A Unified Strategy for Formal Derivation and Proof of Binary Tree Nonrecursive Algorithms

0 Introduction

The binary tree, as a classic nonlinear data structure, provides regular data storage and supports powerful search algorithms. Non-recursive algorithms are more efficient in terms of computation time and storage space\(^1,2\). As a result, it is significant to formally derive and prove the binary tree’s nonrecursive algorithm\(^3,4\).

Loop invariants serve as the foundation for understanding, deriving, and proving the loop program in formal derivation\(^5\). The loop invariants of the binary tree nonrecursive algorithm, however, are difficult to describe. In the past, Gries\(^6\) and Dijkstra\(^7\) constructed a loop invariant for the nonrecursive preorder binary tree traversal algorithm. This loop invariant has a complicated logical relationship and a laborious expression. Xue\(^5\) identified the limitations of the traditional standard strategy for developing loop invariants and proposed two new strategies based on the stack’s recursive relation.

We decompose the binary tree problems and discover two recursive solving sequences: stack and queue. For the recursive relation, a unified formal derivation and proof strategy is proposed. The entire refinement process is realized, from the abstract specification to the executable program.

Section 1 of this paper presents and compares related work. Sections 2 and 3 present the derivation strategies for the binary tree queue and stack recursive relations, respectively. Section 4 validates the strategies’ effectiveness by using binary tree inorder tra-
versal and hierarchical traversal as examples. Conclusion is given in Section 5.

1 Related Work

Loginov et al[5] used destructive pointer manipulation to validate the Deutsch-Schorr-Waite (DSW) algorithm for traversing a binary tree, and the Test Vector Leakage Assessment (TVLA) method to validate the DSW algorithm’s correctness. However, the DSW algorithm verification requires an analysis of the set of specified node relations, which is more difficult.

Qin et al[6] used the HIP/SLEEK to formally verify a highly balanced binary tree and binary search tree, but there are numerous constraints on the synthesis of loop invariants. You et al[7] built loop invariants based on the stack’s recursive relation and verified the correctness of three types of nonrecursive algorithms for binary trees, but they did not address the generation of executable programs.

Based on the recursive relationship of the stack, Xue[8] proposed two new strategies for developing loop invariants. Based on the two new strategies for developing loop invariants, we propose a unified strategy for the formal derivation and proof of binary tree nonrecursive algorithms. Our strategy simplifies program derivation, improves formal derivation effectiveness, and ultimately produces the correct executable program.

2 Derivation Strategy for the Binary Tree Stack Recursive Relation Problem

This paper proposes a formal derivation and proof strategy for solving the binary tree problem, whose recursive relation sequence is the stack, based on Ref. [10]. A stack is represented by the sequence $S[0...#S−1]$[11]. We define $S[#S−1]$ as the top of the stack and $S[0]$ as the bottom. The following sequence of operations can then be used to implement a stack’s common operations:

- Test whether stack $S$ is empty: $#S = 0$
- Take the top element of stack $S$: $X = S[#S−1]$
- Implement a push operation on stack $S$: $S = S↑[X]$
- Perform an output operation on the stack $S$: $S = S[0...#S−2]$

A general loop invariant template for the binary tree stack recursive relation is obtained by deriving and proving three nonrecursive algorithms for traversing a binary tree in preorder, inorder, and postorder as follows:

$\rho$: $\text{Order}(T) = X↑\text{Order}(q)↑F(S)$

$\text{Order}(T)$ denotes the node stack generated by the non-recursive algorithm for each traversal of the binary tree $T$; $X$ stores a portion of the node stack generated during the traversal; $q$ stores the subtrees of $T$ being visited; $S$ is a stack variable that acts on the stack as a whole; $F(S)$ represents the traversal result of the subtrees to be traversed:

$F([q]↑S) = \begin{cases} \text{Order}(q)↑F(S) & \text{preorder} \\ \text{Order}(q,r)↑F(S) & \text{inorder} \\ \text{Order}(q,r)↑[q,d]↑F(S) & \text{postorder} \end{cases}$

In this paper, we formally derive and prove non-recursive algorithms for traversing a binary tree in preorder, inorder, and postorder. At the same time, we developed additional binary tree non-recursive algorithms. These various algorithms can obtain the corresponding loop invariant based on the loop invariant template. Furthermore, the difference between these loop invariants is dependent on the definition of $F(S)$.

3 Derivation Strategy for the Binary Tree Queue Recursive Relation Problem

We discover that the recursive relationship of the solving sequence can also be a queue by decomposing the binary tree problems and finding the recursive relationship. Furthermore, we summarize six steps of nonrecursive algorithm derivation of queue recursive relations through formal derivation and proof of several examples.

3.1 Derivation Steps

Step 1 The goal of the work on the binary tree queue’s recursive relation problem is clarified by constructing the formal specification. The program’s formal specification consists of pre-assertion ($Q$) and post-assertion ($R$). $Q$ denotes the condition that a program’s input parameters must satisfy. $R$ denotes the condition that the program’s final solution must satisfy[12].

Step 2 Decomposition generates the form of the algorithm or recursive relations for binary tree queue problems[13], and different algorithms have different decompositions. We can get ideas for the binary tree problem decomposition from the formal specification in Step 1. A certain number of subtree problems can be obtained by decomposing the binary tree problem. The subtree prob-
lem is then decomposed in the same manner until each subtree problem has been solved.

Step 3 We can confirm recursive relationships of solving sequences and loop invariants by analyzing the mathematical properties of binary tree problems. Predicates can then be used to express recursive relations and loop invariants. A queue can also be thought of as a sequence \( q[0...\#q-1] \). The queue’s head is \( q[0] \), and its tail is \( q[\#q-1] \). Furthermore, the following sequence operations can be used to implement the common operations of a queue:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test whether queue ( S ) is empty</td>
<td>( #q = 0 )</td>
</tr>
<tr>
<td>Take the top element of queue ( S )</td>
<td>( X = q[0] )</td>
</tr>
<tr>
<td>Implement a push operation on queue ( S )</td>
<td>( q = q \uparrow [X] )</td>
</tr>
<tr>
<td>Perform an output operation on the queue ( S )</td>
<td>( q = q[1...] )</td>
</tr>
</tbody>
</table>

The binary tree queue’s recursive relation can be expressed as a corresponding recursive relation expression. That is, the quantifier transformation method is used to derive the recursive relation \( S = F(S) \) for the queue, and the initial values are assigned to the functions and variables.

Step 4 The loop invariants of the algorithm program are obtained using the recursive relation expression constructed in Step 3 and the recursive definition technique of the loop invariants. It is also worth noting that we use the recursive definition technique to define the contents of a queue based on the relationship between the elements in the variables and the recursive relationship of the subtrees.

Step 5 The Apla program of an algorithm is derived from the recursive relation expression in Step 3 and the loop invariant in Step 4. Then, we use the Dijkstra-Gries standard proving technique to prove the Apla program’s correctness.

Step 6 The Apla to C++ automated generating system can transform the nonexecutable Apla program from Step 5 into an executable program.

3.2 General Loop Invariant Template for the Binary Tree Queue Recursive Relation Problem

In this section, we first define and develop the loop invariant of the binary tree queue recursion relation using the recursive definition technique.

Definitions of loop invariant are as follows:

Definition 1 Given a loop do and its set \( A \) of all loop variables. An assertion is said to be a loop invariant of the loop do if it reflects the variation law of each element in \( A \) and is consistently true before and after each iteration.

Definitions of loop variables are as follows:

Definition 2 The loop variable is the value of a variable that changes with the loop body. Set \( A \) contains these loop variables. The three most common loop variables are \( X, S \), and \( q \). The queue variable \( X \) is used to store the sequence of node flags that have been visited in the binary tree queue recursive relation problems. \( S \) is a queue that contains nodes that have yet to be visited. \( q \) is used to keep track of which nodes are being visited.

The following strategies for developing loop invariants are based on the recursive relationship of the solving sequence being the queue:

Strategy 1: Based on the loop program correctness verification conditions, we investigate the pre-assertion \( (Q) \) of the loop and post-assertion \( (R) \) on termination, analyze background knowledge, mathematical properties of the problem to be solved by the program, and program properties, and describe the variation laws of all loop variables using induction reasoning. These laws are the loop invariants that we require.

Strategy 2: The general strategy and all required loop variables for the binary tree queue recursive relation problem are generated using a reliable algorithm design method. To begin, we describe the variation laws of each variable, as well as the laws required to make the loop invariant. In addition, if the number of the subsolutions in the recursive relation is greater than 1, one sequence variable that will be used as a queue or a set variable must be added, and the sequence’s content is defined recursively.

We obtain the general loop invariant template by deriving and proving binary tree non-recursive algorithms, such as hierarchical binary tree traversal, binary tree depth calculation, and perfect binary tree judgement:

\[
\rho: \text{Inv} \land \text{Cons}
\]

where

\[
\text{Inv} = \text{Lay}(T) = X \uparrow [q. d] \uparrow F(S \uparrow [q. l] \uparrow [q. r])
\]

Cons denotes the assertion that the conditions are satisfied by conjunction based on the characteristics and requirements of various problems. The variable \( X \) stores the sequence of node flags visited during the traversal, \( q \) stores the subtrees of \( T \) that are being visited, and \( S \) is a queue stores the nodes that have not yet been visited. The traversal result of the subtree to be traversed is represented by \( F(S) \). It should be noted, however, that Inv is the loop invariant representing the binary tree’s hierarchical traversal algorithm. To summarize the loop vari-
ant template, we use the hierarchical traversal algorithm as the parent algorithm. The loop invariants for hierarchical binary tree traversal ($\rho_1$), calculating the depth of a binary tree ($\rho_2$), and judging the perfect binary tree ($\rho_3$) are as follows:

$$\rho_1: \text{Inv}$$
$$\rho_2: \text{Inv}/\text{Flag} = \{ (\text{num num} = \#(S) - 1) \}$$

$$\rho_3: \text{Inv}/\text{Flag} = \forall (a: a \in X): ((a.l \neq \% \land (a.r \neq \%)) \lor ((a.l = \% \land (a.r = \%))))$$

We have developed other binary tree non-recursive algorithms in addition to the hierarchical traversal of a binary tree, calculating the depth of a binary tree, and judging the perfect binary tree. According to our derivation experiments, these problems can be derived using the above loop invariant template.

4 Case Studies

By decomposing the binary tree problems, we discover that there are two recursive solving sequences for the binary tree: stack and queue. We will use the inorder traversal of a binary tree and the hierarchical traversal of a binary tree as examples in this section. Inorder binary tree traversal uses the stack recursive relation problem strategy, while hierarchical binary tree traversal uses the queue recursive relation problem strategy.

4.1 Inorder Traversal of a Binary Tree

- Derivation

The stack structure is used to simulate recursion during the binary tree’s inorder traversal. Naturally, the preorder and postorder traversals are analogous to the in-order traversal. The Apla algorithm program is derived from the formal derivation of a nonrecursive inorder traversal algorithm. It is useful for program development and can be easily converted into a variety of executable programs. The derivation process consists of the five steps listed below[16]:

Step 1 Constructing the algorithm program’s formal specification.

Step 2 Decomposing the original problem and obtaining $\text{Inv}(T) = \text{Inv}(T.\downarrow) \uparrow [T. d] \uparrow \text{Inv}(T.\uparrow)$.

Step 3 Identifying recursive relations and creating an overall strategy for solving the inorder traversal of a binary tree using non-recursive algorithms.

a) Set three variables: $X, S$, and $q$.

b) The stack variable $X$ stores the sequence of visited node flags. After completing the $T$ traversal, $X = \text{Lay}(T)$.

c) The root of each subtree and its right subtree of a finite binary tree $T$ are stored in $S$.

d) The subtrees of $T$ that are traversed are stored in $q$.

Step 4 For binary tree stack recursive relation problems, we employ loop invariant development strategies. To form the required loop invariant, $X, S$, and $q$ satisfy the following equation:

$$\rho: \text{Inv}(T) = X \uparrow \text{Inv}(q) \uparrow F(S)$$

Step 5 The Apla algorithm program is derived from the recursive relation and loop invariant of a binary tree’s inorder traversal:

Apla Algorithm Program: Binary Tree’s Inorder Traversal

```plaintext
procedure hierarchical (T: btree(char, 40); var X: list(char, 40));
2 PQ: given a finite binary tree T;
3 PR: X = In(T);
4 var S: list(char, 40); var q: btree(char, 40);
5 begin
6 X, S, q = [], [], T;
7 \rho: \text{Inv}(T) = X \uparrow \text{Inv}(q) \uparrow F(S);
8 do (q*%a) \rightarrow 
9 q, S: = q, l, [q] \uparrow S;
10 while (S* []) \rightarrow 
11 q, S, X: = S[h].r, S[h+1..t], X \uparrow [S[h].d];
12 od;
13 end
```

- Formal proof

Since the Apla program may contain errors, we use the Dijkstra-Gries standard proving technique to prove its correctness.

1) Before executing the loop, prove that $\rho$ is true.

The $Q$ in the assertion is the $Q$ of the entire program, not the $Q$ of the do statement. To confirm that $\rho$ is true before executing the do statement, i.e. $Q \Rightarrow \text{wp}(S_o, \rho)$, we should verify the following assertion:

Proof: $Q \Rightarrow \text{wp}(S_o, \rho)$

Statement $S_o$: $X, S, q: = [], [], T$

$\Rightarrow \text{Inv}(T) = (X \uparrow \text{Inv}(q) \uparrow F(S))[X, S, q \setminus [], [], T]$

$\Rightarrow \text{Inv}(T) = [] \uparrow \text{Inv}(T) \uparrow F([])$

$\Rightarrow \text{Inv}(T) = \text{Inv}(T)$

$\Rightarrow \text{true}$

By assigning the three variables in $S_o$ to $\rho$, we can prove $Q \Rightarrow \text{wp}(S_o, \rho)$. Obviously, the preceding assertion
is true.

2) Prove that $\rho$ is the loop invariant.

In the loop body, for the first conditional clause:

$\neg (q = %) \rightarrow q, S = q, l, [q] \uparrow S$

We need to verify that the following assertion is true:

$\rho \land \neg (q = %) \Rightarrow \wp(q, S = q, l, [q] \uparrow S, X \uparrow \ln(q) \uparrow F(S))$

The following assertion must be guaranteed:

Proof: $\rho \land C_1 \Rightarrow \wp(S, \rho)$

Statement $S_1$: $q, S = q, l, [q] \uparrow S$

Condition $C_1$: $q \neq %$

$\Rightarrow \ln(T) = X \uparrow \ln(q) \uparrow F(S) \land q = % \Rightarrow (X \uparrow \ln(q) \uparrow F(S)) | q, S = q, l, [q] \uparrow S$.

$\Rightarrow \ln(T) = X \uparrow \ln(q) \uparrow F(S) \land q = % \Rightarrow X \uparrow \ln(q) \uparrow F(S)$

$\Rightarrow \ln(T) = X \uparrow \ln(q) \uparrow F(S) \land q = % \Rightarrow X \uparrow \ln(q) \uparrow F(S)$

$\Rightarrow \ln(T) = X \uparrow \ln(q) \uparrow F(S) \land q = % \Rightarrow X \uparrow \ln(q) \uparrow F(S)$

We need to verify that the following assertion is true:

$\rho \land q \Rightarrow \wp(q, S, X = S[h], r, S[h+1..t], X \uparrow [S[h], d])$

Then the following assertion must be guaranteed:

Proof: $\rho \land C_2 \Rightarrow \wp(S, \rho)$

Statement $S_2$: if conditional statement

Condition $C_2$: $q = % \land S \neq []$

$\Rightarrow \ln(T) = X \uparrow \ln(q) \uparrow F(S) \land q = % \Rightarrow (X \uparrow \ln(q) \uparrow F(S)) | q, S = S[h], r, S[h+1..t], X \uparrow [S[h], d] | d, X \uparrow [g, d] \uparrow F(S) \uparrow [q, l] \uparrow [g, r])$

$\Rightarrow \ln(T) = X \uparrow \ln(q) \uparrow F(S) \land q = % \Rightarrow X \uparrow [S[h], d] \uparrow \ln[S[h], r, F(S) \land S[h] \neq [] \Rightarrow X \uparrow F(S)]$

$\Rightarrow \ln(T) = X \uparrow \ln(F(S) \land S[h] \neq [] \Rightarrow X \uparrow F(S))$

$\Rightarrow \ln(T) = X \uparrow \ln(F(S) \land S[h] \neq [] \Rightarrow X \uparrow F(S)$

$\Rightarrow \ln(T) = X \uparrow \ln(F(S) \land S[h] \neq [] \Rightarrow X \uparrow F(S)$

$\Rightarrow \ln(T) = X \uparrow \ln(F(S) \land S[h] \neq [] \Rightarrow X \uparrow F(S)$

$\Rightarrow \ln(T) = X \uparrow \ln(F(S) \land S[h] \neq [] \Rightarrow X \uparrow F(S)$

By assigning two variables in $S$, to $\rho$, we can prove $\rho \land C_2 \Rightarrow \wp(S, \rho)$. The second conditional clause obviously holds.

3) Prove that at the end of the loop, the post-assertion $R$ must be true.

The following assertion must be guaranteed to ensure that $R$ is true and $\rho \land \neg (C_1 \cup C_2) \Rightarrow R$:

Proof: $\rho \land \neg (C_1 \cup C_2) \Rightarrow R$

4) The termination of the loop clearly holds.

In summary, the preceding steps prove the Apla program's correctness for binary tree inorder traversal.

- **C++ generation system**

  The Apla program can be converted automatically into C++ executable programs using the Apla to C++ program automatic generation system developed by our team [1]. As shown in Fig. 1, the Apla program for the nonrecursive algorithm for inorder traversal of the binary tree is on the left, and the converted C++ program is on the right. We put the C++ program to the test by feeding it a 10-node binary tree (Fig. 2). As shown in Fig. 3, the result satisfies the output of the inorder traversal binary tree from small to large and left to right.

As a result of testing, the generated C++ program is correct. Furthermore, when we compare the Apla program on the left with the C++ program on the right (Fig. 1), we can clearly see that the Apla program is much simpler. There are only four key lines of code, which greatly improves the efficiency of the algorithm and the program development.

### 4.2 Hierarchical Traversal of a Binary Tree

- **Derivation**

  **Step 1** Specification of the algorithm program.

  We assume $T$ is a finite binary tree and $\text{Lay}(T)$ is the sequence of nodes obtained by traversal $T$ hierarchically. When the sequence of visited node flags is stored in the sequence variable $X$, the algorithm specification for this problem is as follows:

  $\text{AQ} \leftarrow \{1: \text{list}(\text{char}, 40); T: \text{btree}(\text{char}, 40)\} | \{\text{AR}: \text{X} = \text{Lay}(T)\}$

  **Step 2** Identifying recursive relations.

  Recursive relations make it easier to write recursive algorithmic programs. We use the following derivation to get a nonrecursive algorithm program:

  $\text{Lay}(T) = [T, d] \uparrow [T, l, d] \uparrow [T, r, d] \uparrow [T, l, l, d] \uparrow [T, l, r, d] \uparrow [T, r, l, d] \uparrow [T, r, r, d] \uparrow \ldots$

  As a result, we can obtain the overall strategy for solving the nonrecursive binary tree hierarchical traversal algorithm:

  a) Set three variables: $X$, $S$, and $q$. 

b) The sequence variable \( X \) stores the sequence of visited node flags. After completing the \( T \) traversal, \( X = \text{Lay}(T) \).

c) \( S \) is a queue used to store the nodes to be visited.

d) \( q \) is used to store the nodes being visited.

The node to be visited is the first element in the queue variable \( S \). After it has been visited, its neighbors who have not yet been traversed will be added to the queue of nodes to be visited. As a result, the function \( F \) defines the contents of \( S \) as follows:

\[
F([\ ]) = [ \ ] \quad (I)
\]
\[
F(S) = [S. h] \uparrow F([S[h+1:]]) \uparrow [S[h]. l] \uparrow [S[h]. r] \quad (II)
\]

When \( q \) is not visited:

\[
F([q \uparrow S]) = [q. d] \uparrow F([S] \uparrow [q. l] \uparrow [q. r]) \quad (III)
\]

When \( q \) has been visited:

\[
F([q \uparrow S]) = F([S]) \quad (IV)
\]

**Step 3** Constructing the loop invariant.

It is hard to develop the loop invariant of binary tree queue recursive relation problems by the traditional methods\(^{[6,7]}\), and the expression of the resulting loop invariant is complicated. As a result, formal derivation and proof of the algorithm program will be difficult.

In this section, we use the loop invariant development strategy for binary tree queue recursive relation problems, specifically the recursive definition technique in Strategy 2. Finally, we will obtain a simple expression.

---

**Fig.1** Apla to C++ program generation system generates the C++ program for inorder traversal

**Fig.2** Inorder traversal binary tree algorithm example

**Fig.3** Test result of C++ program for binary tree inorder traversal
for the loop invariant:
\[
\rho: \text{Lay}(T) = X \uparrow [q. d] \uparrow F(S \uparrow [q. l] \uparrow [q. r])
\]

The recursive relation (II) and the loop invariant (V) can be used to derive the Apla program:

```
procedure hierarchical (T: btree(char, 40); var X: list(char, 40));
PQ: given a finite binary tree T;
PR: X = Lay(T);
var S: list (char, 40); var q: btree (char, 40);
begin
X, S, q: = [], [], T;
rho: Lay(T) = X \uparrow [q. d] \uparrow F(S \uparrow [q. l] \uparrow [q. r]);
do (q \neq \%) ->
S, X, q: = S \uparrow \% \uparrow [q. r] \uparrow [q. d], %;
[] (S \neq \%)->
q, S: = S[h], S[h+1..];
od;
end
```

The Apla program was derived from the resulting algorithm and the loop invariant. It perfectly reflects Apla's functional abstraction, data abstraction, and other modern programming concepts. In general, the Apla program for nonrecursive binary tree hierarchical traversal algorithm is simple and practical, and it can be easily converted into a variety of executable programs.

- **Formal proof**

  We use the Dijkstra-Gries standard proof technique to prove the correctness of the Apla algorithm given above, as we did in Section 4.1.

  1) Before executing the loop, prove that \( \rho \) is true.

  To confirm that \( \rho \) is true before executing the `do` statement, i.e. \( Q \models wp(S_o, \rho) \), we should verify the following assertion:

  Proof: \( Q \models wp(S_o, \rho) \)

  Statement \( S_o: X, S, q: = [], [ ], T \)

  \( \equiv \text{Lay}(T) = (X \uparrow [q. d] \uparrow F(S \uparrow [q. l] \uparrow [q. r])[X, S, q \ \{\ }, [ ], T]) \)

  \( \equiv \text{Lay}(T) = [ ] \uparrow [T. d] \uparrow F([T. l] \uparrow [T. r]) \)

  \( \equiv \text{Lay}(T) = \text{Lay}(T) \)

  {Use: recursive relation (IV)}

  \( \equiv \text{true} \)

  By assigning the three variables in \( S_o \) to \( \rho \), we can prove \( Q \models wp(S_o, \rho) \). Obviously, the preceding assertion is true.

  2) Prove that \( \rho \) is the loop invariant.

  In the loop body, for the first conditional clause:

  \( \neg (q = \%) \rightarrow S, X, q: = S \uparrow [q. l] \uparrow [q. r], X \uparrow [q. d], \%

  We need to verify that the following assertion is true:

  \( \rho \Lambda \neg (q = \%) \Rightarrow wp(S, X, q: = S \uparrow [q. l] \uparrow [q. r], X \uparrow [q. d], \%

  Then the following assertion must be guaranteed:

  Proof: \( \rho \Lambda C_o \Rightarrow wp(S_o, \rho) \)

  Statement \( S_o: S, X, q: = S \uparrow [q. l] \uparrow [q. r], X \uparrow [q. d], \%

  Condition \( C_o: q \neq \% \)

  \( \equiv \text{Lay}(T) = X \uparrow [q. d] \uparrow F(S \uparrow [q. l] \uparrow [q. r]) \uparrow q \neq \%

  \( \Rightarrow (X \uparrow [q. d] \uparrow F(S \uparrow [q. l] \uparrow [q. r]))[S, X, q \ \{\ ] \uparrow [\]) \uparrow \)

  \( \Rightarrow \text{Lay}(T) = X \uparrow [q. d] \uparrow F(S \uparrow [q. l] \uparrow [q. r]) \uparrow q \neq \%

  \( \Rightarrow X \uparrow F(S) \)

  {Use: recursive relation (V)}

  \( \equiv \text{true} \)

  By assigning two variables in \( S_o \) to \( \rho \), we can prove \( \rho \Lambda C_o \Rightarrow wp(S_o, \rho) \). The first conditional clause obviously holds.

  In the loop body, for the second conditional clause:

  \( q = \% \Rightarrow q \neq \%

  We need to verify that the following assertion is true:

  Proof: \( \rho \Lambda C_o \Rightarrow wp(S_o, \rho) \)

  Statement \( S_o: S, X, q: = S[h], S[h+1..]; \%

  Condition \( C_o: q \neq \%

  \( \Rightarrow \text{Lay}(T) = X \uparrow [q. d] \uparrow F(S \uparrow [q. l] \uparrow [q. r]) \uparrow S[h], S[h+1..]; \%

  \( \Rightarrow X \uparrow [S[h], d] \uparrow F(S[h+1..]) \uparrow [S[h], l] \uparrow [S[h], r]) \)

  {Use: Recursive relation (IV) and recursive relation (V)}

  \( \equiv \text{true} \)

  By assigning two variables in \( S_o \) to \( \rho \), we can prove \( \rho \Lambda C_o \Rightarrow wp(S_o, \rho) \). The second conditional clause obviously holds.
3) Prove that the post-assertion $R$ must be true at the end of the loop.

The following assertion must be guaranteed to ensure that $R$ is true and $\rho \land \neg (C_1 \lor C_2) \Rightarrow R$:

**Proof:**

If $\rho \land \neg (C_1 \lor C_2) \Rightarrow R$ holds.

Given:

$$S = [ \neg ] \Rightarrow \text{Lay}(T) = X$$

Then:

$$\text{Lay}(T) = X \land q = \% \land S = [ ] \Rightarrow \text{Lay}(T) = X$$

{Use: recursive relation (V)}

$\equiv$ true

4) The termination of the loop clearly holds.

In summary, the preceding steps prove the Apla program’s correctness for binary tree hierarchical traversal.

- **C++ generation system**

As shown in Fig. 4, the Apla program for the non-recursive algorithm for hierarchical traversal of the binary tree is on the left, and the converted C++ program is on the right. The generated C++ program was tested with a binary tree input of 10 nodes (as shown in Fig. 2), and the test results are shown in Fig. 5. As a result, the test result indicates that the C++ program is correct, and we have achieved a complete refinement of the process from abstract specification to the concrete executable program.

```cpp
void main() {
    cout << "Hierarchical traversal binary tree algorithmrente" << size1 << "nodes=" << endl;
    t.Copy(tempTree0.SetEmpty());
    i = 1;
    t.Copy(cin); //cin
    t.Copy(tempTree1.SetEmpty());
    i = i + 1;
}
else
    break;
while(true); //cin
S.Copy(tempList1.SetEmpty());
X.Copy(tempList0.SetEmpty());
q.Copy();
if (i < size1) {
    S.Copy(tempTree1.SetEmpty());
    X.Copy(tempList1.SetEmpty());
    q.Copy(tempTree1.SetEmpty());
}
```

In Fig. 4 Apla to C++ program generation system generates C++ program for hierarchical traversal

In Fig. 5 Test result of C++ program for binary tree hierarchical traversal

5 Conclusion

We decompose the binary tree problem to find recursive relations and present a unified formal derivation and proof strategy. We investigate the similarities and differences between the two types of loop invariants and construct general loop invariant templates for the binary tree’s queue and stack recursive relations. The non-recursive Apla algorithm is then derived from the problem specification by deriving the recursive relation expression and loop invariant, and the correctness of the Apla program is proven using the Dijkstra-Gries standard proving technique. Finally, the tool is used to convert the Apla abstract program into a C++ executable program. This paper has the following advantages:

1) From the recursive relationship between queues and stacks, this paper creates two general loop invariant templates. The same binary tree solving sequence relation has a common feature in the derivation process: a general loop invariant template can be constructed. Our templates eliminate the time-consuming derivation process and complexity of traditional loop invariants for bi-
nary tree problems while also improving the efficiency of algorithmic derivation for binary tree problems, as demonstrated by the above examples.

2) Using the C++ generation system, we can automatically generate C++ executable programs from Apla programs to improve the completeness of the formal derivation of the algorithm program. As a result, we have completed the refinement process from the abstract specification to the executable program.

The proposed strategy improves the efficiency and completeness of deriving algorithm programs with commonality, suggests exploring the loop invariants of the non-recursive algorithm for recursive problems, and provides guidance for the formal derivation and proof of algorithm programs in the nonlinear data structure.

References


