our approach, but lacks support for reasoning about pointer aliasing when positions are involved. In some circumstances, SMALLFOOT cannot verify the program using the rolling and unrolling technique alone, thus it uses a collection of inductive hypotheses, which means extra work and might be an obstacle for automatic checking – different data structures may need different hypotheses to help with the proof. In [11], a technique known as rippling is used for automatic assertion refinement, but yet there is no experimental evaluation.

In pointer program verification, the closest approach to

ours is the one advocated by Jonkers [12]. There, the memory state is made of the set of all access paths in the data structures together with an equivalence relation (aliasing) on them. It had been further studied by Deutsch [9] to develop a may-alias algorithm. Recent publications include the work by Venet [20], Bozga *et al.* [5] and Rinetzky *et al.* [18]. Venet follows the earlier work of [12, 9], and proposes an analysis based on abstract interpretation that discovers potential sharing relationships among the data structures. Bozga *et al.* develop a weakest precondition calculus (an alias logic) for Jonkers' storeless model with a Hoare-like proof system. Rinetzky *et al.* focus on local reasoning on Jonkers' storeless model – an interprocedural analysis is described where objects are treated specially when they separate the “local heap” that can be mutated by a procedure from the rest of heap. However, these works based on Jonkers' model suffer from the fact that their representation of memory states is redundant and costly, which can be a great obstacle to program verification. On the other hand, Pointer Logic not only enjoys most of the merits of being a storeless semantic approach, but also relieves its limitations and thus furthers the practical use of storeless approach.

In addition, shape analysis is also used for pointer program verification. PALE (Pointer Assertion Logic Engine) [14] is an automatic verification tool which encodes programs in Pointer Assertion Logic (PAL) – an extension of second order monadic logic on graph types which are tree-

shaped data structures with extra pointers. In PAL programs are restricted to updating only the underlying tree structure, therefore PAL is more applicable to programs manipulating graph types than to unrestricted imperative programs. In Pointer Logic, this is not the problem because our approach is based on sets of access paths describing the heap structure. Moreover, it is natural to adapt our approach for alias analysis, which seems to be an advantage over PAL.

TVLA (Three Valued Logic Analyzer) [19] is based on three valued logic to automatically abstract memory configurations. It is more automatic than ours in a sense that TVLA analyzes programs with only pre- and post-conditions, while our approach often requires the user to provide loop invariants. But still in TVLA the user may be required to provide some instrumentation predicates. Furthermore, because TVLA performs fixpoint iterations on abstract descriptions of the store, our approach using invariants is faster than TVLA.

8. Conclusion

This paper presents an extension of Pointer Logic for pointer program verification. First, we provide a method to deal with modular reasoning by introducing reachability to split program states. Then we extend Pointer Logic rules when limited pointer arithmetic is allowed in PointerC. Finally, we implement a tool – plcc(v2.0) to automatically verify a number of programs, including basic operations on lists, trees, circular doubly-linked lists. For the future, we are planning to apply our technique on compile-time garbage collection in Java. Furthermore, as the current Pointer Logic rules lack the ability to relate two states before and after a function call, the work to extend Pointer Logic with stamps is undergoing.

9. Acknowledgements

We would like to thank the anonymous referees for useful comments on a previous draft of this paper. This research is based on work supported in part by grants from National Natural Science Foundation of China under Grant (No.60673126, No.90718026) and Intel China Research Center. Any opinions, findings and conclusions contained in this document are those of the authors and do not reflect the views of these agencies.

References

[1] A. W. Appel. Foundational proof-carrying code. In *Proc. of LICS'01*, Washington: IEEE Computer Society, 2001, pp. 247–258.

[2] J. Berdine, C. Calcagno, and P.W. O’Hearn. Smallfoot: modular automatic assertion checking with separation logic. In *Proc. of FMCO’05*, LNCS 4111, pp. 115–137, 2005.

[3] J. Berdine, C. Calcagno, and P.W. O’Hearn. Symbolic execution with separation logic. In *Proc. of APLAS’05*, pp. 52–68, 2005.

[4] R. Bornat. Proving pointer programs in hoare logic. In *Proc. of MPC’00*, pp. 102–126, London, UK, 2000.

[5] M. Bozga, R. Iosif, and Y. Laknech. Storeless semantics and alias logic. In *Proc. of PEPM’03*, pp. 55–65, 2003.

[6] Y. Chen, L. Ge, B. Hua, Z. Li, and C. Liu. Design of a certifying compiler supporting proof of program safety. In *Proc. of TASE’07*, pp. 117–126, Shanghai, China, Jun 2007. IEEE Computer Society.

[7] Y. Chen, L. Ge, B. Hua, Z. Li, C. Liu, and Z. Wang. A pointer logic and certifying compiler. *Frontiers of Computer Science in China*, 1(3):297–312, July 2007.

[8] The Coq Proof Assistant. <http://coq.inria.fr/>

[9] A. Deutsch. A storeless model of aliasing and its abstractions using finite representations of right-regular equivalence relations. In *Proc. of ICCL’92*, pp. 2–13. IEEE Computer Society Press, 1992.

[10] P. O’Hearn, J. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *CSL’01*, LNCS, Vol 2142, pp. 1–19. Springer, 2001.

[11] A. Ireland. Towards Automatic Assertion Refinement for Separation Logic. In *Proc. of ASE’06*, pp. 309–312, 2006.

[12] H.B.M. Jonkers. Abstract storage structures. In de Bakker and van. Vliet, editors, *Algorithmic Languages*, pp. 321–343. IFIP, North. Holland, 1981.

[13] F. Mehta and T. Nipkow. Proving pointer programs in higher-order logic. *Information and Computation*, 199(1–2):200–227, 2005.

[14] A. Møeller and M. Schwartzbach. The pointer assertion logic engine. In *Proc. of PLDI’01*, June 2001.

[15] J. G. Morrisett, D. Walker, K. Cray, et al. From system F to typed assembly language. In *Proc. of POPL’98*, New York: ACM Press, 1998, 85–97.

[16] G. C. Necula. Proof-carrying code. In *Proc. of POPL’97*, New York: ACM Press, 1997, 106–119.

[17] J. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proc. of LICS’02*, pp. 55–74, Washington, DC, USA, 2002. IEEE Computer Society.

[18] N. Rinetzky, J. Bauer, T. Reps, M. Sagiv, and R. Wilhelm. A semantics for procedure local heaps and its abstractions. In *Proc. of POPL’05*, pp. 296–309, 2005.

[19] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM TOPLAS*, 24(3):217–298, 2002.

[20] A. Venet. Automatic analysis of pointer aliasing for untyped programs. *Science of Computer Programming*, 35(2):223–248, 1999.

[21] Zhifang Wang and Zhenming Wang. plcc home page. Available at <http://mail.ustc.edu.cn/~zfwang7/research/plcc.htm>, Nov. 2007.

[22] H. W. Xi. Applied type system (extended abstract). In *post-workshop Proceedings of TYPES 2003*, LNCS, Vol 3085. Berlin: Springer-Verlag, pp. 394–408, 2004.

[23] D. C. Yu, N. A. Hamid, Z. Shao. Building certified libraries for pcc: dynamic storage allocation. *Science of Computer Programming*, 2004, 50(1–3): 101–127.