Software Engineering Group
Department of Computer Science
Nanjing University
http://seg.nju.edu.cn

Technical Report No. NJU-SEG-2015-IC-001

2015-IC-001

Assertion-Directed Precondition Synthesis for Loops over Data Structures

Juan Zhai, Hanfei Wang, Jianhua Zhao

Symposium on Dependable Software Engineering Theories, Tools and Applications 2015

Assertion-Directed Precondition Synthesis for Loops over Data Structures

Juan Zhai[1,2(✉)], Hanfei Wang[1,3], and Jianhua Zhao[1,3]

[1] State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China
{zhaijuan,wanghaifei1988}@seg.nju.edu.cn, zhaojh@nju.edu.cn
[2] Software Institute, Nanjing University, Nanjing, China
[3] Department of Computer Science and Technology, Nanjing University, Nanjing, China

Abstract. Program verification typically generates verification conditions for a program to be proven and then uses a theorem prover to prove their correctness. These verification conditions are normally generated by means of weakest-precondition calculus. Nevertheless, the weakest-precondition calculus faces a big challenge when dealing with loops. In this paper, we propose a framework that automatically generates preconditions for loops that iterate over commonly-used data structures. The preconditions are generated based on given assertions of loops and they are proved to be strong enough to ensure those given assertions hold. The data structures dealt with in our framework include one-dimensional arrays, acyclic singly-linked lists, doubly-linked lists and static lists. Such loops usually achieve their final results by focusing on one element in each iteration. In many such cases, the given assertion and the corresponding precondition of the loop separately reflect the part and the whole or vice versa. Inspired by this, our framework automatically generates precondition candidates for loops by transforming a given assertion. Then the framework uses the SMT solver Z3 and the weakest-precondition calculator for non-loop statements provided in the interactive code-verification tool Accumulator to check whether they are strong enough to prove the given assertion. The framework has been integrated into the tool Accumulator to generate suitable preconditions for loops, which greatly relieves the burden of manually providing preconditions for loops.

1 Introduction

Program verification is a classic approach to improve software reliability by verifying program correctness. A standard method for program verification is to generate weakest-preconditions for assertions of a program and then prove these weakest-preconditions using theorem provers. The generation of weakest-preconditions is of great significance in the research field of program verification and has been explored in many literatures, see e.g., [1–5]. Weakest-preconditions for simple program statements can be obtained easily by weakest-precondition calculus techniques while weakest-preconditions for loop statements are difficult

X. Li et al. (Eds.): SETTA 2015, LNCS 9409, pp. 258–274, 2015.
DOI: 10.1007/978-3-319-25942-0_17

to generate. In order to verify programs containing loop statements, programmers are required to provide weakest-preconditions for loops, which increases the burden for programmers. Automatic generation of weakest-preconditions for loops will facilitate the formal verification of programs containing loops.

However, it is challenging to calculate weakest-preconditions for loop statements due to two main factors: (1) there are a great many kinds of loops which makes it difficult to find a uniform way to automatically gain weakest-preconditions for all kinds of loop statements. (2) It is difficult to determine whether the loop terminates and obtain the exact number of loop iterations. In this paper, we present a framework to automatically generate preconditions for the kind of loops that manipulate commonly-used data structures. These data structures include one-dimensional arrays, acyclic singly-linked lists, doubly-linked lists and static lists. The data stored in these data structures can be modified in the loop, but the shapes of these data structures cannot be modified. We generate preconditions for such loops with respect to both intermediate assertions inside the loops and post-conditions of the loops. Although the preconditions generated using our framework may not be the weakest-preconditions, they are proved to be practically useful to prove the given assertions of the kind of loops that operate frequently-used data structures.

The framework proposed in this paper is inspired by our statistic analysis on loops that operate frequently-used data structures occurring in several open-source softwares. These softwares include *memcached*, *Apache httpd* and *nginx*. The inspiring observations are as follows:

1. We found that about eighty percent of loops manipulate a data structure by iterating over its elements. From a practical standpoint, automatic generation of preconditions for this kind of loops would cover a great number of real-world cases and make the task of verifying such loops much easier.
2. This kind of loops usually achieves the final goals by concerning one element in each iteration. In this sense, a precondition of such loops with respect to an intermediate assertion inside the loop is usually a quantified result of the intermediate assertion. On the contrary, a precondition of such loops with respect to a post-condition of the loop usually applies the property in the post-condition to some specific elements. This leads us to believe that we are able to generate practical preconditions for such loops by transforming given assertions of these loops.

Our research is based on Scope Logic [6], though the core idea presented in this paper can be applied in code verifications using other logics. Scope Logic is an extension of Hoare Logic for verification of pointer programs with recursive data structures. An interactive tool named Accumulator(Available at http://seg.nju.edu.cn/scl.html) has been implemented to support code verification in Scope Logic. The weakest-precondition calculation for assignments and conditionals is well-supported in this logic, which greatly eases the verification tasks of programs without loops. However, loop statements cannot benefit from this calculation. This motivates us to provide an automatic framework to generate preconditions for loops to relieve the burden of verify loops.

Given a loop, we first collect information like names/types of the loop control variable, and check whether the loop can be handled by our approach. Then we generate precondition candidates of the loop based on a given assertion, which are subsequently checked to see whether they are strong enough to prove the given assertion. The checking process uses the high-performance SMT solver Z3 [7] and the weakest-precondition calculator for non-loop statements in Accumulator. For an intermediate assertion inside a loop, we first compute the weakest-precondition of the loop body with respect to the intermediate assertion using the weakest-precondition calculator provided in Accumulator. Then precondition candidates are obtained by transforming the generated weakest-precondition according to whether it contains the loop control variable. By contrast, we generate preconditions of the loop for a post-condition by transforming the post-condition itself based on whether it contains loop variables.

The main contribution of this paper is a novel framework that automatically generates preconditions for loops that manipulate commonly-used data structures, including one-dimensional arrays, acyclic singly-linked lists, doubly-linked lists and static lists. The framework has been implemented as a module of the tool Accumulator. We have evaluated it on several programs and the results show that the framework is capable of generating suitable preconditions to prove the partial correctness of the loops manipulating commonly-used data structures.

The remainder of the paper is organized as follows. Section 2 makes a brief introduction to Scope Logic and its weakest-precondition calculus. Section 3 gives a motivating example to show why automatic generation of preconditions for loops are necessary in code verification. Section 4 gives the details of generating preconditions for while-loops by dealing informally with an example program. Section 5 sketches the implementation and application of the framework. Section 6 lists the limitations of our approach together with the future work. Section 7 surveys related work and, finally, Section 8 concludes the paper.

2 Preliminary

In this section, we present a brief overview of Scope Logic and the weakest-precondition calculus in Scope Logic.

2.1 Scope Logic

Scope Logic is an extension of Hoare Logic for verification of pointer programs with recursive data structures. For details, please refer to [6]. The basic idea of Scope Logic is that the value of an expression e depends only on the memory units which are accessed during the evaluation of e. The set of these memory units are called the memory scope of e, denoted as $\mathfrak{M}(e)$. If no memory unit in the set $\mathfrak{M}(e)$ is modified by program statements, the value of e keeps unchanged.

$$isSL(x : P(Node)) : bool \triangleq (x == null)?true : isSL(x \rightarrow link)$$

$$Nodes(x : P(Node)) : SetOf(P(Node)) \triangleq (x == null)?\emptyset : (\{x\} \cup Nodes(x \rightarrow link))$$

$$isSLSeg(x : P(Node), y : P(Node)) : bool \triangleq (x == null)?false : ((x == y)?true : isSLSeg(x \rightarrow link, y))$$

$$NodesSeg(x : P(Node), y : P(Node)) : SetOf(P(Node)) \triangleq (x == null)?\emptyset : ((x == y)?\emptyset : (\{x\} \cup NodesSeg(x \rightarrow link, y)))$$

Fig. 1. A set of recursive functions of singly-linked lists

User-Defined Recursive Functions. Scope Logic allows users to define recursive functions to describe properties of recursive data structures. For example, four user-defined recursive functions are given in Fig. 1 to specify properties of acyclic singly-linked lists. Here an acyclic singly-linked list node contains two fields: the data field d and the link field $link$. These functions will be used in the rest of this paper to verify code manipulating acyclic singly-linked lists.

Example 1. In Fig. 1, the function $isSL(x)$ asserts that if a node x is a null pointer or $x \rightarrow link$ points to an acyclic singly-linked list, then x is an acyclic singly-linked list. The function $Nodes(x)$ yields the node set of the singly-linked list x. The function $isSLSeg(x, y)$ asserts that if the node x can reach node y along the field $link$, then the segment from x to y is an acyclic singly-linked list segment. The function $NodesSeg(x, y)$ yields the set of nodes from node x to node y (excluded) along the filed $link$. □

Nevertheless, some properties should be provided to support local reasoning because first-order logic cannot handle recursive functions directly. Some selected properties of the user-defined functions in Fig. 1 are listed in Table 1. Take the first property as an example, it describes that if the expression x is null, then x is an acyclic singly-linked list and the node set of x is empty.

Table 1. Properties of user-defined acyclic singly-linked list functions

ID	Property
1	$\forall x(x == null) \Rightarrow (isSL(x) \wedge Nodes(x) == \emptyset)$
2	$\forall x(x \neq null \wedge isSL(x)) \Rightarrow (Nodes(x) == \{x\} \cup Nodes(x \rightarrow link) \wedge x \notin Nodes(x \rightarrow link))$
3	$\forall x \forall y(x == y) \Rightarrow (NodesSeg(x, y) == \emptyset)$
4	$\forall x \forall y(x \neq null \wedge y == null) \Rightarrow (NodesSeg(x, y) == Nodes(x))$
5	$\forall x \forall y(isSL(y) \wedge isSLSeg(x, y)) \Rightarrow (isSL(x) \wedge Nodes(x) == NodesSeg(x, y) \cup Nodes(y))$

Program-Point-Specific Expressions. In Scope Logic, assertions and verifications are written in the proof-in-code form. Formulas are written at program points which are places before and after program statements. All the program points are uniquely numbered. A formula at a program point is used to describe a property that the program satisfies. If a program runs into the program point j before it runs into the program point i, $e@j$ can be written at the program point

```
{1: sl ≠ null , isSL(sl) , ∀x ∈ Nodes(sl).x → d > 0}
min = sl→d;
{2: sl ≠ null , isSL(sl) , min == sl → d , min > 0 , ∀x ∈ Nodes(sl).x → d > 0 ,
    min ∈ (λx.x → d)[Nodes(sl)]}
p = sl→link;
{3: sl ≠ null , isSL(sl) , p == sl → link , isSL(p) , ∀x ∈ Nodes(sl).x → d > 0 , min > 0 ,
    p ∈ Nodes(sl)}
while(p ≠ null){
    {4: p ≠ null , p ∈ Nodes(sl) , ∀x ∈ Nodes(sl).x → d > 0}
    if(p→d < min)
        {5: p ≠ null , p → d < min}
        min = p→d;
        {6: p ≠ null , min == p → d , min > 0 , min ∈ (λx.x → d)[Nodes(sl)]}
    else
        {7: p ≠ null , not(p → d < min)}
        skip;
        {8: p ≠ null }
    {9: p ≠ null }
    p = p→link;
    {10: isSL(sl) }
}
{11: p == null , ∀x ∈ Nodes(sl).x → d > 0 ,
    (λx.x → d)[Nodes(sl)] == (λx.x → d)[Nodes(sl)]@1 ,
min ∈ (λx.x → d)[Nodes(sl)] , ∀x ∈ Nodes(sl).x → d ≥ min}
```

Fig. 2. Find the minimum value of an acyclic singly-linked list

i to represent the value of e evaluated at the program point j. Expressions like $e@j$ are called program-point-specific expressions. With this kind of expressions, the relations between different program states can be specified.

Set-Image Expression. Set-image expression is of the form $\lambda x.exp[setExp]$, which means the set of values derived by applying the anonymous function defined by $\lambda x.exp$ to the elements in the set $setExp$.

Restricted Quantifier Expression. Restricted quantifier expression is of the form $\forall x \in setExp.exp$, which asserts that for each element x in the set $setExp$, exp is satisfied.

Example 2. A program written in the proof-in-code form is given in Fig. 2 where the numbered program points and some formulas are also shown. This program finds the minimum element in the acyclic singly-linked list sl. The entrance program point and the exit point are respectively 1 and 11. The preconditions of this program are shown at the program point 1 while the post-conditions are written at the program point 11.

2.2 Weakest-Precondition Calculus in Scope Logic

The weakest-precondition was introduced by Dijkstra in [8]. For a statement S and a predicate Q on the post-state of S, the weakest-precondition of S with

respect to Q, written as $wp(S, Q)$, is a predicate that characterizes all the pre-states of S from which no execution will go wrong and from which every terminating execution ends in a state satisfying Q. In program verification, weakest-preconditions are often used to prove the correctness of programs in regard to assertions represented by preconditions and post-conditions. Here we assume that P stands for the predicates on the pre-state of S, we can verify $\{P\}S\{Q\}$ by proving $P \Rightarrow wp(S, Q)$.

The calculation of weakest-preconditions for assignments and conditionals are well-supported in Scope Logic. Suppose that we have an assignment $e_1 = e_2$ and the program points before/after this assignment are i and j respectively. The differences between the program state at i and the program state at j result from the modification of the contents in the memory unit $(\&e_1)@j$. The basic idea of generating weakest-preconditions is that for an arbitrary x of a memory unit, the value of $((x \neq (\&e_1)@i)? * x : e_2@i)$ at the program point i is equivalent to the value of $*x$ at the program point j. Firstly, an expression $exp(\&e)$ is constructed. The value of $exp(\&e)$ at i equals to the value $\&e$ at j. After that, $exp(e')$ is constructed as $(exp(\&e) \neq (\&e_1)@i)? * exp(\&e) : e_2@i$. As discussed earlier, the value of $exp(e)$ at i and the value of $*(\&e)$ at j are equivalent. The detailed rules for generating weakest-preconditions are omitted here. Interested readers can refer to the paper [6].

3 Motivating Example

The program *findMin* in Fig. 2 finds the minimum element in the list sl. It is used as an example here to show why preconditions of loops are necessary and helpful to verify this program.

Proving that no null-pointer dereference occurs requires that $p \neq null$ holds in some program points. Take the assertion $p \neq null$ at program point 5 as an example, we can compute the weakest-precondition at program point 4 with respect to it and we get $(p \rightarrow d < min) \Rightarrow (p \neq null)$. At program point 4, this weakest-precondition can be implied by $p \neq null$ which is surly true because of the loop condition.

The above is a simple assertion which can be proved inside the loop. There are many other assertions that cannot be proved without preconditions of the loop, for example, $min > 0$ at program point 6. Just like $p \neq null$, the weakest-precondition

$$(p \rightarrow d < min) \Rightarrow (p \rightarrow d > 0) \tag{1}$$

at program point 4 is first generated for $min > 0$ at program point 6. Proving this weakest-precondition requires ingenuity in generating the precondition of the loop for it.

Based on the weakest-precondition (1), the following precondition of the loop is generated using our framework.

$$\forall x \in Nodes(sl).((x \rightarrow d < min) \Rightarrow (x \rightarrow d > 0)) \tag{2}$$

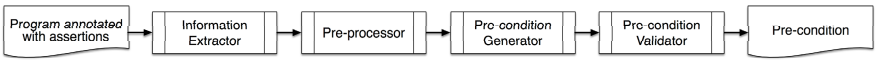


Fig. 3. Overview of our approach

The precondition (2) is proved to be a loop invariant of this loop, thus it holds at the program point 4. The weakest-precondition (1) at program point 4 is implied by (2) together with $p \neq null$ and $p \in Nodes(sl)$. In this way, the assertion $min > 0$ at program point 6 is proved to be true.

Preconditions of loops are also necessary to prove post-conditions of loops. Take the post-condition

$$\forall x \in Nodes(sl).x \rightarrow d \geq min \tag{3}$$

as an example. Our framework generates the precondition of the loop for it and the precondition is

$$\forall x \in NodesSeg(sl, p).x \rightarrow d \geq min \tag{4}$$

This precondition is proved to be a loop invariant and the post-condition (3) can be implied by this precondition (4) together with the loop exit condition $p == null$, the precondition of the loop $l \neq null$ and Property 4 in Table 1.

From this, we can see that automatically generating preconditions for loops is useful and practical in verifying programs.

4 Design

In this section, we present our approach of automatically generating preconditions for the kind of loops that manipulate commonly-used data structures. Fig. 3 gives the overview of our approach, which takes the program annotated with assertions as input, and uses information extractor, pre-processor, pre-condition generator and pre-condition validator to generate pre-conditions for the loop in this program. The information extractor takes the program as input, and extracts necessary information to generate pre-conditions and checks whether the loop can be handled by our approach. The pre-processor derives some simple but useful loop invariants used as the premises to check the generated preconditions. The pre-condition generator generates pre-conditions from a given assertion and the information extracted before. The pre-condition validator makes use of the SMT solver Z3 and the weakest-precondition calculator to check whether the generated pre-condition is strong enough to prove the correctness of the assertion based on which to generate the pre-conditions.

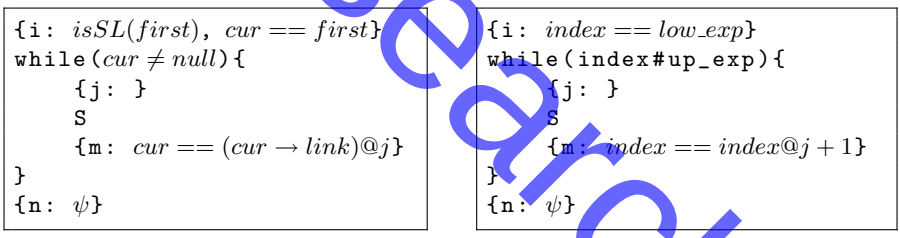
In the rest of this section, we discuss the details of our approach. The discussion is driven by the example shown in Fig. 2.

4.1 Information Extractor

The information extractor mainly performs the following two tasks:

Information Collector. Our approach collects different kinds of information of the loop, which include names, types, initial values and final values of the loop control variables, the loop condition, and the data structure manipulated by the loop. If the data structure manipulated by the loop is a one-dimensional array, the size of the array and the traverse pattern are also needed to be gathered. Here the traverse pattern means whether the loop iterates over the array elements from left to right or the other way.

Loop Checker. Our approach is capable of generating pre-conditions for while-loops that iterate over elements stored in a data structure without modifying the shape of this data structure. These loops should conform to our pre-defined loop patterns. Because of space limitation, we only gives the patterns for loops manipulating acyclic singly-linked lists and one-dimensional arrays.



(a) (b)

Fig. 4. The loop patterns for acyclic singly-linked lists and one-dimensional arrays

(1) Pattern of Loops Manipulating Acyclic Singly-Linked Lists. The pattern of the while-loops that manipulate an acyclic singly-linked list in C syntax is given in Fig. 4a. Here $first$ represents the acyclic singly-linked list traversed in the loop and cur represents the expression used to access each list node. In this pattern, cur is also the loop control variable. This is often the case when an acyclic singly-linked list is iterated over in practice. For example, in Fig. 2, p is the loop control variable and the loop body accesses each data stored in the list node referred to by p.

For the loop to be handled by our approach, the following are also required:

1) $isSL(first)$ and $cur == first$ hold at the program point i.
2) The loop control condition is either $cur \neq null$ or $null \neq cur$.
3) $cur == (cur \rightarrow link)@j$ holds at the program point m, which guarantees that the value of cur after the loop body equals to the value of $cur \rightarrow link$ before the loop body.

4) For each assignment $e_1 = e_2$ in the loop body, $\&e_1 \notin \mathfrak{M}(isSL(first))$ holds at the point before the assignment statement, which guarantees that the loop does not modify the field *link* of all the nodes. Hence the shape of the singly-linked list is not modified.

Together with the definition of an acyclic singly-linked list in Fig. 1 and the loop control condition, the condition 3) listed above can guarantee that the loop will terminate.

(2)Pattern of Loops Manipulating One-Dimensional Arrays. The pattern of the while-loops that iterate over a one-dimensional array from left to right in C syntax is shown in Fig. 4b. The pattern of the while-loops that iterate over a one-dimensional array from right to left is symmetrical, which is omitted here. In this pattern, *index* represents the subscript used to access each element of the array being traversed. The expressions *low_exp* and *up_exp* respectively represent the lower/upper bound expressions of *index*. In this pattern, *index* is also the loop control variable. The expression *index*#*up_exp* represents the loop control condition where # can be operators $<$, $\leq$ or $\neq$.

For the loop to be handled by our approach, the following are also required:

1) $up\_exp == up\_exp@j$ holds at the program point m.
2) The loop control condition is one of the following six forms: $index < up\_exp$, $index \neq up\_exp$, $up\_exp > index$, $up\_exp \neq index$, $index \leq up\_exp$ and $up\_exp \geq index$.
3) $index == index@j + 1$ holds at the program point m.

Together the loop control condition, the condition 3) listed above can guarantee that the loop will terminate.

4.2 Pre-processor

The pre-processor makes an attempt to generate some simple but useful loop invariants which are used to check the generated preconditions. These loop invariants are verified via Z3 and the weakest-precondition calculator. For example, in Fig. 2, the loop invariant $(p \neq null) \Rightarrow (p \in Nodes(sl))$ is generated.

4.3 Pre-condition Generator

In this section, we describe in details how to generate preconditions candidates based on a given intermediate assertion inside a loop and a post-condition of a loop.

Generating Precondition Candidates from an Intermediate Assertion. Generating preconditions based on an intermediate assertion inside a loop is given in Algorithm 1. The algorithm takes as arguments a loop program *loop*, a given intermediate assertion *assertion* of *loop* and the information $info$ of *loop*. This algorithm returns a set of precondition candidates with respect to *assertion*.

Algorithm 1. Generating Preconditions from an Intermediate Assertion

Input: a loop program *loop*; an intermediate assertion *assertion* of *loop*; loop information $info$;
Output: a set of precondition candidates with respect to *assertion*;

```
1: set ← ∅;
2: let i be the program point before the loop body of loop
3: let j be the program point of assertion inside loop
4: wp ← compute_wp(assertion, j, i);
5: swp ← simplify(wp);
6: if swp contains the loop control variable then
7:     qwp = quantify(swp, info);
8:     set ← set ∪ qwp;
9: else
10:    set ← set ∪ swp;
11: end if
12: let cond be the loop control condition of loop
13: for each exp ∈ set do
14:    exp' = transform(exp, cond);
15:    set ← set ∪ exp';
16: end for
17: return set;
```

To start with, the variable *set* used to store precondition candidates of *loop* is initialized to an empty set. In line 4, the algorithm invokes the procedure *compute_wp* to obtain the weakest-precondition at program point i with respect to the intermediate assertion *assertion* at program point j. Since *wp* may contain redundant information, this algorithm simplifies it via the procedure $simplify$ in line 5 and the simplified result is stored in *swp*. Then the algorithm analyzed *swp* to see whether *swp* contains the loop control variable. If so, what *swp* reflects is the property which holds by a set of elements. In this case, *swp* is universally quantified via the procedure $quantify$ in line 7. The quantified result *qwp* is added to *set* as a precondition candidate. Otherwise, we simply regard *swp* as a precondition candidate and add it to *set* in line 10. Lines 13-16 make a transformation of each expression in *set* and add the transformation result to *set*. The procedure $tranform$ in line 14 construct a new expression $(cond) \Rightarrow (exp)$ where *cond* represents the loop control condition of *loop*. Eventually, *set* which contains all the precondition candidates is returned in line 17.

Example 3. Consider the intermediate assertion $isSL(l)$ at program point 10 of the program in Fig. 2. Firstly, Algorithm 1 computes the weakest-precondition at program point 4 for $isSL(l)$ and we get $p \rightarrow d < min?(isSL(l)) : (isSL(l))$ which can be simplified to $isSL(l)$. Since $\&p \notin \mathfrak{M}(isSL(l))$ holds at program point 4, the algorithm simply adds $isSL(l)$ to the set containing precondition candidates. After that, the pre-condition $(p \neq null) \Rightarrow (isSL(l))$ is constructed via the procedure $transform$ in line 14 and added to the candidate set. □

Universal Quantification. It is obvious that when a formula at the program point before the loop body contains the loop control variable, this formula actually describes some property that should be held by the set of elements which are manipulated in the loop. For this reason, our framework universally quantifies the weakest-precondition at the program point before the loop body with respect to an intermediate assertion inside the loop body to generate a precondition candidate of the loop if this weakest-precondition contains the loop control variable. We achieve this by introducing a fresh variable which does not appear elsewhere in the program or in the weakest-precondition. The details of quantifying for loops manipulating singly-linked lists and one-dimensional arrays are given below. Doubly-linked lists and static lists are also dealt with in this paper, but the details are omitted here because of space limitation.

(1) Acyclic Singly-Linked List. Suppose that $first$ represents an acyclic singly-linked list and cur represents the loop control variable used to access each list node of $first$. The concrete expressions of $first$ and cur are obtained through a static analysis. Besides, we assume that the expression exp represents the weakest-precondition at the program point before the loop body. Note that exp contains the loop control variable cur. By universally quantifying over cur which appears in exp, we get the following precondition candidate of the loop:

$$\forall x \in Nodes(first).(exp[cur \longmapsto x])$$

Example 4. In Fig. 2, the weakest-precondition at program point 4 with respect to the intermediate assertion

$$min \in (\lambda x.x \to d)[Nodes(sl)] \tag{5}$$

at program point 6 is as follows:

$$(p \to d < min) \Rightarrow (p \to d \in (\lambda x.x \to d)[Nodes(sl)]) \tag{6}$$

where p is the loop control variable used to access each list node. Our framework universally quantifies the weakest-precondition (6) by introducing a new variable y to substitute p. The quantified result is the following precondition candidate:

$$\forall y \in Nodes(sl).((y \to d < min) \Rightarrow (y \to d \in (\lambda x.x \to d)[Nodes(sl)])) \tag{7}$$

The precondition candidate (7) is proved to be strong enough to guarantee (6) holds at program point 4. Thus the assertion (5) is verified to be true at program point 6. □

(2) One-Dimensional Arrays. Suppose that arr represents a one-dimensional array, $index$ represents the loop control variable used as the subscript to access each element of arr, low and $high$ respectively represent the lower/upper bound expressions of $index$. The concrete expressions of arr, $index$, low and $high$ are obtained through a static analysis. Besides, we assume that the expression exp represents the weakest-precondition at the program point before the loop body.

```
{1: a ≠ null, b ≠ null, ∀x ∈ [0,99].a[i] ≥ 0}
size = 100;
{2: a ≠ null, b ≠ null, ∀x ∈ [0,99].a[i] ≥ 0, size == 100}
i = 0;
{3: a ≠ null, b ≠ null, ∀x ∈ [0,99].a[i] ≥ 0, size == 100, i == 0}
while(i < size){
    {4: i ≥ 0, i < size, a ≠ null, a[i] ≥ 0}
    b[i] = a[i];
    {5: b ≠ null, b[i] == a[i], b[i] ≥ 0}
    i = i + 1;
    {6: b[i − 1] ≥ 0}
}
{7: ∀x ∈ [0,99].b[x] == a[x], (λx.a[x])[0,99] == (λx.b[x])[0,99]}
```

Fig. 5. Array Copy

Here *exp* contains the loop control variable *index*. By universally quantifying over *index*, we get the following precondition candidate of the loop:

$$\forall x \in [low, high].(exp[index \longmapsto x])$$

Example 5. We now illustrate the quantifying process for programs operating one-dimensional arrays using the program *arrayCopy* in Fig. 5. This program copies each element of the array a to the corresponding position of the array b. The subscript expression i is the loop control variable. The lower bound expression of i is 0 while the upper bound of i is 99.

The weakest-precondition at program point 4 with respect to the intermediate assertion $b[i-1] \geq 0$ at program point 6 is $a[i] \geq 0$. Our framework universally quantifies $a[i] \geq 0$ by introducing a new variable x to substitute i and get the following quantified result:

$$\forall x \in [0, 99].a[i] \geq 0 \tag{8}$$

The precondition candidate (8) is proved to be strong enough to guarantee $a[i] \geq 0$ holds at program point 4. Thus $b[i-1] \geq 0$ is verified to be true at program point 6. □

Generating Precondition Candidates from a Post-condition. Algorithm 2 illustrates the process of generating preconditions based on a post-condition of a loop. The algorithm takes as arguments a loop program *loop*, a given post-condition *post* of *loop* and the information $info$ of *loop*. This algorithm returns a set of precondition candidates with respect to *post*.

This algorithm divides post-conditions of loops into two distinct categories. The classification criteria is whether the post-condition contains loop variables.

If the post-condition contains loop variables, the algorithm invokes the procedure *gen_equiv_exps* in line 3 to get a set of expressions which are equivalent to *post* at the loop exit point. These expressions are regarded as precondition candidates and they are added to *set* in line 4. The core idea of *gen_equiv_exps*

Algorithm 2. Generating Preconditions from a Post-condition

Input: a loop program *loop*; a post-condition *post* of *loop*; loop information $info$;
Output: a set of precondition candidates with respect to *post*;
1: $set \leftarrow \emptyset$;
2: **if** post contains loop variables **then**
3: $exps = gen\_equiv\_exps(post, info)$;
4: $set \leftarrow set \cup exps$;
5: **else**
6: $set \leftarrow set \cup post$;
7: **end if**
8: **return** *set*;

is to substitute some sub-expressions of a given expression with some new sub-expressions which equals to the original one at the loop exit point. The substitution follows the heuristics given in our previous paper [9]. Further details of *gen_equiv_exps* can also be found in [9]. As stated in that paper, when the argument of the procedure *gen_equiv_exps* is a post-condition of a loop, the generated expressions are very likely to be loop invariants which can be used to imply the post-condition. Consequently, these expressions can be regarded as preconditions of the loop to guarantee that the post-condition holds at the loop exit point.

Example 6. The post-condition

$$min \in (\lambda x.x \rightarrow d)[Nodes(sl)] \tag{9}$$

at program point 11 of the loop in Fig. 2 contains a loop variable *min*. Algorithm 2 generates some precondition candidates for it by invoking the procedure *gen_equiv_exps*. One of the candidates is

$$(l \neq p) \Rightarrow (min \in (\lambda x.x \rightarrow d)[NodesSeg(sl, p)]) \tag{10}$$

and it is proved to be a loop invariant. The post-condition (9) can be implied by (10) together with $l \neq null$, $p == null$ and Property 4 in Table 1. Thus (10) is a precondition that is strong enough to prove the post-condition (9). □

If the post-condition does not contain any loop variable, there is a great possibility that the execution of the loop has no influence on the post-condition. Considering this, Algorithm 2 thinks of the post-condition itself as a precondition candidate and adds it to *set* in line 6.

Example 7. The post-condition

$$\forall x \in Nodes(sl).x \rightarrow d > 0 \tag{11}$$

at program point 11 of the loop in Fig. 2 does not contain any loop variable. Algorithm 2 regards (11) as a precondition. The candidate (11) is proved to be a loop invariant and holds at program point 11. Apparently the post-condition (11) can be proved with the presence of itself as the precondition of the loop. □

4.4 Checking Precondition Candidates

After the precondition candidates are generated, we check their validity utilizing the SMT solver Z3 and the weakest-precondition calculator for non-loop statements provided in the tool Accumulator.

Checking Precondition Candidates for an Intermediate Assertion. Suppose that *assertion* is an intermediate assertion of a loop, *wp* is the weakest-precondition of *assertion* at the program point before the loop body and *pre* is the generated precondition of the loop. To guarantee *assertion* holds, *wp* at the program point before the loop body must be true each time the program runs into this point. If *pre* is a loop invariant and it is strong enough to imply *wp*, then *assertion* can be proved. For *pre* to be our desired precondition of the loop, it must satisfy the following conditions:

1. The known preconditions imply *pre* at the loop entry point;
2. *pre* holds at the points before/after the loop body;
3. *pre* holds at the loop exit point;
4. *pre* and the proven properties at the program point before the loop body imply *wp* with respect to the intermediate assertion *assertion*.

If *pre* satisfies the first three conditions, it is a loop invariant. If the fourth condition is also met, it means that *pre* is strong enough to ensure the intermediate assertion *assertion* holds.

In our framework, the first condition and the last condition are checked using the SMT solver Z3.

The correctness of *pre* at the point after the loop body is checked using the following steps. Firstly, we use Z3 to check whether *pre* holds at the point after the loop body. If not, we compute the weakest-precondition of the loop body with respect to *pre* and check whether this weakest-precondition can be implied by *pre* together with the proven properties at the point before the loop body and the loop control condition. If so, it means *pre* holds at the point after the loop body.

As long as *pre* holds at the loop entry point and the point after the loop body, it surely holds at the point before the loop body and the loop exit point. As a result, the condition 2 and the condition 3 are satisfied.

If *pre* passes all these validation steps, it is a suitable precondition of the loop with respect to the given intermediate assertion.

Checking Precondition Candidates for a Post-condition. Suppose that *post* is a post-condition of a loop and *pre* is the generated precondition of the loop based on *post*. If *pre* is true both at the loop entry point and the loop exit point, *post* is sure to hold. To ensure *pre* holds at the loop exit point, it must be true at the program point after the loop body in addition to the loop entry point. In this case, *pre* is actually a loop invariant according to the definition of loop invariant. Thus if the precondition *pre* of the loop is a loop invariant

and it is strong enough to imply *post*, it is the desired precondition. To be more specific, if the following conditions are satisfied, the generated precondition *pre* is strong enough to ensure *post* holds at the loop exit point.

1. The preconditions imply *pre* at the loop entry point;
2. *pre* holds at the points before/after the loop body;
3. *pre* holds at the loop exit point;
4. *pre* and the proven properties at the loop exit point imply the given post-condition of the loop.

Our framework checks these conditions similarly in the way it deals with the precondition generated from an intermediate assertion.

5 Implementation and Application

We have implemented the proposed framework as part of the interactive code-verification tool Accumulator. The framework has been evaluated using various programs. The results show that by automatically generating suitable preconditions of loops, our framework can be fully leveraged to help prove some assertions of loops that manipulate commonly-used data structures. In this way, the tasks of proving the partial correctness of programs can be greatly eased. For details of these examples, please visit http://seg.nju.edu.cn/toolweb/casestudy.html.

6 Limitations

Our framework currently can deal with while-loops that manipulate commonly-used data structures including one-dimensional arrays, two-dimensional arrays, acyclic singly-linked lists and static lists. We plan to deal with more types of loops to cover a greater variety of real-world programs, such as foreach loops and loops that contain break and continue statements. Programs with nested loops would be another interesting extension since we deeply believe that the same techniques can be applied. In addition, it is possible that similar techniques can be developed for loops that manipulate data structures like binary search trees, heaps and multi-dimensional arrays.

Furthermore, the loops dealt with in our framework iterate over each element without modifying their shapes, which limits the scope of programs that can be handled. We will attempt to handle loops that modify the shape of singly-linked lists, such as inserting or removing a node from the original list.

7 Related Work

As always, automatic inference of preconditions for loops is a critical challenge. In recent years, there is a plenty of research on the automatic generation of preconditions for loops.

The majority of the existing works compute preconditions for loops by transforming loops into acyclic forms. In this way, they are able to use the techniques for successive sequential statements to compute preconditions for loops. Some works like [4,5] achieve transforming loops by bounding the number of loop iterations. Other papers, such as [2,3], work by de-sugaring loops with loop invariants. Some works attempt to automatically derive the necessary loop invariants while others expect programmers to provide loop invariants. Although our work uses the concept of loop invariant when we generate preconditions for loops, no transformations of loops are needed in our work.

Another approach is proposed in [10] and this approach computes preconditions for loops based on invariant relations [11]. Intuitively, an invariant relation is a pair of states (s, s') in which s' can be derived from s by application of an arbitrary number of iterations of the loop body. This work focuses on numeric computations while our work can identify more types of preconditions, such as the shape of a recursive data structure and quantifying information.

The works [9,12,13] share the similarities with our work in that user-defined predicates and lemmas are used to allow programmers to describe a wide range of data structures. The work [12] aims at generating post-conditions while our work focuses on generating pre-conditions. The goals of the works [9,13] are to synthesize loop invariants which is different from our pre-condiiton generation goal.

8 Conclusion

An automatic framework of generating preconditions for loops is presented in this paper, which deals with loops manipulating commonly-used data structures by iterating over the elements. We first generate precondition candidates for a loop by transforming a given assertion of the loop or the weakest-precondition of the loop body with respect to a given assertion inside the loop. Then we check the validity and the effectiveness of these precondition candidates via the SMT solver Z3 and the weakest-precondition calculator for non-loop statements in Accumulator. Whether the precondition generated is strong enough to imply the given assertion is checked since it is the ultimate goal of generating preconditions.

The key novelty of our framework is that we focus on loops that manipulate heavily-used data structures. This kind of loops appears frequently in real-world programs according to our statistic analysis. Thus our framework is of great use to boost automation and efficiency in the code verification of many practical programs. Though in actual programs, some loops iterate over elements of a container data structure via an iterator, this kind of loops is essentially the same with the loops studied in this paper. When we can handle these interfaces well, we will be able to generate preconditions for these kinds of complex loops using the idea presented in this paper.

The framework has been implemented as part of the verification tool Accumulator. Its effectiveness and practicability have been validated by several programs. By generating useful preconditions for loops manipulating commonly-used data structures, our framework significantly reduces the burden of providing appropriate preconditions for loops manually.

References

1. Berghammer, R.: Soundness of a purely syntactical formalization of weakest preconditions. Electronic Notes in Theoretical Computer Science 35 (2000)
2. Flanagan, C., Saxe, J.B.: Avoiding exponential explosion: Generating compact verification conditions. In: ACM SIGPLAN Notices. Volume 36, ACM (2001)
3. Barnett, M., Leino, K.R.M.: Weakest-precondition of unstructured programs. In: ACM SIGSOFT Software Engineering Notes. Volume 31, ACM (2005)
4. Leino, K.R.M.: Efficient weakest preconditions. Information Processing Letters **93**(6), 281–288 (2005)
5. Jager, I., Brumley, D.: Efficient directionless weakest preconditions. Technical report, CMU-CyLab-10-002, CMU, CyLab (2010)
6. Jianhua, Z., Xuandong, L.: Scope logic: an extension to hoare logic for pointers and recursive data structures. In: Liu, Z., Woodcock, J., Zhu, H. (eds.) ICTAC 2013. LNCS, vol. 8049, pp. 409–426. Springer, Heidelberg (2013)
7. de Moura, L., Bjørner, N.S.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
8. Dijkstra, E.W.: Guarded commands, nondeterminacy and formal derivation of programs. Commun. ACM **18**(8), 453–457 (1975)
9. Zhai, J., Wang, H., Zhao, J.: Post-condition-directed invariant inference for loops over data structures. In: 2014 IEEE Eighth International Conference on Software Security and Reliability-Companion (SERE-C), IEEE (2014)
10. Mraihi, O., Ghardallou, W., Louhichi, A., Labed Jilani, L., Bsaies, K., Mili, A.: Computing preconditions and postconditions of while loops. In: Cerone, A., Pihlajasaari, P. (eds.) ICTAC 2011. LNCS, vol. 6916, pp. 173–193. Springer, Heidelberg (2011)
11. Mraihi, O., Louhichi, A., Jilani, L.L., Desharnais, J., Mili, A.: Invariant assertions, invariant relations, and invariant functions. Volume 78, 1212–1239 Elsevier (2013)
12. Chin, W.N., David, C., Nguyen, H.H., Qin, S.: Automated verification of shape, size and bag properties via user-defined predicates in separation logic. Sci. Comput. Program. **77**(9), 1006–1036 (2012)
13. Qin, S., He, G., Luo, C., Chin, W.N., Chen, X.: Loop invariant synthesis in a combined abstract domain. Journal of Symbolic Computation **50**, 386–408 (2013)