Two New Strategies for Developing Loop Invariants and Their Applications

Xue Jinyun (薛锦云)

Computer Science Department, Jiangxi Normal University, Nanchang 330027

Received July 9, 1991; revised February 10, 1992.

Abstract

The loop invariants take a very important role in the design, proof and derivation of the algorithmic program. We point out the limitations of the traditional standard strategy for developing loop invariants, and propose two new strategies for proving the existing algorithmic program and developing new ones. The strategies use recurrence as vehicle and integrate some effective methods of designing algorithms, e.g. Dynamic Programming, Greedy and Divide-Conquer, into the recurrence relation of problem solving sequence. This lets us get straightforward an approach for solving a variety of complicated problems, and makes the standard proof and formal derivation of their algorithmic programs possible. We show the method and advantages of applying the strategies with several typical nontrivial examples.

Key words: Loop invariant, standard proof and formal derivation of program, recurrence relation, algorithm design.

1. Introduction

The algorithmic program is an algorithm described with an implemented or unimplemented programming language. The standard proof and formal derivation of an algorithmic program were the central research project of programming methodology in the past. The researchers working in the design and analysis of algorithms have paid special attention to the work in recent years [7]. They noticed that the correctness of an algorithm can be guaranteed only by using the standard program proving method. The loop invariant is a key for understanding the principles of an algorithmic program and implementing its standard proof, formal derivation and automatic development[13].

But existing techniques of developing loop invariant are only suitable for some simple problems. There are a lot of complicated algorithmic programs which cannot get satisfying loop invariant using these techniques. This leads to that many computer scientists doubt the possibility of deriving or proving algorithmic programs using loop invariant[8,10]. Some scholars even think that choosing a correct loop invariant needs more intelligence than designing an algorithm[7]. In a word, people treat loop invariant as an esoteric topic and this blocks its advance and widespread application.

In this paper, we try to reveal more properties of loop invariant and analyse the relationship between the algorithmic program and its loop invariant. Two new strategies will be presented and some comparisons with traditional strategies are made. Several typical nontrivial examples are given to specify the method using those strategies.

Remark about symbols and programming language. We use Dijkstra's Guard Command Language to describe programs. The variables in the programs are integer type without special declaration. $b[i:j]$ denotes all elements $b[k]$ in array $b.i\ k\ j$; if $i>j$, $b[i:j]$ denotes empty segment. We use a unified format $Q(i:r(i):f(i))$ given by Dijkstra to denote quantifiers including extended one, where Q can be A (universal quantifier), E (existing quantifier), MIN (minimal value of elements of a set), S (sum of all elements of a set), etc., and i is a

Research supported by the National Natural Science Foundation of China.

bounded variable, $r(i)$ the variant range of the value of i and $f(i)$ the function. The meanings of other symbols are: $\Longrightarrow$ (implication), $<=$ (less than or equal to), AND (conjunction), OR (disjunction), NOT (negation).

2. The New Strategies for Developing Loop Invariants

At present, the definition of loop invariant in widespread use is an assertion which is invariably true before and after each iteration, is called loop invariant.

Let DO denote loop statement

$$\text{do } B \rightarrow S \text{ od}$$

B is any predicate and S statement sequence. The predicates Q and R are pre- and post-condition of DO, and I is the loop invariant. The existing definition and developing techniques of loop invariant have the following defects:

1. The existing definition of loop invariant does not characterize the loop invariant. For example, Hoare Reasoning Rule and Loop Invariance Theorem[1,3] still hold after replacing the invariant in them by any tautology;

2. The loop invariants of many complicated problems cannot be developed by just weakening post-condition, the central idea of the Dijkstra-Gries standard strategies; even for some simple problems, there is no standard or rule to determine which strategy or weakening way should be selected for a given post-condition;

3. The condition for developing loop invariant using program function[9] is that we have program function and its program. But in general, determining the program function is quite difficult.

Therefore, we must study more properties of loop invariant and search for new strategies of developing loop invariant which are of universal significance and easy to use.

The universe is full of contradictions of variability and invariability. There are invariable pattern or properties in variable things. We call the pattern or properties invariant. We define variables of a loop program whose values are changed during execution of the loop as loop variables. Loop invariant controls the change of those loop variables and reflects their variation law. Therefore, we can give loop invariant a more exact definition.

Definition 2.1. *Given loop DO and its set A of all loop variables. An assertion which reflects variation law of each element of A, and is invariably true before and after each iteration is called loop invariant of loop DO.*

Obviously, the loop invariant defined by Definition 2.1 satisfies Hoare's Reasoning Rule of Loop Program, Loop Invariance Theorem and Verification Condition of Loop Program Correctness of [1, 3].

Let I denote the loop invariant of DO. I, by Definition 2.1, is the function of A, $I(A)$. The set A consists of 3 subsets: the set X of loop control variables, the set Y of loop variables in post-condition R of DO and the set Z, $Z=A-X-Y$. Thus the pre-condition of DO can be denoted by $Q(A)$, the post-condition by $R(Y)$.

According to the Verification Condition of Loop Program Correctness, $I(A)$ must be true before and after each iteration. We have $Q(A) \Rightarrow I(A)$ and $R(Y) \Rightarrow I(A)$. If Z is empty, we can get $I(A)$ by weakening $R(Y)$; otherwise it is difficult to get $I(A)$. So, we must search for new approach of developing loop invariant.

We know that loop DO transforms the state satisfying $Q(A)$ to the state satisfying $R(Y)$ AND $R(X)$ AND $R(Z)$ by changing A repeatedly, where $R(X)$ and $R(Z)$ are the states of X and Z on the termination. The loop invariant $I(A)$ just reflects the variation laws of A. Since loop of an algorithmic program embodies the central idea of the program, so does loop invariant. The idea of an algorithm includes background knowledge of the problem to be solved as well as its mathematical properties, the design techniquies of efficient algorithm such as dynamic programming, greedy, branch-and-bound, divide-conquer, etc. The pre- and post-condition of a loop cannot include all ideas of an algorithm. Therefore, the development of

loop invariant cannot just depend on weakening or generalizing the post-condition. This is the main difference of knowledge with the standard strategies of developing loop invariant [1,3,4,6].

We generalize the notion of recurrence relation of a sequence of numbers (difference equation) to problem solving sequence.

Suppose that the solution of problem P is given by the solution sequence: $P_1, P_2, \cdots, P_n$, where each P_i, $1 \leqslant i < n$, is the subsolution of a subproblem of P in its right, P_n is the solution of P. The equation that couples P_i with one or several P_j, $1 \leqslant j < i$, is called Recurrence Relation of problem solving sequence, or simply the recurrence relation, denoted by $P_i = F(\overline{P_j})$, where $\overline{P_j}$ is a sequence of several subsolutions. In general, the algorithm designed by using the recurrence relation is more efficient because it uses the result produced by its subproblem. Therefore one of effective approaches for designing algorithms is to search for the recurrence relation. In fact, some effective methods of designing algorithms, such as greedy, dynamic programming, divide-conquer, etc., may be unified to find recurrence relation.

Based on the above argument, we proposed two new strategies of developing loop invariant

Strategy 2.1. [for the existing algorithmic program]

Based on the Verification Conditions of Loop Program Correctness, investigate the pre-condition $Q(A)$ of the loop and assertion $R(Y)$ AND $R(X)$ AND $R(Z)$ AND NOT B on the termination, analyse background knowledge, mathematical properties of the problem to be solved by the program and the properties of the program itself, describe the variation laws of all loop variables by induction reasoning. The laws are needed loop invariants.

Strategy 2.2. [for not developed algorithmic program]

Investigate the pre- and post-condition as well as the mathematical properties of the problem, use the design techniques of efficient algorithm to determine general strategy of solving the problem (in most cases, to determine the recurrence relation of problem solving sequence) and all needed variables, describe the variation laws of each variable. The laws are needed loop invariants; if the number of the subsolutions in the recurrence relation is more than 1, one sequence variable which will be used as a stack or a set variable must be added and the content of the sequence is defined recursively.

We will, using typical examples, demonstrate the method of applying two strategies to develop loop invariant in the next two sections. Deriving algorithmic program is not the task of this paper. The readers who are interested in the topic may refer to the related chapters in [1] and [3].

3. The Method of Applying Strategy 2.1

We investigate a complex algorithmic program.

Example 3.1. Generating a random cyclic permutation.

In [5], we prove the algorithm of generating random permutations given by Sattolo using standard program-proving methods. In this paper, we just pay main attention to developing the loop invariant used in the proof. The following is Sattolo's algorithmic program and its pre- and post-condition, where procedure random (r) assigns to r a random number uniformly distributed and satisfying $0 \leqslant r < 1$ and function floor.x yields the integer part of x, for $x \geqslant 0$.

$\{Q: A(i: 0 \leqslant i < n: b[i] := i)\}$
$i := n-1;$
do $i <> 0 \longrightarrow$ random (r) (1)
$s :=$ floor. $(i * r)$; (2)
$b[i], b[s] := b[s], b[i]$; (3)
$i := i-1$
od
$\{R1: b[0: n-1]$ contains a cyclic permutation of integer 0, 1, 2, $\cdots, n-1$
AND
$R2:$ each cyclic permutation stored in b generated by the program has equal probability.$\}$

For simplicity, we just develop the loop invariant corresponding to $R1$. After investigating the pre- and post-condition, we have:

1. At the beginning of the loop, $b[0:n-1]$ contains a permutation of integer 0, 1, 2,⋯, $n-1$. Only the positions of elements of b are changed by the statement (3) of the program. Therefore, b contains a permutation of integer $0,1,\cdots,n-1$ during the execution of the program throughout. Thus we get the first property of array b:

$I0$: $b[0:n-1]$ contains a permutation of $0, 1, 2,\cdots, n-1$.

2. At the beginning of the loop, there are n cycles in $b[0:n-1]$. After $n-1$ iterations, the loop is terminated and R1 holds, i.e. there is one cycle in $b[0:n-1]$. So we guess that the number of cycles in $b[0:n-1]$ reduces one in each iteration. This means that array b may have another property:

$I1$: $b[0:n-1]$ contains $i+1$ cycles, $0\leqslant i<n$

What is the condition the guess holds? i.e., what is the condition under which the number of cycles reduces one by each execution of statement (3)? After studying the mathematical property of cyclic permutation [1,2,5], we discovered and proved the following lemma.

Lemma 3.1. *Exchanging two elements from different cycles of a permutation merges the two cycles into one cycle.*

After further study on statements (1), (2) and (3), we find that array b has another property:

$I2$: the elements of $b[0:i]$ are in $i+1$ different cycles of the permutation.

This means $b[i]$ and $b[s]$, $0\leqslant s<i$, are in two different cycles of permutation and satisfy the condition under which Lemma 3.1 holds. The guess $I1$ is also confirmed. It can be verified that $I0$ AND $I1$ AND $I2$ is valid loop invariant.

4. The Method of Applying Strategy 2.2

It is quite effective for developing loop invariant of not developed algorithmic program which does not pursue efficiency using the standard strategies in [1, 3]. For the algorithmic programs which pursue efficiency, the standard strategy in [4] is used. In order to compare our strategy with the traditional standard strategies, we develop the loop invariant of the Minimum-Sum Problem in [4] again.

Example 4.1. Given an array $b[0:n-1]$, where $n\geqslant 1$. A minimum-sum section of b is a sequence of adjacent elements whose sum is a minimum (the empty section is in b and has sum zero). Desired is a program that stores in a variable s the sum of a minimum-sum section of b. It is to establish:

$$R: s=\text{minsum}(b[0:n-1])$$

where $\text{minsum}(b[0:n-1])=\text{MIN}(i,j: 0\leqslant i\leqslant j<n: \text{sum}(i,j))$

$$\text{sum}(i,j)=S(k: i\leqslant k\leqslant j: b[k])$$

An obvious method to establish R is listing the sum of all sections, then choosing the minimum. But the computation complexity is $O(n^3)$. In order to get efficient algorithm, we search for the recurrence relation to compute the minsum $(b[0:n-1])$. Suppose minsum $(b[0:i-1])$, $0\leqslant i<n$, has been computed, we pay main attention to how to compute minsum $(b[0:i])$ by minsum $(b[0:i-1])$. Following lemma is an answer:

Lemma 4.1. *Given b as above, suppose $m(i)$ is minimum sum section of b ending in $b[i]$, i.e.:*

$$m(i)=MIN(j: 0\leqslant j\leqslant i: \text{sum}(j,i))$$

then

$$minsum(b[0:i])=min(minsum(b[0:i-1]), m(i)).$$

Proof. Let $i=0$. minsum $(b[0])$ is zero or $b[0]$ corresponding to $b[0]\geqslant 0$ or $b[0]<0$. The lemma holds.

Now, let $i>0$. From the definition of function minsum and $m(i)$, we have

$$\text{minsum}(b[0:i-1])\leqslant m(i-1)$$

Therefore, the possible section sum of $b[0:i]$ which is less than minsum $(b[0:i-1])$ is

$m(i-1)+b[i]$ and $b[i]$. But according to the definition of $m(i)$, it is obvious that

$$m(i)=\min(m(i-1)+b[i], b[i]).$$

Therefore

$$\text{minsum}(b[0:i])=\min(\text{minsum}(b[0:i-1]), m(i)).$$ End of the proof.

From Lemma 4.1, we notice that optimal solution minsum($b[0:n-1]$) comprises only optimal subsolution. That is a Dynamic Programming.

Let minsum ($b[0:i-1]$) be stored in s and $m(i-1)$ in variable c. We have loop invariant:

$$I: s=\text{minsum}(b[0:i-1]) \text{ AND } c=m(i-1) \text{ AND } 0\leqslant i\leqslant n$$

Based on R, I and bound function $n-i$, we can derive an $O(n)$ algorithmic program:

```
i, s, c, := 0, 0, 0;
do i< > n → c := min(c+b[i], b[i]);
            s := min(s, c);
            i := i+1
od
```

The loop invariants of minimum-sum problem appearing in this paper and [4] are almost the same, but the developing methods are quite different. We were guided by our principle: An effective approach of designing efficient algorithm is to search for the recurrence relation and found the relationship among minsum ($b[0:i]$), minsum ($b[0:i-1]$) and $m(i)$, i.e. Lemma 4.1. Then we determined all needed variables as well as their properties for solving the problem. This is loop invariant I. The whole process is quite straightforward. But in [4], the author found the (first approximation to the) invariant by weakening the post-conditions R(replacing a constant by a variable), developed the program on it, then investigated the efficiency of the algorithmic program and introduced fresh variables, thus modified the invariant, but gave no principle or rule showing us how to increase the efficiency of the algorithmic program development. We believe that by consciously employing our approach used in this example to Plateau, Maximum-sum, Upsequence and Maximum-product problems, their own loop invariant and algorithmic programs can be developed more straightforward.

Example 4.2. The single source shortest paths problem.

Given a directed connected graph $G=(V, E, C)$, in which V is the set of vertices, E the set of edges, C the set of the nonnegative "length" of each edge. s in V is specified as the source. Our problem is to determine the distance of the shortest path from the source s to given vertex t in V, denoted by mindis (s, t), where the distance of a path is just the sum of the length of the edges on the path. We search for the recurrence relation of computing mindis (s, t), then determine its invariant.

Let $(s:t)$ denote one of the paths from source s to vertex t. Suppose the distance of each shortest path from s to every internal vertex in $(s:t)$ has been computed, (x, t) is an edge of the path $(s:t)$, mindis (s, x) is already known. Let the set M contain all vertices whose shortest distance from the source s has been computed. There is the shortest path from s to t (t in $V-M$) that passes only through vertices in M. We call such a path special path whose length is denoted by function spdis (s, t). Obviously, the value of spdis (s, t) must be revised after adding a vertex to M. Let spdis$_i$ (s, t) denote the value of spdis (s, t) after i-th revision. Initially, M contains only source s. If $s=t$, mindis (s, s)= spdis $(s, s)=0$, spdis(s, y) = the length of edge (s, y), y in $V-M$. Now let $s< >t$, we have the following recurrence relation for computing mindis (s, t):

Lemma 4.2. *Suppose $V< >M$, x is the last and i-th vertex added to M. $spdis_{i-1}(s, y)$ denotes the distance of special path from s to any vertex y, y in $V-(M-\{x\})$; k, a vertex in $V-M$, is adjacent to x. We have:*

(i) *For each y in $V-M$:*

$$spdis_i(s, y)=min(spdis_{i-1}(s, y), mindis(s, x)+C(x, y));$$

(ii) $mindis(s, k)=spdis_i(s, k)$, *where*

$$spdis_i(s, k)=MIN(y: y \text{ in } V-M: spdis_i(s, y)).$$

Proof. (i) The special path from s to any y, y in $V-M$, has only 3 cases:

a. not pass through x, its length is $\text{spdis}_{i-1}(s, y)$;

b. pass through x directly to y, its length is $\text{mindis}(s, x)+c(x, y)$;

c. pass through x, then to some vertex j of $V-(M-\{x\})$, then to y.

We prove that the length of the path in case c must be no shorter than $\text{spdis}_{i-1}(s, y)$. Since j is added in M before x, the shortest one of all paths from s to j passes through $M-\{x\}$ alone. Therefore, the path to j through x is no shorter than the path directly to j through $M-\{x\}$. As a result, the path from s to x, j and y is no less than $\text{spdis}_{i-1}(s, y)$. Therefore, $\text{spdis}_i(s, y)$ can only be the shorter one of the paths in cases a and b.

(ii) If there is a nonspecial path that first leaves M, and goes directly to j in $V-M$, then (perhaps) wanders into and out of M several times, finally reaches k and its length is less than $\text{spdis}_i(s, k)$, therefore, $\text{spdis}_i(s, j)<$ the length of the nonspecial path $<\text{spdis}_i(s, k)$. This causes a contradiction with $\text{spdis}_i(s, k)=\text{MIN}(y: y \text{ in } V-M: \text{spdis}_i(s, y))$.

Based on Lemma 4.2, we immediately have:

Lemma 4.3. *t is the nearest vertex to s in $V-M$.*

Lemma 4.3 tells us that above algorithm uses "Greedy" technique which is used in Dijkstra's Algorithm of Shortest Path Problem[15].

Obviously, from Lemma 4.2, if $k=t$, $\text{mindis}(s, k)$ is the shortest distance from s to t and $A(v: v \text{ in } M: \text{mindis}(s, v)=\text{spdis}_i(s, v))$ holds. Thus, if $V=M$, we get the shortest distance from s to every other vertex in V. Let the value of each $\text{spdis}(s, v)$ be stored in an array d, the followings are the post-condition R, the loop invariant I and bound function T:

$R: A(x: x \text{ in } V: d(x)=\text{mindis}(s, x))$

$I: A(x: x \text{ in } M: d(x)=\text{mindis}(s, x) \text{ AND } d(t) \geqslant d(x)) \text{ AND}$

$A(x: x \text{ in } V-M: d(x)=\text{spdis}(s, x) \text{ AND } d(t) \leqslant d(x)) \text{ AND } (t \text{ IN } V)$

$T: \#V-\#M$

Based on R, I and T, the following program can be derived:

(4.2)
```
M, t, d(s) := {s}, S, 0;
FOREACH i in V-M do d(i) := C(s, i) od;
do V <> M →
        FOREACH y in V-M do d(y) := min(d(y), d(t)+C(t, y)) od;
        choosemin(t, V-M, d(t));
        M := M ∪ {t}
od
```

where procedure choosemin has function to choose t in $V-M$ such that $d(t)$ is a minimum; function $C(x, y)$ denotes the length of the edge (x, y). If there is no edge between x and y, we define $C(x, y)$ to be infinity.

In general, if there are two more subsolutions in a recurrence relation, then it is convenient to describe algorithmic program using recursive procedure. We call this kind of problems recursive problem. But in order to pursue efficiency, we often use loop program instead of recursive procedure. It was universally thought that solving the recursive problem using loop is quite difficult. The following example shows the method to develop the loop invariant of the problem using Strategy 2.2.

Example 4.3. Preorder traversal of a binary tree.

Let T be a finite binary tree. If it is empty, it is denoted by $T=\%$; otherwise, $T=(T.l, T.i, T.r)$, where $T.i$ is the flag of root node, $T.l$ and $T.r$ are left and right subtrees of T respectively.

Let Pre. T denote the preorder traversal of T. It produces a sequence of all node flags of T. Let the sequence be stored in a sequence variable X. We have the post-condition of the algorithmic problem:

$R: X=\text{Pre. } T$

Let $\wedge$ denote the catenation of sequences and [ ] denote empty sequence. The recurrence relation of computing Pre. T is:

Pre. $T=[\,]$ if $T=\%$

Pre. $T = T.i \wedge$ pre. $(T.l) \wedge$ pre. $(T.r)$ if $T <> \%$

Obviously, this embodies the thought of Divide-conquer and we have:

$$\begin{aligned}\text{Pre. } T &= T.i \wedge \text{Pre. } (T.l) \wedge \text{pre. } (T.r)\\ &= T.i \wedge T.l.i \wedge \text{Pre. } (T.l.l) \wedge \text{Pre. } (T.l.r) \wedge \text{Pre. } (T.r)\\ &= T.i \wedge T.l.i \wedge T.l.l.i \wedge \text{Pre. } (T.l.l.l) \wedge \text{Pre. } (T.l.l.r) \wedge \text{Pre. } (T.l.r) \wedge \text{Pre. } (T.r)\end{aligned}$$

Thus, we get the general strategy of solving Preorder problem: introduce 3 variables X, S, q, where X is used to store the node flag sequence, on the termination $X = \text{Pre. } T$; S is a sequence variable which is used as a stack, $S.0$ denotes the first element of S, $S[1...]$ denotes the remained part of S after deleting $S.0$, the content of S is given by function F. The definition of F is:

$F.[\,] = [\,]$;

$F.(P \wedge S) = \text{Pre. } P \wedge F.S$

On the termination, $s = [\,]$; q is used to store the subtree which is about to be visited. Thus, X, S and q satisfy the following equation:

$$I: X \wedge \text{Pre. } q \wedge F.S = \text{Pre. } T.$$

Let seg (t) denote sequence type whose element type is t. We can derive the following program with loop invariant I :

(4.3) Var X : seq (integer); S: seq (tree); q: tree

$X, q, S := [\,], T, [\,]$;

do $q = \%$ AND $S <> [\,] \longrightarrow q, S := S.0, S[1...]$;

$q <> \% \longrightarrow X, q, S := X \wedge q.i, q.l, q.r. \wedge S$

od

It is easy to verifty that I is really a loop invariant. On the termination of the loop, $q = \%$ AND $S = [\,]$ AND $X \wedge \text{Pre. } q \wedge F.S = \text{Pre. } T \Rightarrow X = \text{Pre. } T$.

It is difficult to develop the loop invariant of recursive problem using the standard strategy and the form of the loop invariant is quite complicated. Here, due to Strategy 2.2, especially its recursive technique, we get straightforward the loop invariant with a simple form. We believe that this technique has universal significance for developing and expressing the loop invariant of the recursive problem.

The final remark. The algorithms in above examples appeared in many literatures, but we did not mention any design techniques used in those algorithms. The only technique used in our development is "searching for the recurrence relation".

5: Conclusion and Discussion

In summary, we have:

1. The loop invariant embodies design thought of algorithmic program and is also the bridge of understanding, proving and designing algorithmic program.

2. An algorithmic program or algorithm problem certainly has loop invariant. Currently, developing it is a creative activity and cannot be done by using several simple strategies or rules mechanically. This is the same as to develop new notion of mathematics and new laws of physics and chemistry. Our Strategies 2.1 and 2.2 put special emphasis on the algorithm design ideas and give principles to show how to get a creative result. This is the main difference between our strategies and traditional standard strategies.

3. Three examples in Section 4 illustrate that searching for recurrence relation is a powerful technique for designing efficient algorithmic program. For a given problem, we need not make choice among the existing algorithm design techniques, e.g. Dynamic Programming, Greedy and Divide-Conquer etc. and just pay main attention to searching for the recurrence relation. It is noticed that some problems can be solved by traditional technique of "Replacing a constant by a variable", but it is just a special case of our Recurrence Relation technique. We believe that the recurrence relation technique may become a systematic approach for designing algorithmic program and developing its loop invariant.

Acknowledgements

Thanks go to David Gries who encouraged and guided my interests in this field.

References

[1] Dijkstra, E. W., A Discipline of Programming. Prentice Hall, Englewood Cliffs, 1976.
[2] Dijkstra, E. W. and van Gasteren, A. J. M., A simple fixpoint argument without the restriction to continuity. *Acta Informatica*, 1986, 23, 1 — 7.
[3] Gries, D., The Science of Programming. Springer Verlag, New York, 1981.
[4] Gries, D., A note on a standard strategy for developing loop invariants and loops. *Science of Computer Programming*, 1982, 2, 207 — 214.
[5] Gries, D. and Xue, Jinyun, Generating a Random Cyclic Permutation. *BIT*, 1988, 28, 569 — 572.
[6] Gries, D. and Wadkins, J., An Introduction to Proofs of Program Correctness for Teacher of College-level Introduction Programming Courses. TR90—1102, Dept. of CS, Cornell Univ., 1990.
[7] Harel, D., Algorithmics: The Spirit of Computing. Addison-Wesley, 108 — 109, 1987.
[8] Denning, P., A debate on teaching computing science. *CACM*., 1989, 32, 1397 — 1414.
[9] Linger, R. C., Mills, H. D. and Witt, B. I., Structured Programming: Theory and Practice. Reading Mass, Addison-Wesley, 1979.
[10] Manna, Z. and Waldinger, R., Synthesizer: Dream⇒ Program. *IEEE Transaction on Software Engineering*, 1979, 5, 294 — 328.
[11] Remmers, J., A technique for developing loop invariants. *Information Processing Letters*, 1984, 18, 137 — 139.
[12] Xue, Jinyun and Gries, D., Developing a linear algorithm for cubing a cycle permutation. *Science of Computer Programming*, 1988, 11, 161 — 165.
[13] Li Zhaoren, A formal method of programming and the experimental system. *Chinese Journal of Computers*, 1985, 8(1), 8 — 18.
[14] Xue, Jinyun, On Loop Invariant and Its Developing Techniques. Proceedings of China 4th Conference On Software Engneering, 1991. (in Chinese).
[15] Aho, A., Hopcroft, J. and Ullman, J., Data Structures and Algorithms. Addison-Wesley Publishing Company, 1983.