Technical Report No. NJU-SEG-2016-IC-003

2016-IC-003

Precondition Calculation for Loops Iterating over Data Structures

Juan Zhai, Bin Li, Zhenhao Tang, Jianhua Zhao, Xuandong Li

International Conference on Software Quality, Reliability and Security 2016

2016 IEEE International Conference on Software Quality, Reliability and Security

Precondition Calculation for Loops Iterating over Data Structures

Juan Zhai*†, Bin Li*, Zhenhao Tang*, Jianhua Zhao*, Xuandong Li*
*State Key Laboratory for Novel Software Technology
Department of Computer Science and Technology
Nanjing University, Nanjing, Jiangsu, P.R.China 210093
†Software Institute, Nanjing University, Nanjing, Jiangsu, P.R.China 210093
Email: {zhaijuan, hsslb, tangzh}@seg.nju.edu.cn, {zhaojh, lxd}@nju.edu.cn

***Abstract*—Precondition calculation is a fundamental program verification technique. Many previous works tried to solve this problem, but ended with limited capability due to loop statements. We conducted a survey on loops manipulating commonly-used data structures occurring in several real-world open-source programs, and found that about 80% of such loops iterate over elements of a data structure, indicating that automatic calculation of preconditions with respect to post-conditions of these loops would cover a great number of real-world programs and greatly ease code verification tasks.**

In this paper, we specify the execution effect of a program statement using the memories modified by the statement and the new values stored in these memories after executing the statement. Thus, conditional statements and loop statements can be uniformly reduced to a sequence of assignments. Also we present an approach to calculate preconditions with respect to given post-conditions of various program statements including loops that iterate over elements of commonly-used data structures (e.g., acyclic singly-linked lists) based on execution effects of these statements. With execution effects, post-conditions and loop invariants can also be generated. Our approach handles various types of data including numeric, boolean, arrays and user-defined structures. We have implemented the approach and integrated it into the code verification tool, Accumulator. We also evaluated the approach with a variety of programs, and the results show that our approach is able to calculate preconditions for different kinds of post-conditions, including linear ones and universally quantified ones. Preconditions generated with our approach can ease the verification task by reducing the burden of providing loop invariants and preconditions of loop statements manually, which improves the automatic level and efficiency, and makes the verification less error-prone.

Keywords*—*precondition; loop; data structure; execution effect; equivalent expression

I. Introduction

Program verification is an effective approach to ensure rigorous, unambiguous guarantees on program correctness. One basic approach for program verification is to reduce a program specification $\{p\}\ s\ \{q\}$ into a logical formula $p \Rightarrow \mathsf{WP}(q, s)$ where $\mathsf{WP}(q, s)$ represents the weakest precondition [9] of s with respect to q. Many previous works attempt to calculate weakest preconditions, but the existence of loop statements limits their abilities. The work [16] is able to calculate weakest preconditions for programs that manipulate pointers, but loops are not supported. Works like [10], [19] calculate weakest preconditions for loops by de-sugaring loops with loop invariants. However, automatic generation of loop invariants itself remains largely unfulfilled. Other works like [14] calculate weakest preconditions for loops by bounding the number of iterations to transform loop programs into acyclic programs, which sometimes may lose accuracy. Weakest precondition calculation for loop statements is of great significance in program verification, but it is still a tough challenge, which remains unresolved to this day.

As one of the basic component in modern programming languages, container data structures are frequently used in real-world, widely-used applications. We have conducted a statistic analysis on loops manipulating commonly-used container data structures occurring in several open-source softwares, such as *Apache httpd* and *nginx*. We found that about 80% of such loops iterate over elements of a container data structure. From a practical point of view, automatic calculation of preconditions with respect to given post-conditions of this kind of loops would cover a great number of cases in real-world widely-used programs, and greatly ease the code verification tasks. This motivates us to develop an automatic precondition calculator for loops that iterate over elements of commonly-used container data structures.

In this paper, we specify the execution effect of a statement as memories modified by the statement and new values stored in these memories after executing the statement, and present a novel approach to calculate preconditions with respect to given post-conditions of program statements including loops based on the execution effect. Naturally, our approach supports nested loops. Furthermore, other assertions including post-conditions and loop invariants can be derived with the presence of the execution effect of a statement. Our approach handles a wide range of data types (e.g., numeric, boolean, arrays and structures like acyclic singly-linked list node) and it is able to deal with universally quantified formulas which are necessary and important to express properties of programs that manipulate unbounded data structures like acyclic singly-linked lists. Our contributions are highlighted as follows.

1) To the best of our knowledge, we are the first to specify the execution effect of a loop statement as memories modified by the loop and new values stored in these memories after executing the loop, which reduces a loop statement into a sequence of assignments and effectively summarizes a loop statement.
2) We propose a novel approach to synthesize execution effects of loop statements that manipulate commonly-used container data structures including one-dimensional

arrays, two-dimensional arrays and acyclic singly-linked lists, which reduces a loop statement into a sequence of assignment statements.

3) With the execution effect of a loop statement, we propose an approach to calculate preconditions, post-conditions and loop invariants for this loop statement.
4) Based on the proposed approaches, we implemented a prototype, and integrated it into the code verification tool, Accumulator [1]. We also evaluated our approaches, and the results show that we can effectively generate various types of assertions, including preconditions, post-conditions and loop invariants.

The remainder of the paper is organized as follows. Section II introduces Scope Logic [16]. Section III gives an example which motivates our work. Section IV specifies the execution effect of a program statement. Section V presents the method to calculate equivalent expressions based on execution effects. Section VI demonstrates how to synthesize execution effects of assignments, sequential statements and conditional statements. Section VII summarizes the types of loops that our approach can handle, and how they are treated in our approach. Section VIII gives some applications of execution effects. Section IX sketches the implementation of the approach and presents a real world case. Section X surveys related works, and in the end, Section XI concludes the paper.

II. Preliminary

In this section, we present a brief overview of Scope Logic [16], which is an extension of Hoare Logic to deal with pointers and recursive data structures.

The basic idea of Scope Logic is that the value of an expression e depends only on the contents stored in a finite set of memories. The value of e keeps unchanged if no memory in this set is modified by program statements. This set of memories is denoted as $\mathfrak{M}(e)$.

In Scope Logic, specifications and verifications are written in the proof-in-code form. Formulas are written at **program points**, which are places before and after program statements. For a sequential statement $s_1; s_2$, the program point after s_1 is just the program point before s_2. A formula at a program point means that this formula holds each time the program runs into this program point. Formulas at the program point before a statement and the program point after a statement are separately preconditions and post-conditions of this statement.

Scope Logic introduces **program-point-specific expressions** to specify the relations between program states at different program points. A program point j is said to dominate a program point i if the program must go through the program point j before it goes to the program point i. In this case, we write $e@j$ at the program point i to denote the value of e evaluated at the program point j when the program was at the program point j the last time. At the program point j, the program-point-specific expression $e@j$ equals to e, and at a program point other than j including the program point i, $e@j$ is treated as a constant.

To describe program properties, especially properties of recursive data structures, Scope Logic allows users to define recursive functions. For example, the recursive functions defined in Fig. 1 specify properties of acyclic singly-linked lists. The function $isSL(x)$ asserts that if a node x is a null pointer or $x \rightarrow n$ points to an acyclic singly-linked list, then x is an acyclic singly-linked list, and the function $isSLSeg(x, y)$ states that the node x can reach the node y along the field n and the nodes from the node x to the node y make up an acyclic singly-linked list segment. The function $nodes(x)$ yields the set of all the nodes of the singly-linked list x, while the function $nodesSeg(x, y)$ yields the set of nodes from the node x to the node y (excluded) along the filed n.

First-order logic cannot deal with user-defined functions. To support local reasoning about programs whose properties are expressed using user-defined functions, we need to provide properties of these functions. Both the given properties of the use-defined functions and the definitions of these functions are used to reason about programs. Some properties for the user-defined functions in Fig. 1 are given in Fig. 2. Take the first property as an example, it describes that if the pointer variable x is $null$, then x represents an acyclic singly-linked list and the node set of x is empty.

III. Background and Motivation

A. Weakest Preconditions for Loops

Weakest preconditions have been widely used to prove the correctness of a program with respect to specifications represented by pairs of preconditions and post-conditions. Automatic generation of weakest preconditions can greatly facilitate the formal verification of programs to improve software quality and reliability. However, precondition calculation faces formidable challenges when loop statements are considered. Traditionally, preconditions for loops are provided manually, which is a tedious and error-prone task, and also increases the burden for programmers. This poses a tremendous obstacle to the practical use of code verification.

There is an enormous variety of loops which makes it impossible to find a uniform way to automatically calculate preconditions for all of them. From our statistic analysis on loops manipulating frequently-used data structures occurring in several open-source softwares including *memcached*, *Apache httpd* and *nginx*, we found that about eighty percent of such loops achieve their goals by iterating over elements of a data structure. From a practical standpoint, automatic calculation of preconditions which are powerful enough to prove post-conditions for this kind of loops would cover a great number of real-world widely-used programs, making the code verification much easier and less error-prone.

The above factors motivate us to develop an approach to automatically calculate preconditions with respect to given post-conditions of loop statements that manipulates commonly-used data structures to ease the code verification tasks and improve the quality and reliability of softwares.

B. Our Idea: Calculating Preconditions from Modified Memories and New Values

The value of an expression e depends only on the values stored in a finite set of memories, and the value of e keeps unchanged if no memory in this set is modified by program statements. Given a program statement s, and i, j be respectively the program points before and after s, the program state

$$
\begin{aligned}
isSL(x:P(Node)):bool &\triangleq (x==null)?true:isSL(x\rightarrow link)\\
Nodes(x:P(Node)):SetOf(P(Node)) &\triangleq (x==null)?\emptyset:(\{x\}\cup Nodes(x\rightarrow link))\\
isSLSeg(x:P(Node),y:P(Node)):bool &\triangleq (x==null)?false:((x==y)?true:isSLSeg(x\rightarrow link,y))\\
NodesSeg(x:P(Node),y:P(Node)):SetOf(P(Node)) &\triangleq (x==null)?\emptyset:((x==y)?\emptyset:(\{x\}\cup NodesSeg(x\rightarrow link,y)))
\end{aligned}
$$

Fig. 1: Recursive functions of acyclic singly-linked lists

$$
\begin{aligned}
\forall x(x==null) &\Rightarrow (isSL(x)\wedge nodes(x)==\emptyset)\\
\forall x(x\neq null\wedge isSL(x)) &\Rightarrow (nodes(x)==\{x\}\cup nodes(x\rightarrow n)\wedge x\notin nodes(x\rightarrow n))\\
\forall x\forall y(x==y) &\Rightarrow (nodesSeg(x,y)==\emptyset)\\
\forall x\forall y(x\neq null\wedge y==null) &\Rightarrow (nodesSeg(x,y)==nodes(x))\\
\forall x\forall y(isSL(y)\wedge isSLSeg(x,y)) &\Rightarrow (isSL(x)\wedge nodes(x)==nodes(y)\cup nodesSeg(x,y))\\
\forall x\forall y\forall z(isSLSeg(x,y)\wedge isSLSeg(y,z)) &\Rightarrow (isSLSeg(x,z)\wedge nodesSeg(x,z)==nodesSeg(x,y)\cup nodesSeg(y,z))
\end{aligned}
$$

Fig. 2: Properties of acyclic singly-linked lists

at the program point i is different with the program state at the program point j only on the memories modified by the statement s. Suppose that m is the memory modified by s and v is the new value stored in m after executing s. Both m and v are evaluated at the program point i. For any memory address x in e, the value of $(x==m)?v:*x$ at the program point i equals to the value of $*x$ at the program point j, and thus we can calculate an equivalent expression at i which is equal to e at j. When e is a post-condition, the equivalent expression is actually the precondition.

Basic Idea: Given a loop statement and its post-conditions, we firstly attempt to analyze the loop body to get the memories modified by the loop body and the new values in these memories after executing the loop body, based on which we then synthesize the memories modified by the loop and the new values stored in these memories after executing the loop. With the synthesized modified memories and the new values, the loop is equivalent to a sequence of assignments which assigns the new values to the modified memories.

Example. Fig. 3 gives a program that operates on two one-dimensional arrays. The loop in the program manipulates the arrays a and b via the loop control variable i which iterates over the closed integer interval $[0,s-1]$. The program computes the sum of the elements whose indexes range from 0 to $s-1$ in the array b, and sets the k-th element of the array a with the sum of the elements whose subscripts range from 0 to k in the array b. Also, each element in the array b is set to the value 0. The numbered program points of the program, together with some formulas, are also shown. The entrance program point and the exit program point are respectively 1 and 10. The formulas like $sum==(\sum_{x=0}^{s-1}b[x])@4$ at the program point 10 are the post-conditions of the loop, which are also the post-conditions of the program.

```
{1: $a \neq b$}
s = 100;
{2: $s == 100$}
i = 0;
{3: $i == 0$}
sum = 0;
{4: $sum == 0,\ \ i == 0$}
while(i < s)
{
      {5: $sum == (\sum_{x=0}^{i-1} b[x])@4$}
      sum = sum + b[i];
      {6: $sum == sum@5 + b[i]$}
      a[i] = sum;
      {7: $a[i] == sum$}
      b[i] = 0;
      {8: $b[i] == 0$}
      i = i + 1;
      {9: $i == i@5 + 1$ }
}
{10: $sum == (\sum_{x=0}^{s-1} b[x])@4$,
        $\forall x \in [0, s-1].a[x] == (\sum_{y=0}^{x} b[y])@4$,
        $\forall x \in [0, s-1].b[x] == 0$,
        $\forall x \in [1, s-1].a[x] > a[x-1]$}
```

Fig. 3: The program *summation*

Through an analysis of the loop body, we know that the loop body respectively assigns the new values $sum+b[i]$, $sum+b[i]$, 0 and $i+1$ evaluated at the program point 5 to the memories $\&sum$, $\&a[i]$, $\&b[i]$ and $\&i$. As the loop control variable i iterates over the closed integer interval $[0,s-1]$, the value of sum at the program point 5 is evaluated to be the value of $\sum_{x=0}^{i-1}b[x]@4$. And hence in each iteration, the value stored in the memory address $\&a[i]$ is evaluated as the value of $\sum_{x=0}^{i} b[x]@4$.

Based on the result, we can synthesize the memories modified by the loop and the corresponding new values, and the result is as follows: the values of the variables i and sum are respectively set to $s-1$ and $(\sum_{x=0}^{s-1} b[x])@4$, and for each x in the interval $[0, s-1]$, $a[x]$ is set to the value of $(\sum_{y=0}^{x} b[y])@4$, and $b[x]$ is set to 0.

With the memories modified by the loop and the new values stored in the memories, we are able to calculate preconditions with respect to post-conditions of this loop. Take the post-condition

$$\forall x \in [1, s-1].a[x] > a[x-1] \qquad (1)$$

of the loop as an example. From the modified memories and the new values of the loop, we know that the value of s is not modified by the loop, and for each x in the interval $[0, s-1]$, the value of $a[x]$ at the point 10 equals to the value of $\sum_{y=0}^{x} b[y]$ which is evaluated at the program point 4. By substituting the expressions $a[x]$ and $a[x-1]$ in the post-condition (1) with their values $\sum_{y=0}^{x} b[y]$ and $\sum_{y=0}^{x-1} b[y]$, we get the formula $\forall x \in [1, s-1]. \sum_{y=0}^{x} b[y] > \sum_{y=0}^{x-1} b[y]$ which can be simplified to

$$\forall x \in [1, s-1].b[x] > 0 \qquad (2)$$

The value of (2) evaluated at the program point 4 equals to the value of (1) evaluated at the program point 10, and the formula (2) is just the precondition of the loop with respect to the post-condition (1).

IV. Execution Effect

In this section, we specify the execution effect of a program statement as the memories modified by this statement and the new values stored in these memories after executing the statement, which is the basis of calculating preconditions for a statement.

Executing a program statement is to manipulate memories related to this statement. In our work, we group the memories into two categories which are separately defined in Definition 1 and Definition 2.

Definition 1: A **single memory expression (SiM)** is any expression with the type $P(t)$, i.e., a pointer type to a type t. Here t can be **int**, **boolean**, or some other pointer types.

Definition 2: A **set memory expression (SeM)** is an expression of the form $\lambda x.e[set]$, where e is a memory, set is an expression with a set type, and x is bound in set.

The **kernel (K)** of a memory m, denoted as $\kappa(m)$, and the **range variables (RV)** of a memory m, denoted as $\gamma(m)$, are defined as follows.

- If m is a SiM, then $\kappa(m) = \{m\}$ and $\gamma(m) = \emptyset$;
- If m is a SeM and of the form $\lambda x.m'[set]$, then $\kappa(m) = \kappa(m')$ and $\gamma(m) = \{x\} \cup \gamma(m')$.

Intuitively, the effect of executing a program statement is assigning new values to a finite set of memories while keeping the values of other memories unchanged. We specify the execution effect of each program statement as a set of tuples of memories modified by this statement and the corresponding new values stored in these memories after executing the statement. More formally,

Definition 3: The **execution effect** of a program statement s, denoted as $\tau(s)$, is a finite set of tuples of the form $\langle m, v\rangle$ where m is a memory expression, and v is the new value expression of m. Both m and v are evaluated at the program point before executing s. The type of $\kappa(m)$ and v are respectively $P(t)$ and t for some t, and the variables in $\gamma(m)$ can occur in v.

A special expression ϱ is introduced in our work to specify the value of v when the new value in the memory m cannot be determined. In other words, if we know that a statement s modifies the memory m, but we cannot get the new value in m, then the execution effect of s is specified as $\{\langle m, \varrho\rangle\}$.

- If m is a SiM, then the tuple means the value stored in the memory m is v;
- If m is a SeM and of the following form

$$\lambda x_1.(\ldots(\lambda x_k.e[set_k])\ldots)[set_1]$$

then the tuple means that for each memory expressed as $m[v_1/x_1][v_2/x_2]\ldots[v_k/x_k]$, the corresponding new value is represented as $v[v_1/x_1][v_2/x_2]\ldots[v_k/x_k]$, where v_i is a value in the set set_i for $i = 1, 2, \ldots, k$.

Take the loop in Fig. 3 as an example. Its execution effect is as follows:

$$\left\{\begin{array}{c} \langle \&i, s\rangle, \\ \langle \&sum, \sum_{x=0}^{s-1} b[x]\rangle, \\ \langle \lambda x.(\&b[x])[0, s-1], 0\rangle, \\ \langle \lambda x.(\&a[x])[0, s-1], \sum_{y=0}^{x} b[y]\rangle \end{array}\right\}$$

The first two tuples specify that the new values of the variables i and sum are respectively the values of s and $\sum_{x=0}^{s-1} b[x]$ evaluated at the loop entrance point, namely at the program point 4. The third tuple specifies that the elements indexed from 0 to $s-1$ in the array b are set to 0 while the last one specifies that the elements indexed from 0 to $s-1$ of the array a are set to the value of $\sum_{y=0}^{x} b[y]$ evaluated at the program point before executing the loop.

The set of memories modified by a statement s, denoted as $\chi(s)$, is the set of memory expressions in the execution execution $\tau(s)$. The memory expressions in $\chi(s)$ may overlap with each other. It is required that if two memory expressions specify two overlapped memory sets, their corresponding new-value expressions must agree on the common memory units. For example, the execution effect of the sequential statement " *p=1; *q=2; " is specified as

$$\left\{\begin{array}{c} \langle p, (p == q?2:1)\rangle, \\ \langle q, 2\rangle, \end{array}\right\}$$

The memory units specified by p and q may be the same. When the condition $p == q$ holds, their new value expressions are both evaluated to 2.

What should be highlighted here is that for any program statement including the conditional statement and the loop statement, its execution effect actually reduces it into a sequence of special assignment statements. Thus it is feasible to calculate predictions for a loop statement in the same way as with an assignment statement.

V. Equivalent Expression Calculation

In this section, we introduce the concept of equivalent expression, and present the method to calculate and simplify equivalent expressions.

Definition 4: If s is a program statement and e is an expression at the program point after s, then the equivalent expression e' of s with respect to e is an expression at the the program point before s, which is equivalent to the expression e after executing the statement s. When e is a post-condition of the statement s, e' is the precondition of s with respect to the post-condition e.

For example, if $a[i]$ is an expression at the program point after "i=j; ", then $a[j]$ is the equivalent expression at the program point before "i=j; " with respect to $a[i]$ since the value of $a[i]$ at the program point after the assignment equals to the value of $a[j]$ at the program point before the assignment.

Note that the concept of equivalent expressions can be viewed as a generalization of pre-conditions. A pre-condition is in regard to a predicate while an equivalent expression is in regard to a predicate or just a simple expression like a variable.

A. Equivalent Expression Calculation

Given a program statement s and an expression e, the equivalent expression of s with respect to e, denoted as $\epsilon(e, s)$, is calculated recursively. The calculation of $\epsilon(e, s)$ is reduced based on the rules given in Table I. Column "e" lists the formats of the expressions while Column "$\epsilon(e, s)$" shows how to calculate the equivalent expression of the statement s with respect to the expression e.

From the table, we can see that the equivalent expression calculation $\epsilon(e, s)$ is reduced to the calculations of the equivalent expressions of the sub-expressions of e. Consequently, as long as we can calculate the equivalent expression for the expression $*e'$, the calculation of equivalent expressions for other expressions can be solved simply.

The equivalent expression of the program statement s with respect to the expression $*e'$, namely $\epsilon(*e', s)$, depends on whether the memory address specified by e' is modified by the statement s. If so, the new value in the memory e' is crucial for the value of $\epsilon(*e', s)$. As mentioned in Section IV, $\tau(s)$ represents the tuples of the memories modified by the program statement s and the corresponding new values after executing s. Thus $\epsilon(*e', s)$ can be derived by iteratively calculating the equivalent expression with respect to each tuple in $\tau(s)$. Suppose that p represents the program point before the statement s, the equivalent expression of $*(e'@p)$ with respect to each tuple $\langle m, v\rangle$ is a conditional expression $c?e'' : *(e'@p)$, where c tests whether the memory specified by $e'@p$ overlaps with the memory specified by m, and e'' is constructed based on the value v.

- If m is a SiM, then the equivalent expression is $(m == e')@p?v@p : *(e'@p)$
- If m is a SeM, $\gamma(m) = \{x_1, x_2, \ldots, x_n\}$ and the corresponding sets of the variables $x_1, x_2, \ldots, x_n$ are respectively $set_1, set_2, \ldots, set_n$.
  - If we can find n expressions $e_1, e_2, \ldots, e_n$ such that e' is the same as the expression derived by substituting $x_1, x_2, \ldots, x_n$ in $\kappa(m)$ with $e_1, e_2, \ldots, e_n$, then $e' \in m$ holds if and only if $e_i \in set_i$ holds for $i = 1, 2, \ldots, n$. In this case, the equivalent expression is as follows:
  $$((e_1 \in set_1) \wedge (e_2 \in set_2) \wedge \cdots \wedge (e_n \in set_n))@p? \\ (v[e_1/x_1][e_2/x_2] \ldots [e_n/x_n])@p : *(e'@p)$$
  - Otherwise, the equivalent expression is $(e' \in m)@p?\varrho : *(e'@p)$

B. Equivalent Expression Simplification

The calculated equivalent expressions may contain conditional sub-expressions, program-point-specific sub-expressions ($@p$), and the special expression ϱ. These sub-expressions should be simplified to make the equivalent expressions easy to handle. Because the equivalent expression is evaluated at the program point p, all the $@p$ sub-expressions can be removed after the equivalent expressions are finally constructed. Moreover, a conditional expression $c?e_1 : e_2$ can be simplified to e_1 if the condition c holds and it can be simplified to e_2 if $\neg c$ holds. The following premises are used to check whether c or $\neg c$ holds.

- The given preconditions of the program statement s;
- The memory layout properties of the programming language shown in Table II;
- The conditions enclosing the sub-expression under simplification.
  - For a conditional expression $c?e_1 : e_2$, the condition c is used as a premiss when e_1 is being simplified and $\neg con$ is used as a premiss when e_2 is being simplified;
  - For expressions of the forms $\lambda x.e[set]$ and $\forall x \in set.e$, $x \in set$ is used as a premiss when the sub-expression e is being simplified.

We design and implement a simple decision procedure which employs the state-of the-art SMT Solver Z3 [8] to simplify the calculated equivalent expressions. If the final expression still contains ϱ which means unknown as a sub-expression, we just use ϱ as the final result.

Considering the loop statement, denoted as L, in Fig. 3, now we show how the equivalent expression $\epsilon(\forall x \in [0, s-1].a[x] > 0, L)$ is calculated.

In the expression $\forall x \in [0, s-1].a[x] > 0$, the memory-access sub-expressions are s and $a[x]$ which are transformed to $*(\&s)$ and $*(\&a[x])$ respectively.

In regard to the first tuple $\langle \&sum, \sum_{x=0}^{s-1} b[x]\rangle$ in $\tau(L)$, the equivalent expression is

$$\forall x \in [0, ((\&sum == \&s)@4?(\textstyle\sum_{x=0}^{s-1} b[x])@4 : s) - 1]. \\ ((\&sum == \&a[x])@4?(\textstyle\sum_{x=0}^{s-1} b[x])@4 : a[x]) > 0$$

Because the two conditions $\neg(\&sum == \&s)@4$ and $\neg(\&sum == \&a[x])@4$ hold, the equivalent expression is simplified to itself.

Similarly, the equivalent expression is still itself after the tuples $\langle \&i, s\rangle$ and $\langle \lambda x.(\&b[x])[0, s-1], 0\rangle$ are treated.

With regard to the last tuple $\langle \lambda x.(\&a[x])[0, s-1], \sum_{y=0}^{x} b[y]\rangle$, the constructed equivalent expression is

$$\forall x \in [0, ((\&s \in \lambda x.(\&a[x])[0, s-1])@4?\varrho : s) - 1]. \\ ((x \in [0, s-1])@4?(\textstyle\sum_{y=0}^{x} b[y])@4 : a[x]) > 0$$

TABLE I: Rules to calculate an equivalent expression

e	$\epsilon(e,s)$	e	$\epsilon(e,s)$
`a constant/` `a quantified variable`	e	$e_0?e_1 : e_2$	$\epsilon(e_0,s)?\epsilon(e_1,s) : \epsilon(e_2,s)$
$e'@k(k \neq j)$	$e'@k$	$op\ e'(op \neq *)$	$op\ \epsilon(e',s)$
$\&v$	$\&v$	$e_1\ op\ e_2$	$\epsilon(e_1,s)\ op\ \epsilon(e_2,s)$
$*e'$	$\epsilon(*(\epsilon(e',s)),s)$	$\&(e'.n)$	$\&(\epsilon(\&e',s) \to n)$
v	$\epsilon(*(\&v),s)$	$\&(e' \to n)$	$\&(\epsilon(e',s) \to n)$
$e'.n$	$\epsilon(*(\&e'.n),s)$	$\&(e_1[e_2])$	$\&(*\epsilon(\&e_1,s)[\epsilon(e_2,s)])$
$e' \to n$	$\epsilon(*(\&e' \to n),s)$	$\lambda x.e_1[e_2]$	$\lambda x.\epsilon(e_1,s)[\epsilon(e_2,s)]$
$e_1[e_2]$	$\epsilon(*(\&e_1[e_2]),s)$	$\forall x \in e_1.e_2$	$\forall x \in \epsilon(e_1,s).\epsilon(e_2,s)$

Since the equivalent expression is evaluated at the program point 4, by simplifying it and removing '@4', we have the final equivalent expression $\forall x \in [0, s-1]. \sum_{y=0}^{x} b[y] > 0$.

VI. Synthesis of Execution Effects for Assignments, Sequential Statements and Conditional Statements

As has been pointed out in the previous section, the execution effect is the key to the calculation of equivalent expressions as well as preconditions. This section will explain in detail how to synthesize execution effects of assignments, sequential statements and conditional statements.

A. *Synthesizing Execution Effects for Assignments*

The execution effect of the assignment $e_1 = e_2$ is $\{\langle \&e_1, e_2 \rangle\}$ where $\&e_1$ and e_2 are evaluated at the program point before this assignment. For example, the execution effect of the statement "sum = sum+b[i]" is a one-element set $\{\langle \&sum, sum + b[i] \rangle\}$.

B. *Synthesizing Execution Effects for Sequential Statements*

Let s be a sequential statement $s_1; s_2$. The memories modified by s are the union of those modified by s_1 and s_2. Since $\chi(s_2)$ is evaluated at the program point before s_2, we must first calculate the equivalent representation of $\chi(s_2)$ at the program point before s_1 and thus $\chi(s)$ is $\chi(s_1) \cup \{\epsilon(e, s_1) \| e \in \chi(s_2)\}$. For each memory m in $\chi(s)$, its new value in $\tau(s)$ is calculated as $\epsilon(\epsilon(*(\kappa(m)@p), s_2), s_1)$ where p is the program point before s. For example, $\tau($" $t = t + 2; a[t] = 0;$ "$)$ is $\{\langle \&t, t+2 \rangle, \langle \&a[t+2], 0 \rangle\}$.

TABLE II: Axioms for memory layout and memory access

name	axiom
DEREF-REF	$*\&e = e$
REF-DEREF	$e \neq null \Rightarrow \& * e == e$
PVAR-1	$\&v \neq null$
PVAR-2	$\&v_1 \neq \&v_2$
PVAR-3	$\&v \neq \&r \to n$
PVAR-4	$\&v \neq \&a[i]$
REC-1	$r \neq null \Rightarrow \&r \to n \neq null$
REC-2	$(r_1 \to n = r_2 \to n) \Leftrightarrow (r_1 = r_2)$
REC-3	$r_1 \to n_1 \neq r_2 \to n_2$
ARR-1	$a \neq null \wedge (0 \leq i < c) \to \&((*a)[i] \neq null$
ARR-2	$(\&((*a_1)[i_1]) = \&((*a_2)[i_2])) \Leftrightarrow$ $(a_1 == a_2 \wedge i_1 == i_2 \wedge 0 \leq i_1, i_2 < c)$
ARR-REC	$r_1 \to n_1 \neq r_2 \to n_2$

TABLE III: Calculation rules

condition	new value
$m \in \chi(s_1) \wedge m \in \chi(s_2)$	$c?v_1 : v_2$
$m \in \chi(s_1) \wedge m \notin \chi(s_2)$	$c?v_1 : \epsilon(*(\kappa(m)@p_1), s_2)$
$m \notin \chi(s_1) \wedge m \in \chi(s_2)$	$c?\epsilon(*(\kappa(m)@p_2), s_1) : v_2$

[1] v_1 and v_2 are separately the new values of the memory m in $\tau(s_1)$ and $\tau(s_2)$

[2] p_1 and p_2 are separately the program points before the else-branch and the then-branch

C. *Synthesizing Execution Effects for Conditional Statements*

The execution effect of a conditional statement is synthesized from the execution effects of its branches. Let s be a conditional statement if c then s_1 else s_2. The modified memories of s, i.e., $\chi(s)$, are the union of the modified memories of the branches, i.e., $\chi(s_1) \cup \chi(s_2)$. For each memory expression m in $\chi(s)$, the corresponding new value expression is computed shown in Table III.

Considering the conditional statement in Fig. 4, the execution effects of the then-branch and the else-branch are respectively

$$\left\{ \begin{array}{l} \langle \&absSum, absSum + a[i] \rangle, \\ \langle \&posSum, posSum + a[i] \rangle \end{array} \right\}$$

and

$$\left\{ \begin{array}{l} \langle \&absSum, absSum - a[i] \rangle, \\ \langle \&negSum, negSum + a[i] \rangle \end{array} \right\}$$

Based on the rules in Table III, we can get the execution effect of the conditional statement below

$$\left\{ \begin{array}{l} \langle \&absSum, a[i] > 0?absSum + a[i] : absSum - a[i] \rangle, \\ \langle \&posSum, a[i] > 0?posSum + a[i] : posSum \rangle, \\ \langle \&negSum, a[i] > 0?negSum : negSum + a[i] \rangle \end{array} \right\}$$

VII. Synthesis of Execution Effects for Loops

In this section, we specify the loop statements that are dealt with in our approach and describe how to synthesize execution effects for such loops.

```
  if(a[i]>0){
    absSum = absSum + a[i];
    posSum = posSum + a[i];
  }
  else{
    absSum = absSum - a[i];
    negSum = negSum + a[i];
  }
```

Fig. 4: The program *summation*

A. Processible Loops

Our approach is able to synthesize the execution effect of a while-statement that is controlled by a variable which iterates over an integer interval, or a set of acyclic singly-linked list nodes, and the loop iterations update the memories in some regular patterns. Nested Loops are supported in our approach. More specifically, given a while-statement while (c) s, denoted as L, and the program point p before this loop statement, we require that the while-statement belongs to one of the following two categories:

- The loop control condition c is of the forms $w \sim e$ or $e \sim w$, where $\sim$ is an operator in $\{<, >, \geq, \leq, \neq\}$, w is an integer-typed variable and e is an expression with the type int. In this case, we also require:
  - The condition $\epsilon(e, s) == e$ holds, which means that the value of the expression e is not modified by the loop body s;
  - If c is one of the forms $w < e$, $w \leq e$, $e > w$, or $e \geq w$, the condition $\epsilon(w, s) == w + 1$ should be met;
  - If c is one of the forms $w > e$, $w \geq e$, $e < w$, or $e \leq w$, the condition $\epsilon(w, s) == w - 1$ should be met.
  - If c is one of the forms $w \neq e$ or $e \neq w$, the condition $\epsilon(w, s) == w - 1 \vee \epsilon(w, s) == w + 1$ should be met.
- The loop control condition c is of the forms $w \neq null$ or $null \neq w$, where w is a pointer to an acyclic singly-linked list node. In this case, we also require:
  - $isSL(w)$ holds at the program point p;
  - The condition $\epsilon(w, s) = w \rightarrow link$ is met, which means that the loop iterates over the nodes of the acyclic singly-linked list one by one;
  - For each assignment $e_1 = e_2$ in the loop body, $\&e_1 \notin \mathfrak{M}(isSL(sl))$ holds at the program point before the assignment where sl is the initial value of w before entering the loop and $\mathfrak{M}(isSL(sl))$ contains the memories of the field n of each list node. This formula guarantees that the loop does not modify the field n of all the nodes being iterated, and thus the shape of the singly-linked list is not modified.

In both categories, w is the loop control variable. The loop statements which satisfy the conditions in each category are sure to terminate.

In the rest of this paper, $\xi(w)$ is used to represent the valid value set of the control variable w, which means that with any value in $\xi(w)$, we can access an element of the data structure being iterated over. $\xi(w)$ can be obtained by analyzing the loop control condition and the known assertions at the program point p. Considering the loop in Fig. 3, i is the loop control variable and $\xi(i)$ is $[0, s@4 - 1]$.

Through an analysis of such loops, we can see that the memories modified by the loop body, i.e., $\chi(s)$ can be grouped into the following two categories:

- A memory m is a **fixed address (FA)** if and only if the condition $\epsilon(m, s) == m$ is satisfied.
- A memory m is a **shifting address (SA)** if and only if the condition $\epsilon(m, s) == m[\epsilon(w, s)|w]$ is satisfied.

From this, we can see that if a memory expression m is a FA, each loop iteration updates the same memories, and if m is a SA, each loop iteration updates different memories as the loop control variable iterates over its valid value set. For example, the memory $\&sum$ in Fig. 3 is a FA while the memory $\&a[i]$ is a SA.

B. Synthesizing Execution Effects for Loops

In this subsection, we discuss how to synthesize the execution effect of a loop statement by analyzing and transforming the execution effect of its loop body.

We will first define some predicates which will be used in the later sections.

Let w be the loop control variable and w_0 be a value in the valid value set $\xi(w)$. The set of values of w before w equals to w_0, denoted as $\mu(w, w_0)$, is shown in Fig. 5. With $\mu(w, w_0)$, we define the predicate $\mathcal{NB}(e, w_0)$ to specify that an expression e is not modified by the iterations before the iteration when the value of w equals to w_0:

$$\mathcal{NB}(e, w_0) := (w \in \mu(w, w_0)) \Rightarrow (\epsilon(e, s) == e)$$

Similarly, $\mathcal{NA}(e, w_0)$ is defined to specify that an expression e is not modified by the iterations after the iteration when the value of w equals to w_0:

$$\mathcal{NA}(e, w_0) := (w \in \nu(w, w_0)) \Rightarrow (\epsilon(e, s) == e)$$

where $\nu(w, w_0)$ shown in Fig. 6 is the set of values of w after w equals to w_0.

Suppose that $\langle m, v\rangle$ is a tuple in the execution effect of the loop body, namely, $\tau(s)$, the method of synthesizing the execution effect of the loop $\tau(L)$ is discussed below.

1) m is a FA: When m is a FA, each loop iteration updates the same memory m, and thus m is a memory modified by the loop and it belongs to $\chi(L)$. The new value of m in $\tau(L)$ is computed differently in the following cases:

- If $\epsilon(v, s) == v$ holds, then v is not modified by the loop body, which means each loop iteration assigns the same value v to the memory m. In this case, the tuple corresponding to m in $\tau(L)$ is $\langle m, c?v : *m\rangle$
- If m is a SiM and v is one of the forms $*m$ op e or e op $*m$, where op is an operator in $\{+, -, *, \div, \vee, \wedge\}$, and $\mathcal{NB}(e[w_0/w], w_0)$ holds for an arbitrary value w_0 in $\xi(w)$, then the value stored in the memory m in $\tau(L)$ is specified in Table IV.
  Considering the tuple $\langle \&sum, sum + b[i]\rangle$ of the execution effect of the loop body in Fig. 3, the memory $\&sum$ is a FA and its value $sum + b[i]$ is of the form $*m$ op e. Based on the the first rule in Table IV, we can get the

$$\mu(v, v_0) = \begin{cases} \{x | x \in \xi(v) \wedge x < v_0\} & : \text{when } v \text{ is an integer-typed variable and it increases by 1 in each iteration} \\ \{x | x \in \xi(v) \wedge x > v_0\} & : \text{when } v \text{ is an integer-typed variable and it decreases by 1 in each iteration} \\ \mathsf{nodesSeg}(v@p, v_0) & : \text{when } v \text{ points to an acyclic singly-linked list} \end{cases}$$

Fig. 5: The value set $\mu(v, v_0)$

$$\nu(v, v_0) = \begin{cases} \{x | x \in \xi(v) \wedge x > v_0\} & : \text{when } v \text{ is an integer-typed variable and it increases by 1 in each iteration} \\ \{x | x \in \xi(v) \wedge x < v_0\} & : \text{when } v \text{ is an integer-typed variable and it decreases by 1 in each iteration} \\ \mathsf{nodesSeg}(v_0, v@p) & : \text{when } v \text{ points to an acyclic singly-linked list} \end{cases}$$

Fig. 6: The value set $\nu(v, v_0)$

tuple $\langle \&sum, sum + \sum_{x=0}^{s-1} b[x] \rangle$ for the execution effect of the loop.

- In other cases, the new value of m cannot be computed, and the corresponding tuple is $\langle m, \varrho \rangle$.

TABLE IV: Construction rules for value expressions

op	new value
$+, -$	$*m\ op\ \sum_{x \in \xi(v)} exp[x/v]$
$*, \div$	$*m\ op\ \prod_{x \in \xi(v)} exp[x/v]$
$\vee$	$*m\ op\ \bigvee_{x \in \xi(v)} exp[x/v]$
$\wedge$	$*m\ op\ \bigwedge_{x \in \xi(v)} exp[x/v]$

2) m is a SA: When m is a SA, each loop iteration updates a different set of memories according to the value of the loop control variable. Given a tuple $\langle m, v \rangle$ in $\tau(s)$, the memory modified by the iteration w_0 and the corresponding new value after the iteration are respectively specified by the expressions $m[w_0/w]$ and $v[w_0/w]$ evaluated when the iteration starts.

If the condition $\forall w_0 \in \xi(w).\mathcal{NB}(\kappa(m)[w_0/w], w_0)$ holds, then the expression $\kappa(m)[w_0/w]$ is evaluated to the same value at the program point before the loop and the program point before the iteration w_0, and thus $\lambda x.(m[x/w])[\xi(w)]$ is a memory modified by the loop.

Furthermore, if both $\forall w_0 \in \xi(w).\mathcal{NB}(v[w_0/w], w_0)$ and $\forall w_0 \in \xi(w).\mathcal{NA}(*(\kappa(m)[w_0/w], w_0)$ hold, then $v[w_0/w]$ is evaluated to the same value at the program point before the loop as the value $*(\kappa(m)[w_0/w])$ evaluated at the program point after the loop. In this case, $v[w_0/w]$ is the new value stored in $m[w_0/w]$ after executing the loop, and $\langle \lambda x.(m[x/w])[\xi(w)], v[x/w] \rangle$ is a tuple in the execution effect of the loop. Otherwise, $\langle \lambda x.(m[x/w])[\xi(w)], \varrho \rangle$ is a tuple in the execution effect of the loop.

Take the loop in Fig. 3 as an example, for brevity, the loop and the loop body are respectively denoted as L and S. The variable i is the loop control variable of the loop and $\xi(i)$ is $[0, 99]$. The tuple $\langle \&b[i], 0 \rangle$ is in the execution effect of the loop body, namely, $\tau(S)$, and the three conditions $\forall w_0 \in [0, 99].\mathcal{NB}(\&b[w_0], w_0)$, $\forall w_0 \in [0, 99].\mathcal{NB}(0, w_0)$ and $\forall w_0 \in [0, 99].\mathcal{NA}(*(\&b[w_0], w_0)$ are met, thus we can obtain the tuple $\langle \lambda x.(\&b[x])[0, 99], 0 \rangle$ in $\tau(L)$.

C. Incremental Synthesis

Sometimes we cannot get a tuple for a loop from a tuple of the loop body directly since some conditions may not hold. In some cases, such conditions can be satisfied by substituting some sub-expressions of a value expression in a tuple of the loop body with their equivalent sub-expressions. Thus we propose the incremental synthesis of substituting sub-expressions of a value expression of a tuple of the loop body with new sub-expressions.

The sub-expression, denoted as ma_exp, to be substituted is a memory-access expression and the memory being accessed, denoted as m_exp, is a memory modified by the loop. The sub-expression, denoted as v_exp', used to substitute ma_exp is the value of m_exp evaluated at the beginning of each loop iteration. The sub-expression v_exp' is obtained by transforming the value expression v_exp of the memory m_exp in the corresponding tuple of the loop generated in previous synthesis processes. In general, the transformation is to substitute the range of a variable by substituting the whole range expression in v_exp with the range that covers elements which have been iterated over. The substitution heuristics are the same with those in our previous work [27]. The details are omitted here because of space limitation. We do this incremental synthesis until the result execution effect of the loop reaches a fixed point.

Considering the loop in Fig. 3, $\langle \&a[i], sum + b[i] \rangle$ and $\langle \&sum, sum + b[i] \rangle$ are two tuples in the execution effect of the loop body. After the first synthesis of these two tuples, we get $\langle \lambda x.(\&a[x])[0, s-1], \varrho \rangle$ and $\langle \&sum, \sum_{y=0}^{s-1} b[y] \rangle$ for the loop. Here, we cannot obtain the value expression for the memory $\langle \lambda x.(\&a[x])[0, s-1]$ because some conditions are not met, which triggers the second synthesis for the tuple $\langle \&a[i], sum + b[i] \rangle$. In the second synthesis, we first substitute the range $[0, s-1]$ of the variable y in the value $sum + \sum_{y=0}^{s-1} b[y]$ with the range $[0, i-1]$, and get the value expression $sum + \sum_{y=0}^{i-1} b[y]$. Then we use the expression $sum + \sum_{y=0}^{i-1} b[y]$ to substitute sum in the value expression in the tuple $\langle \&a[i], sum + b[i] \rangle$ of the loop body, and get the new tuple $\langle \&a[i], sum + \sum_{y=0}^{i} b[y] \rangle$. After that, we analyze the transformed tuple and generate the result tuple $\langle \lambda x.(\&a[x])[0, s-1], \sum_{y=0}^{x} b[y] \rangle$ of the execution effect of the loop.

D. Nested Loops

Precondition calculation is more difficult in the presence of nested loops. Our approach, however, is applicable to programs with nested loops. The execution effect of the outer loop is synthesized with inner loops replaced with their execution

```
    i = 100;
L1: while(i<n){
        j = 0;
        {p2: }
    L2: while(j<m){
            c[i][j] = 0;
            k = 0;
            {p3: }
        L3: while(k<p){
                c[i][j] = c[i][j] + a[i][k]*b[k][j];
                k = k + 1;
            }
            j = j + 1;
        }
        i = i + 1;
    }
```

Fig. 7: The program *matrix_multiplication*

effects. The analysis of a nested loop is therefore reduced to the analysis of a single loop. And thus the precondition calculation for a nested loop is reduced to the calculation for a single loop.

Considering the program in Fig. 7, L_1, L_2 and L_3 are respectively represent the while-statements from outside to inside. Using our approach, we firstly get the execution effect of the innermost loop, $\tau(L_3)$:

$$\left\{ \begin{array}{l} \langle \&c[i][j], (c[i][j] + \prod_0^{p-1} a[i][x] * b[x][j])@p_3\rangle, \\ \langle \&k, p\rangle \end{array} \right\}$$

With $\tau(L_3)$, the loop L_3 is just like a sequence of assignments.

Then, as a summary of L_3, $\tau(L_3)$ is used in the synthesis process of $\tau(L_2)$, and the result execution effect $\tau(L_2)$ is as follows:

$$\left\{ \begin{array}{l} \langle \lambda x.(\&c[i][x])[0, m-1], \\ \qquad (\prod_{x=0}^{m-1} \prod_{y=0}^{p-1} a[i][x] * b[x][y])@p_2\rangle, \\ \langle \&k, p\rangle, \\ \langle \&j, m\rangle \end{array} \right\}$$

Similarly, $\tau(L_1)$ can be obtained.

VIII. In Practice

In this section, we discuss the application of the execution effect and the equivalent expression calculation, namely inferring post-conditions, loop invariants and generating preconditions for loops.

A. Inferring Post-conditions

We can infer post-conditions based on the execution effect of a program statement since it summarizes the memories modified by this statement and the new values in these memories after executing this statement.

Suppose that p represents the program point before a program statement s, then for each tuple $\langle m, v\rangle$ in the execution effect of the statement s, post-conditions of the statement s are generated according to the following rules:

- If m is a SiM, then one post-condition is $*m == v@p$;
- If m is a SeM and of the form $\lambda x.m'[set]$, then one post-condition is $\forall x \in [set]. * m' == v@p$.

Considering the program in Fig. 3, 4 is the program point before the loop statement. The tuple $\langle \&sum, \sum_{y=0}^{s-1} b[y]\rangle$ is among the execution effect of the loop and the memory expression $\&sum$ is a SiM. Based on the first rule, we can generate the post-condition $sum == (\sum_{y=0}^{s-1} b[y])@4$. In the tuple $\langle \lambda x.(\&a[x])[0, s-1], \sum_{y=0}^{x} b[y]\rangle$, the memory expression $\lambda x.(\&a[x])[0, s-1]$ is a SeM. From this tuple, we generate the post-condition $\forall x \in [0, s-1].a[x] == (\sum_{y=0}^{x} b[y])@4$ based on the second rule.

B. Inferring Loop Invariants

Our approach infers loop invariants from post-conditions generated in the last subsection utilizing the framework of automatically generating suitable loop invariants with respect to post-conditions of loops. The framework was proposed and implemented in our previous work [27]. The framework generates loop invariants based on the strategy of substituting some sub-expressions of a given post-condition with some new sub-expressions which equals to the original one at the loop exit program point. Because of space limitation, the details are omitted here.

For instance, the formula $sum == (\sum_{y=0}^{s-1} b[y])@4$ is a post-condition of the loop in Fig. 3, From this post-condition, we can infer the loop invariant $sum == (sum + \sum_{y=0}^{i-1} b[y])@4$ using the framework in [27].

C. Calculating Preconditions

As already mentioned in Definition 4, when an expression e is a post-condition of a program statement s, the equivalent expression of s with respect to e, denoted as e', is actually the precondition of the statement s with respect to the post-condition e. The post-condition e can be proved with the presence of the calculated precondition e'.

IX. Implementation and Case Study

We have implemented the approach proposed in this paper as part of the code verification tool Accumulator, which is a Hoare-style proof system integrated with several automatic and semi-automatic techniques including alias analysis and data flow analysis. We make use of ANTLR as the front-end and Z3 as the back-end SMT solver to implement Accumulator. What should be highlighted here is that we have extended Z3 to deal with user-defined recursive data structures and high-order formulas like lambda expressions. Because of space limitation, the details of the extension of Z3 is not given in this paper.

We have applied our approach to generate assertions including preconditions, post-conditions and loop invariants to ease code verification. Some of the experimental programs are shown in Table V. Column “Data structure” shows the type of the data structure manipulated by the loop statement in each program. Column “Routines” shows operations on the data structure in the loop. The programs operating two-dimensional arrays contain nested loops which can be handled well by our approach. One program name shown in Table V may represent a series of programs. For example, “`count`” counts elements which satisfy some given condition like greater than one element or being non-null, or just counts all the elements which are iterated over. We can successfully synthesize the

TABLE V: Experimental programs

Data Structure	Routines
Closed Integer Intervals	`count, sum, multiply`
One-dimensional Arrays	`copy, assign, count, sum, multiply, search, maximum, minimum`
Two-dimensional Arrays	`copy, assign, count, sum, multiply, search, maximum, minimum, matrix_multiplication`
Acyclic Singly-linked Lists	`assign, count, sum, multiply, search, maximum, minimum`

execution effects of all the program statements of the programs shown in the table and calculate preconditions, post-conditions and loop invariants for all these programs. The process of synthesizing execution effects and generating assertions for each program is less than 5 seconds which includes the time used by Z3.

Next we will illustrate the application of our approach to the verification of a non-trivial program shown in Fig. 8(a). This program sets the i-th element of the one-dimensional array b to the sum of the elements in the i-th line of the two-dimensional array a and computes the sum of all the elements in the array a. The formula $b[i] == (\sum_{x=0}^{n-1} a[i][x])@p_3$ at the program point p_2 and the formula $\forall y \in [0, m-1].b[y] > 0$ at the program point p_4 are respectively the post-conditions of the inner loop and the outer loop. The two post-conditions are among the goals of verifying this program. In this example, we will use lb_1 and lb_2 to respectively represent the inner loop body and the outer loop body.

Firstly, our approach synthesizes the execution effect of the inner loop body lb_1 which consists of two assignments and get the following result:

$$\left\{ \begin{array}{l} \langle \&b[i], b[i] + a[i][j] \rangle, \\ \langle \&j, j+1 \rangle \end{array} \right\}$$

For the inner loop, j is the loop control variable. In the first tuple, the memory address $\&b[i]$ is a FA, and thus it is a memory address modified by the inner loop. The corresponding value $b[i]+a[i][j]$ is of the form $*m+e$, and for any arbitrary value j_0 in the set $\xi(j)$, the predicate $\mathcal{NB}(j, j_0)$ holds. In this case, we can obtain the value stored in $\&b[i]$ after executing the inner loop based on Table IV, and the result value is $b[i] + \sum_{x=0}^{n-1} a[i][x]$. The second tuple is treated in the same way, and hence we get the execution effect of the inner loop shown as follows:

$$\left\{ \begin{array}{l} \langle \&b[i], b[i] + \sum_{x=0}^{n-1} a[i][x] \rangle, \\ \langle \&j, n \rangle \end{array} \right\}$$

Based on the synthesized execution effect, we can generate post-conditions of the inner loop. Since $\&b[i]$ is SiM, the post-condition $b[i] == (b[i]+\sum_{x=0}^{n-1} a[i][x])@p_1$ is generated for the inner loop according to the first rule given in Section VIII-A. Similarly, we can generate the post-condition $j == n$. These two post-conditions are also the intermediate assertions of the outer loop. Through a data flow analysis which is supported in Accumulator, we know that $b[i] == 0$ and $\forall y \in [0, m-1] \forall x \in [0, n-1].a[y][x] == a[y][x]@p_3$ hold at the program point p_1, and hence the post-condition $b[i] == (b[i]+\sum_{x=0}^{n-1} a[i][x])@p_1$ is simplified as $b[i] == (\sum_{x=0}^{n-1} a[i][x])@p_3$ which is the assertion to be proved at the program point p_2.

Then we reduce the outer loop from a nested loop to a single loop by substituting the inner loop shown in box ① in Fig. 8(a) with its execution effect in box ② in Fig. 8(b), and the reduced outer loop is shown in Fig. 8(b). We next analyze the reduced program in Fig. 8(b) to obtain the execution effect of the outer loop body lb_2 and the result is as follows:

$$\left\{ \begin{array}{l} \langle \&b[i], \sum_{x=0}^{n-1} a[i][x] \rangle, \\ \langle \&sum, sum + \sum_{x=0}^{n-1} a[i][x] \rangle, \\ \langle \&j, n \rangle, \\ \langle \&i, i+1 \rangle \end{array} \right\}$$

For the outer loop, i is the loop control variable. In the first tuple, the memory address $\&b[i]$ is a SA, and the condition $\forall i_0 \in [0, m-1].\mathcal{NB}(\&b[i_0], i_0)$ is met. Consequently, for an arbitrary value i_0, $\&b[i_0]$ is evaluated to the same value at the program point p_3 and the program point before the iteration i_0. In this case, $\lambda y.(\&b[y])[0, m-1]$ represents a series of memories modified by the outer loop.

For an arbitrary value i_0, the two conditions $\forall i_0 \in [0, m-1].\mathcal{NB}(\sum_{x=0}^{n-1} a[i_0][x], i_0)$ and $\forall i_0 \in [0, m-1].\mathcal{NA}(*(\&b[i_0], i_0)$ hold, so $\sum_{x=0}^{n-1} a[i_0][x]$ is evaluated to the same value at the program point before the outer loop as the value $*(\&b[i_0])$ evaluated at the program point after the outer loop. In this case, $\sum_{x=0}^{n-1} a[i_0][x]$ is the new value stored in $\&b[i_0]$ in the execution effect of the outer loop, and $\langle \lambda y.(\&b[y])[0, m-1], \sum_{x=0}^{n-1} a[y][x]] \rangle$ is a tuple in the execution effect of the outer loop.

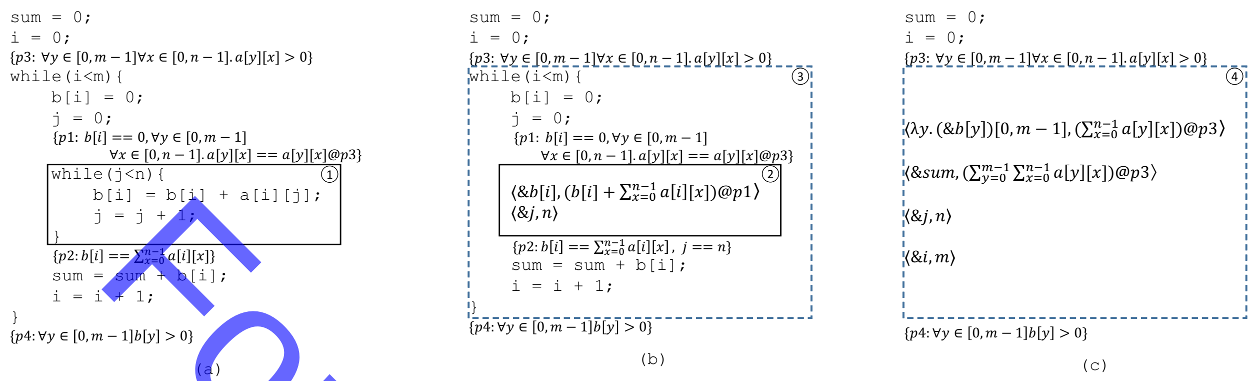
In the third tuple, the memory $\&j$ is a FA and the condition $\epsilon(n, lb_2) == n$ holds, and thus we know that $\&j$ is a memory modified by the outer loop and the value of $\&j$ after executing the outer loop is n.

We deal with the second tuple and the last one just as we do for the two tuples of the execution effect of the inner loop body. The details are omitted here.

After the synthesis process above, we get the following execution effect of the outer loop:

$$\left\{ \begin{array}{l} \langle \lambda y.(\&b[y])[0, m-1], (\sum_{x=0}^{n-1} a[y][x])@p_3 \rangle, \\ \langle \&sum, (\sum_{y=0}^{m-1} \sum_{x=0}^{n-1} a[y][x])@p_3 \rangle, \\ \langle \&j, n \rangle, \\ \langle \&i, m \rangle \end{array} \right\}$$

Then we can reduce the loop shown in box ③ in Fig. 8(b) to a sequence of special assignments shown in box ④ in Fig. 8(c). The special assignments are just the generated execution effect of the outer loop in the original program in Fig. 8(a), and the reduced program is shown in Fig. 8(c). From this, we can see that our approach is able to reduce a loop



```
sum = 0;
i = 0;
{p3: ∀y ∈ [0,m − 1]∀x ∈ [0,n − 1]. a[y][x] > 0}
while(i<m){
    b[i] = 0;
    j = 0;
    {p1: b[i] == 0, ∀y ∈ [0,m − 1]
            ∀x ∈ [0,n − 1]. a[y][x] == a[y][x]@p3}
    while(j<n){                              ①
        b[i] = b[i] + a[i][j];
        j = j + 1;
    }
    {p2: b[i] == ∑_{x=0}^{n−1} a[i][x]}
    sum = sum + b[i];
    i = i + 1;
}
{p4: ∀y ∈ [0,m − 1]b[y] > 0}
```

(a)

```
sum = 0;
i = 0;
{p3: ∀y ∈ [0,m − 1]∀x ∈ [0,n − 1]. a[y][x] > 0}
while(i<m){                                  ③
    b[i] = 0;
    j = 0;
    {p1: b[i] == 0, ∀y ∈ [0,m − 1]
        ∀x ∈ [0,n − 1]. a[y][x] == a[y][x]@p3}
                                             ②
    ⟨&b[i], (b[i] + ∑_{x=0}^{n−1} a[i][x])@p1⟩
    ⟨&j, n⟩

    {p2: b[i] == ∑_{x=0}^{n−1} a[i][x], j == n}
    sum = sum + b[i];
    i = i + 1;
}
{p4: ∀y ∈ [0,m − 1]b[y] > 0}
```

(b)

```
sum = 0;
i = 0;
{p3: ∀y ∈ [0,m − 1]∀x ∈ [0,n − 1]. a[y][x] > 0}
                                             ④
⟨λy. (&b[y])[0,m − 1], (∑_{x=0}^{n−1} a[y][x])@p3⟩

⟨&sum, (∑_{y=0}^{m−1} ∑_{x=0}^{n−1} a[y][x])@p3⟩

⟨&j, n⟩

⟨&i, m⟩

{p4: ∀y ∈ [0,m − 1]b[y] > 0}
```

(c)

Fig. 8: Case study

program to a program without loops by substituting the loops with their execution effects.

Based on the generated execution effect of the outer loop, we can generate the following post-conditions for the loop: $\forall y \in [0, m-1].b[y] == (\sum_{x=0}^{n-1} a[y][x])@p_3$, $sum == (\sum_{y=0}^{m-1}\sum_{x=0}^{n-1} a[y][x])@p_3$, $j == n$ and $i == m$. Among these, the first two post-conditions are the assertions which should be proved to be true in order to prove the correctness of the program in Fig. 8(a).

For the post-condition $\forall y \in [0, m-1].b[y] > 0$ of the outer loop, our approach can calculate its precondition of the outer loop, and the result is $\forall y \in [0, m-1]. \sum_{x=0}^{n-1} a[i][x] > 0$ which is proved to be true using Z3 with the assertion $\forall y \in [0, m-1]\forall x \in [0, n-1].a[y][x] > 0$ being true at the program point p_3. Consequently, the post-condition $\forall y \in [0, m-1].b[y] > 0$ is proved to be true. Similarly, the post-condition $sum == (\sum_{y=0}^{m-1}\sum_{x=0}^{n-1} a[y][x])@p_3$ can be proved.

From the above process, we can see that our approach is effective and practical to help prove the correctness of a program by automatically calculating preconditions and inferring post-conditions.

X. Related Work

Our approach is related to previous works closely in two areas: precondition calculation and summarization.

A. Precondition Calculation

Many precondition calculation techniques for handling loops have been proposed for different purposes. The works [7], [4] generate preconditions of loops that guarantee program termination. The papers [22], [25] derive preconditions for safety assertions in loops. In [20], weakest preconditions of loops are calculated and used for compiler optimizations. The works [10], [19], [14], [23], [2] calculate preconditions to verify the correctness of a program, which are similar to us.

Many works on the precondition calculation for loops are based on de-sugaring loops with invariants. In [10], [19], preconditions for loops with respect to given post-conditions are calculated by de-sugaring loops with loop invariants. In [23], preconditions of loops are computed or approximated by identifying invariant relations [21] of a while loop. The method in [2] removes all back edges in the control-flow graph of unstructured programs to eliminate loops. Loop invariants are added to guard loop bodies. A great number of techniques have been proposed to automatically infer loop invariants, see e.g., [6], [24], [29], [12], [17], [3]. Nevertheless, automatic generation of loop invariants itself is still a complicated problem which often requires ingenuity and human invention. Unlike all of them, our approach can calculate preconditions for loops without the help of loop invariants.

In [14], loops are first transformed into acyclic programs by bounding the number of loop iterations, which may lose accuracy. By comparison, our approach calculates preconditions based on the automatically-synthesized execution effects of program statements, which are accurate.

The work [28] generates preconditions for both intermediate assertions and post-conditions of a loop based on proposed heuristics while this paper summarizes a loop as modified memories of the loop and the new values after executing the loop, and then calculate preconditions based on the summary.

B. Summarization

Automatic loop summarization has been discussed in numerous papers. The work [11] presents an entirely-dynamic method to summarize a loop as preconditions and post-conditions while our work is completely static and the summary result is represented as memories modified by a loop and the new values after executing the loop. By contrast, our summary is more general and has wider applications including inferring preconditions and post-conditions which are the summary result of [11]. For termination analysis, [26] summarizes loops based on abstract interpretation [5] , while [26] summarizes a loop by computing symbolic abstract transformers with respect to a set of abstract domains. Unlike [11], [26], the purpose of our summary is to automatically generate assertions for program verification.

Many previous works have attempted to summarize pointer programs. The work [18] presents a local inter-procedural shape analysis to summarize procedures as transformers of procedure-local heaps. In [15], an inter-procedural dataflow analysis technique is proposed to summarize procedure effects with procedure contracts which are graph transformations capturing the overall effect of a procedure. Similarly, [13] employs graph grammars to abstract pointer operations. Unlike these works, our approach summarizes program statements operating pointers as memories modified by them and the new values stored in these memories after executing the statements. Such a summary reflects the result of executing a statement, which facilitates the program analysis and code verification.

XI. Conclusion

Based on statistic results on loop statements that manipulates commonly-used data structures from real-world programs, this paper (1) specifies the execution effect of a statement as memories modified by the statement and new values of these memories after executing this statement; (2) introduces a novel and efficient way to summarize a loop statement with its execution effect, which reduces a loop statement into a sequence of assignments, making it easier to analyze loop statements and generate preconditions for them; (3) proposes an approach to synthesize execution effects of different types of program statements, including assignments, sequential statements, conditional statements and while-loops manipulating commonly-used data structures; (4) presents a method to generate preconditions, post-conditions and loop invariants for a program statement based on its execution effect; (5) demonstrates our implemented tool, and its evaluated results. The results show that the proposed approach improves the efficiency and quality of code verification by generating assertions including pre-conditions, post-conditions and loop invariants automatically and reducing human involvement, which has great practicability and significance.

Future Work. To ease the code verification task for more programs, we will extend our work in several aspects. Firstly, we will attempt to cover non-linear data structures like binary search trees. Secondly, we are considering dealing with more kinds of loops such as enhanced foreach loops over collections.

XII. Acknowledgement

We thank the anonymous reviewers for their constructive comments. This research was partially supported by National Key Basic Research Program of China No.2014CB340703, National Science Foundation of China No.61561146394, No.91318301, No.91418204, No.61321491 and No.61472179.

References

[1] Accumulator. http://seg.nju.edu.cn/toolweb.

[2] M. Barnett and K. R. M. Leino. Weakest-precondition of unstructured programs. In *ACM SIGSOFT Software Engineering Notes*, volume 31, pages 82–87. ACM, 2005.

[3] D. Beyer, T. A. Henzinger, R. Majumdar, and A. Rybalchenko. Invariant synthesis for combined theories. In *Verification, Model Checking, and Abstract Interpretation*, pages 378–394. Springer, 2007.

[4] M. Bozga, R. Iosif, and F. Konečnỳ. Deciding conditional termination. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 252–266. Springer, 2012.

[5] A. Chawdhary, B. Cook, S. Gulwani, M. Sagiv, and H. Yang. Ranking abstractions. In *Programming Languages and Systems*, pages 148–162. Springer, 2008.

[6] S. Conchon, A. Goel, S. Krstić, A. Mebsout, and F. Zaïdi. Invariants for finite instances and beyond. In *Formal Methods in Computer-Aided Design (FMCAD)*, pages 61–68, 2013.

[7] B. Cook, S. Gulwani, T. Lev-Ami, A. Rybalchenko, and M. Sagiv. Proving conditional termination. In *Computer Aided Verification*, pages 328–340. Springer, 2008.

[8] L. De Moura and N. Bjørner. Z3: An efficient smt solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[9] E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, 1975.

[10] C. Flanagan and J. B. Saxe. Avoiding exponential explosion: Generating compact verification conditions. In *SYMPOSIUM ON PRINCIPLES OF PROGRAMMING LANGUAGES*, pages 193–205. ACM, 2001.

[11] P. Godefroid and D. Luchaup. Automatic partial loop summarization in dynamic test generation. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, pages 23–33. ACM, 2011.

[12] D. Gopan, T. Reps, and M. Sagiv. A framework for numeric analysis of array operations. *ACM SIGPLAN Notices*, 40(1):338–350, 2005.

[13] J. Heinen, C. Jansen, J.-P. Katoen, and T. Noll. Juggrnaut: using graph grammars for abstracting unbounded heap structures. *Formal Methods in System Design*, 47(2):159–203, 2015.

[14] I. Jager and D. Brumley. Efficient directionless weakest preconditions. Technical report, Technical Report CMU-CyLab-10-002, Carnegie Mellon University, CyLab, 2010.

[15] C. Jansen and T. Noll. Generating abstract graph-based procedure summaries for pointer programs. In *Graph Transformation*, pages 49–64. Springer, 2014.

[16] Z. Jianhua and L. Xuandong. Scope logic: An extension to hoare logic for pointers and recursive data structures. In *Theoretical Aspects of Computing–ICTAC 2013*, pages 409–426. Springer, 2013.

[17] L. Kovács and A. Voronkov. Finding loop invariants for programs over arrays using a theorem prover. In *Fundamental Approaches to Software Engineering*, pages 470–485. Springer, 2009.

[18] J. Kreiker, T. Reps, N. Rinetzky, M. Sagiv, R. Wilhelm, and E. Yahav. Interprocedural shape analysis for effectively cutpoint-free programs. In *Programming Logics*, pages 414–445. Springer, 2013.

[19] K. R. M. Leino. Efficient weakest preconditions. *Information Processing Letters*, 93(6):281–288, 2005.

[20] N. P. Lopes and J. Monteiro. Weakest precondition synthesis for compiler optimizations. In *Verification, Model Checking, and Abstract Interpretation*, pages 203–221. Springer, 2014.

[21] A. Louhichi, O. Mraihi, L. L. Jilani, and A. Mili. Invariant assertions, invariant relations, and invariant functions. In *WING 2009 WORKSHOP ON INVARIANT GENERATION*, page 60. Citeseer, 2009.

[22] Y. Moy. Sufficient preconditions for modular assertion checking. In *Verification, Model Checking, and Abstract Interpretation*, pages 188–202. Springer, 2008.

[23] O. Mraihi, W. Ghardallou, A. Louhichi, L. L. Jilani, K. Bsaies, and A. Mili. Computing preconditions and postconditions of while loops. In *Theoretical Aspects of Computing–ICTAC 2011*, pages 173–193. Springer, 2011.

[24] C. S. Păsăreanu and W. Visser. Verification of java programs using symbolic execution and invariant generation. In *Model Checking Software*, pages 164–181. Springer, 2004.

[25] M. N. Seghir and D. Kroening. Counterexample-guided precondition inference. In *Programming Languages and Systems*, pages 451–471. Springer, 2013.

[26] A. Tsitovich, N. Sharygina, C. M. Wintersteiger, and D. Kroening. Loop summarization and termination analysis. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 81–95. Springer, 2011.

[27] J. Zhai, H. Wang, and J. Zhao. Post-condition-directed invariant inference for loops over data structures. In *Software Security and Reliability-Companion (SERE-C), 2014 IEEE Eighth International Conference on*. IEEE, 2014.

[28] J. Zhai, H. Wang, and J. Zhao. Assertion-directed precondition synthesis for loops over data structures. In *Dependable Software Engineering: Theories, Tools, and Applications - First International Symposium, SETTA 2015, Proceedings*, pages 258–274, 2015.

[29] L. Zhang, G. Yang, N. Rungta, S. Person, and S. Khurshid. Feedback-driven dynamic invariant discovery. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pages 362–372. ACM, 2014.