Joint NCC & IIS Bull., Comp. Science, 13 (2000), 56–66

Verification of pointer programs using symbolic method for definite iterations*

V. A. Nepomniaschy

The symbolic method for verifying definite iterations over hierarchical data structures [15] is extended to allow a restricted change of the structures by the iteration body and exit from the iteration body under a condition. A transformation of definite iterations which use exit from the iteration bodies to the standard definite iterations is justified. Programs over linear lists are considered as a case of study. A technique for proving verification conditions based on both induction principles and notions related to the problem domain is developed. Examples which illustrate application of the symbolic method to pointer program verification are considered.

1. Introduction

The axiomatic and functional styles of program verification include the following stages: program annotation through construction of pre-, post-conditions and loop invariants or functions expressing the loop effect; deriving verification conditions with the help of proof rules and proving them [6, 9]. In both approaches loop annotation is still a difficult problem [11, 16]. Difficulties of pointer program verification have been noted for the axiomatic approach in [3]. Decidable logics have been proposed to describe special properties of pointer programs [2, 8]. This allows a verification technique to be developed for loopfree pointer programs [8] but does not simplify the loop annotation.

A natural method of attack on the verification problem is the use of definite iterations, for example, Pascal for-loops. Although the reduction of for-loops to while-loops is often used for verification, attempts to use the specific character of for-loops in the framework of the axiomatic approach should be noted [1, 4, 5, 7]. In the framework of the functional approach, a general form of a definite iteration as an iteration over all elements of a structure, such as list, set, file and tree, has been proposed in [17].

A symbolic method for verifying for-loops with the statement of assignment to array elements as the loop body has been proposed in [12, 13]. This method is based on using the symbols of invariants instead of the invariants in verification conditions and a special technique for proving the conditions. In [14] we extended the symbolic method to definite iterations over data structures without restrictions on the loop bodies. The symbolic method has been developed for definite iterations over hierarchical data structures in [15].

The purpose of this paper is to apply the symbolic method to pointer program verification. A definite iteration over hierarchical data structures which allows for a restricted change of the structures by the iteration body, as distinct from [15], is described in Section 2. A definite iteration which uses exit from the iteration body under a condition is defined in Section 3 where its reduction to the standard definite iteration over suitable hierarchical data structures is justified. Proof rules without invariants for generating verification conditions and induction principles for proving them are considered in Section 4. Definite iterations over linear lists are considered in Section 5 where notions for annotating these programs and proof rules for Pascal pointer statements are discussed. Verification of two programs which perform an in-situ reversal of a linear list and a search in a linear list with reordering is exemplified in Section 6. In conclusion, results and prospects of the symbolic verification method are discussed.

*This work is supported in part by RFBR grant 00-01-00909.

2. Definite iteration over hierarchical data structures

We introduce the following notation. Let $\{s_1, ..., s_n\}$ be a multiset consisting of elements $s_1, ..., s_n$, $U_1 - U_2$ be the difference of multisets U_1 and U_2, $U_1 \cup U_2$ be the union of multisets, and $|U|$ be the power of a finite multiset U. Let $[v_1, ..., v_m]$ denote a vector consisting of elements $v_i (1 \le i \le m)$.

Let us remind the notion of a data structure [17]. Let $memb(S)$ be a finite multiset of elements of a structure S, $empty(S)$ be a predicate "$memb(S)$ is empty", $choo(S)$ be a function which returns an element of $memb(S)$, $rest(S)$ be a function which returns a structure S' such that $memb(S') = memb(S) - \{choo(S)\}$. The functions $choo(S)$ and $rest(S)$ will be undefined if and only if $empty(S)$.

Let us remind a definition of useful functions related to the structure S in the case of $\neg empty(S)$ and $memb(S) = \{s_1, ..., s_n\}$ [14]. Let $vec(S)$ denote a vector $[s_1, ..., s_n]$ such that $s_i = choo(rest^{i-1}(S))$ $(i = 1, ..., n)$. Structures S_1 and S_2 are recognized as equal if and only if $vec(S_1) = vec(S_2)$. A function $head(S)$ returns a structure such that $vec(head(S)) = [s_1, ..., s_{n-1}]$ if $vec(S) = [s_1, ..., s_n]$ and $n \ge 2$. If $n = 1$, then $empty(head(S))$. Let $last(S)$ be a partial function such that $last(S) = s_n$ if $vec(S) = [s_1, ..., s_n]$. Let $str(s)$ denote a structure S which contains the only element s. The functions $vec(S), head(S)$ and $last(S)$ will be undefined in the case of $empty(S)$. A concatenation operation $con(S_1, S_2)$ is defined in [14] so that $con(choo(S), rest(S)) = con(head(S), last(S)) = S$ if $\neg empty(S)$.

We will use $T(S_1, ..., S_m)$ to denote a term constructed from data structures S_i $(i = 1, ..., m)$ with the help of the functions $choo, last, rest, head, str, con$. For a term T which represents a data structure, we denote the function $|memb(T)|$ by $lng(T)$. The function can be calculated by the following rules: $lng(S_i) = |memb(S_i)|$, $lng(con(T_1, T_2)) = lng(T_1) + lng(T_2)$, $lng(rest(T)) = lng(head(T)) = lng(T) - 1$, $lng(str(s)) = 1$.

Let a hierarchical data structure $S = STR(S_1, ..., S_m)$ be defined by the functions $choo(S)$ and $rest(S)$ constructed with the help of conditional $if-then-else$, superposition and Boolean operations from the following components:

- terms not containing $S_1, ..., S_m$;
- the predicate $empty(S_i)$ and the functions $choo(S_i), rest(S_i), last(S_i)$, $head(S_i)$ $(i = 1, ..., m)$;
- terms of the form $STR(T_1, ..., T_m)$ such that $\sum_{i=1}^{m} lng(T_i) < \sum_{i=1}^{m} lng(S_i)$;
- an undefined element ω.

Note that the undefined value ω of the functions $choo(S)$ and $rest(S)$ means $empty(S)$. This definition of hierarchical structures gives us more convenient application of the induction principle 1 from Section 4 to proving the properties of the structures.

Let us consider a definite iteration of the form

$$\textbf{for } x \textbf{ in } S \textbf{ do } v := body\,(v, x) \textbf{ end} \tag{1}$$

where S is a data structure which may be hierarchical, x is a variable called a loop parameter, v is a data vector of the loop body $(x \notin v)$. The result of this iteration is an initial value v_0 of the vector v if $empty(S)$. If $\neg empty(S)$ and $vec(S) = [s_1, ..., s_n]$, the loop body $v := body(v, x)$ iterates sequentially for x defined as $s_1, ..., s_n$, and does not change the structure $rest^i(S)$ when $x = s_i$ for all $i = 1, ..., n-1$. Therefore, $vec(S) = vec(S_0)$ where S_0 is an initial value of the structure S.

3. Definite iteration including exit statement

Definite iteration (1) is extended so that exit is allowed from the iteration body under a condition. Let us consider the statement

$$\textbf{for } x \textbf{ in } S \textbf{ do } v := body_1(v, x); \textbf{ if } cond(v, x) \textbf{ then } EXIT; v := body_2(v, x) \textbf{ end} \tag{2}$$

where S is a data structure which may be hierarchical, x is a loop parameter, v is a data vector ($x \notin v$), and EXIT is the statement of termination of the loop. The result of iteration (2) is an initial value v_0 of the vector v if $empty(S)$. If $\neg empty(S)$ and $vec(S) = [s_1, ..., s_n]$, the loop body iterates sequentially for x defined as $s_1, ..., s_n$ while the condition $cond(body_1(v, x), x)$ is false. When the condition is first true for $x = s_i$, iteration (2) terminates by performing the statement $v := body_1(v, s_i)$. The loop body does not change the structure $rest^i(S)$ when $x = s_i$ and the condition $cond(body_1(v, x), x)$ is false $(i = 1, ..., n-1)$.

Our purpose is to eliminate the output statement from iteration (2) by its transformation to an equivalent program which includes iteration (1). Such a transformation is realized in two stages: a change of the condition $cond(v, x)$ by the condition $cond(v_0, x)$ in iteration (2); elimination of the exit statement with the help of a hierarchical structure which depends on v_0, the condition $cond(v_0, x)$ and the structure S.

At first, we will define restrictions to the iteration (2) which allows us to eliminate the exit statement. For a structure S such that $\neg empty(S)$ and $vec(S) = [s_1, ..., s_n]$, we use S' to denote a structure such that $\neg empty(S')$ and $vec(S') = [(s_1, 1), ..., (s_n, n)]$. A function $body(v, x)$ preserves a condition $cond(v, x)$ with respect to a structure S if in the case of $\neg empty(S)$, $cond(body(v, x'), x) = cond(v, x)$ for all v, x, x', for which there exist integers i, j such that $j \leq i$ and $(x, i), (x', j) \in memb(S')$. A function $body(v, x)$ weakly preserves a condition $cond(v, x)$ with respect to a structure S if in the case of $\neg empty(S), cond(body(v, x'), x) = cond(v, x)$ for all v, x, x', for which there exist integers i, j such that $j < i$ and $(x, i), (x', j) \in memb(S')$. It should be noted that if all elements of a structure S are different, then in these definitions the structure S can be used instead of the structure S'. In this case $x, x' \in memb(S)$ and relations $j \leq i, j < i$ are replaced by relations $x' \leq x, x' < x$, respectively, where $x' \leq x$ denotes that x' does not succeed x in $vec(S)$, and $x' < x$ denotes that x' precedes x in $vec(S)$.

Lemma 1. If the function $body_1(v, x)$ preserves and the function $body_2(v, x)$ weakly preserves the condition $cond(v, x)$ with respect to the structure S, then iteration (2) with an initial value v_0 of the vector v is equivalent to the iteration

$$\textbf{for } x \textbf{ in } S \textbf{ do} v := body_1\ (v, x);\ \textbf{if } cond(v_0, x) \textbf{ then } EXIT;\ v := body_2(v, x) \textbf{ end}. \tag{3}$$

Proof. Lemma 1 is evident if $empty(S)$. Let us suppose $\neg empty(S)$ and $vec(S) = [s_1, ..., s_n]$. Let $v_{2i-1} = body_1(v_{2i-2}, s_i)$, $v_{2i} = body_2(v_{2i-1}, s_i)$ $(i = 1, ..., n)$. We use m to denote an integer such that $1 \leq m \leq n$ and the body of iteration (2) is performed for x defined as $s_1, ..., s_m$. Two cases are possible. In the first case $\neg cond(v_{2i-1}, s_i)$ for all $i = 1, ..., n$ and $m = n$. In the second case $\neg cond(v_{2i-1}, s_i)$ for all $i = 1, ..., m-1$ and $cond(v_{2m-1}, s_m)$. Lemma 1 immediately follows from the condition $\forall j\,(1 \leq j \leq m \to cond(v_{2j-1}, s_j) = cond(v_0, s_j))$. This condition results from the following more general condition for $i = j$:

$$\forall j\,(1 \leq j \leq m \to \forall i\,(1 \leq i \leq j \to cond(v_{2i-1}, s_j) = cond(v_0, s_j))). \tag{4}$$

To prove condition (4), we use induction on $i = 1, ..., j$ for a fixed integer $j\,(1 \leq j \leq m)$. The function $body_1$ preserves the condition $cond$ with respect to the structure S, and $1 \leq j$ holds for $(s_1, 1), (s_j, j) \in S'$. It follows from this that $cond(v_1, s_j) = cond(body_1(v_0, s_1), s_j) = cond(v_0, s_j)$. Therefore, condition (4) holds for $i = 1$. Let us consider the case $i > 1$. From the inductive hypothesis, the premise of Lemma 1 and $i \leq j$, it follows that $cond(v_{2i-1}, s_j) = cond(body_1(v_{2i-2}, s_i), s_j) = cond(v_{2i-2}, s_j) = cond(body_2(v_{2i-3}, s_{i-1}), s_j) = cond(v_{2i-3}, s_j) = cond(v_0, s_j)$. Therefore, condition (4) holds.

Let us define a hierarchical structure $ET(S)$ from the structure S, the condition $cond$ and the initial value v_0 of the vector v as

$(choo(ET(S)), rest(ET(S))) =$

$$\textbf{if } empty(S) \vee cond(v_0, choo(S)) \textbf{ then } (\omega,\omega) \textbf{ else } (choo(S), ET(rest(S))).$$

The following lemma describes elementary properties of the structure $ET(S)$.

Lemma 2.
2.1. If $\neg empty(ET(S))$, then the vector $vec(ET(S))$ is an initial segment of the vector $vec(S)$.
2.2. The condition $\neg cond(v_0, s)$ holds for all $s \in memb(ET(S))$.
2.3. If $ET(S) \neq S$, $vec(S) = [s_1, ..., s_n]$ and $k = |memb(ET(S))| + 1$, then the condition $cond(v_0, s_k)$ holds.
2.4. If $ET(S) \neq S$, then $ET(S) = ET(head(S))$.

Proof. We will use induction on $n = |memb(S)|$. If $n = 0$, then $empty(S)$, $empty(ET(S))$ and Lemma 2 is evident. Let us suppose that $n > 0$ and Lemma 2 holds for $|memb(S)| < n$. In the case of $cond(v_0, s_1)$, it is evident that $empty(ET(S)), k = 1$, $empty(ET(head(S)))$, and, therefore, Lemma 2 holds. Let us consider the case $\neg cond(v_0, s_1)$. Then $ET(S) = con(s_1, ET(rest(S)))$, and assertions 2.1, 2.2 of the lemma follow from the inductive hypothesis. If $ET(S) \neq S$, then $n > 1, ET(rest(S)) \neq rest(S)$, and it follows from the inductive hypothesis that $cond(v_0, s_k)$ for $vec(rest(S)) = [s_2, ..., s_n]$ and $k = |memb(ET(rest(S)))| + 2 = |memb(ET(S))| + 1$. Therefore, assertion 2.3 of the lemma holds. In the case of $ET(S) \neq S$ it follows from the inductive hypothesis that $ET(rest(S)) = ET(head(rest(S)))$, and, therefore, $ET(S) = con(s_1, ET(head(rest(S))))$. To prove assertion 2.4 of the lemma, it remains to notice that $ET(head(S)) = con(s_1, ET(rest(head(S))))$ and $head(rest(S)) = rest(head(S))$.

Lemma 3. Iteration (3) with the initial value v_0 of the vector v is equivalent to the program

$$\textbf{for } x \textbf{ in } ET(S) \textbf{ do } v := body_1(v, x); v := body_2(v, x) \textbf{ end}; \textbf{if } ET(S) \neq S \textbf{ then } v := body_1(v, s_k) \quad (5)$$

where $k = |memb(ET(S))| + 1$ and $vec(S) = [s_1, ..., s_n]$.

Proof. We will use induction on $n = |memb(S)|$. If $n = 0$, then $empty(S)$, $ET(S) = S$ and Lemma 3 is evident. Let us suppose that $n > 0$ and Lemma 3 holds for $|memb(S)| < n$. In the case of $\neg cond(v_0, s_i)$ for all $i = 1, ..., n$, Lemma 3 follows from Lemma 2.3 and $ET(S) = S$. Otherwise, let us fix the least integer $i\,(1 \le i \le n)$ such that $cond(v_0, s_i)$. From Lemma 2 it follows that $ET(S) \neq S$ and $ET(S) = ET(head(S))$. Two cases are possible:

1. $1 \le i \le n - 1$. Then iteration (3) is equivalent to the iteration

for x **in** $head(S)$ **do** $v := body_1(v, x)$; **if** $cond(v_0, x)$ **then** $EXIT$; $v := body_2(v, x)$ **end**

which, by the inductive hypothesis, is equivalent to the program

for x **in** $ET(head(S))$ **do** $v := body_1(v, x)$; $v := body_2(v, x)$ **end**;
if $ET(head(S)) \neq head(S)$ **then** $v := body_1(v, s_k)$

where $k = |memb(ET(head(S)))| + 1$. It remains to notice that $ET(S) \neq head(S)$ follows from Lemma 2.2.

2. $i = n$. Then iteration (3) is equivalent to the program

for x **in** $head(S)$ **do** $v := body_1(v, x)$; $v := body_2(v, x)$ **end**; $v := body_1(v, s_n)$.

It remains to notice that $n = |memb(head(S))| + 1$ and $ET(S) = head(S)$ follows from Lemma 2.

The following theorem immediately follows from Lemmas 1, 3.

Theorem 1. If the function $body_1(v, x)$ preserves and the function $body_2(v, x)$ weakly preserves the condition $cond(v, x)$ with respect to the structure S, then iteration (2) with an initial value v_0 of the vector v is equivalent to program (5).

Corollary 1. If the function $body(v, x)$ weakly preserves the condition $cond(v, x)$ with respect to the structure S, then the iteration

for x **in** S **do if** $cond(v, x)$ **then** $EXIT$; $v := body(v, x)$ **end**

with an initial value v_0 of the vector v is equivalent to the iteration

for x **in** $ET(S)$ **do** $v := body(v, x)$ **end**.

Notice that Corollary 1 extends the theorem Th 7 [15].

Corollary 2. If the function $body(v, x)$ preserves the condition $cond(v, x)$ with respect to the structure S, then the iteration

for x **in** S **do** $v := body(v, x)$; **if** $cond(v, x)$ **then** $EXIT$ **end**

with an initial value v_0 of the vector v is equivalent to the program

for x **in** $ET(S)$ **do** $v := body(v, x)$ **end**; **if** $ET(S) \neq S$ **then** $v := body(v, s_k)$

where $k = |memb(ET(S))| + 1$ and $vec(S) = [s_1, ..., s_n]$.

4. Generating and proving verification conditions

Let $R(y \leftarrow exp)$ be a result of substitution of an expression exp for all occurrences of a variable y into a formula R. Let $R(vec \leftarrow vexp)$ denote the result of a synchronous substitution of the components of an expression vector $vexp$ for all occurrences of corresponding components of a vector vec into a formula R. The proof rule $rl1$ [10] for definite iteration (1) uses the replacement operation $rep(v, S, body)$ where $body$ is the function associated with the right side of the iteration $body$. The replacement operation presents the effect of iteration (1) [14]. Theorem 6 [14] claims that iteration (1) is equivalent to the multiple assignment $v := rep(v, S, body)$. The rule $rl1$ replaces the post-condition Q by $Q(v \leftarrow rep(v, S, body))$. To prove the verification conditions including the replacement operation $rep(v, S, body)$ with the hierarchical structure S, we present two induction principles.

Let $prop(STR(S_1, ..., S_m))$ denote a property expressed by a first-order logic formula only with free variables $S_1, ..., S_m$. The formula is constructed from functional symbols, variables and constants by means of Boolean operations and first-order quantifiers. The functional symbols include $memb, empty, vec, choo, rest, last, head, str, con$.

The following principle is easily proved by induction on $k = \sum_{i=1}^{m} lng(S_i)$.

Induction principle 1. The property $prop(STR(S_1, ..., S_m))$ holds for all structures $S_1, ..., S_m$ if there exists an integer $c \geq 0$ such that the following conditions hold:

(1) for all structures $S_1, ..., S_m$ such that $\sum_{i=1}^{m} lng(S_i) \leq c$, the property $prop(STR(S_1, ..., S_m))$ holds;

(2) for all structures $S_1, ..., S_m$ such that $\sum_{i=1}^{m} lng(S_i) > c$, there exist terms $T_1, ..., T_m$ for which $\sum_{i=1}^{m} lng(T_i) < \sum_{i=1}^{m} lng(S_i)$ and $prop(STR(T_1, ..., T_m)) \to prop(STR(S_1, ..., S_m))$.

Let $prop(rep(v, S, body))$ denote a property expressed by a first-order logic formula with the only free variable S. The formula is constructed from the replacement operation $rep(v, S, body)$, functional symbols, variables and constants by means of Boolean operations, first-order quatifiers and substitution of constants for variables from v.

The following principle is easily proved by induction on $k = lng(S)$.

Induction principle 2. The property $prop(rep(v, S, body))$ holds for each structure S if there exists an integer $c \geq 0$ such that the following conditions hold:

(1) for each structure S such that $lng(S) \leq c$, the property $prop(rep(v, S, body))$ holds;

(2) for each structure S such that $lng(S) > c$, there exists a term $T(S)$ for which $lng(T(S)) < lng(S)$ and $prop(rep(v, T(S), body)) \to prop(rep(v, S, body))$.

Notice that induction principles [14, 15] are the special cases of the principles when $c = 0$.

5. Case of study: programs over linear lists

Let us consider Pascal pointer programs. We will use the method from [10] to describe axiomatic semantics of these programs. Let L be a set of elements to which pointers can refer. An element to which a pointer p refers is denoted by $p\uparrow$ in programs or by $\subset p \supset$ in specifications, or by $L\subset p\supset$ in specifications when it belongs to the set L. We will denote the predicate $\subset p\supset \in L$ as $pnto(L, p)$. Let $upd(L, \subset p\supset, e)$ be a set resulted from the set L by replacing its element to which the pointer p refers with the value of the expression e. In the case when the set L consists of records with the fields k_i $(i = 1, ..., m)$, we use $upd(L, \subset p\supset, (k_1, ..., k_m), (e_1, ..., e_m))$ to denote a set resulted from the set L by replacing its element to which the pointer p refers with an element such that its field k_i is the previously calculated value of the expression e_i $(i = 1, ..., m)$, and the other fields are not changed.

To generate verification conditions for programs which contain statements over the set L, such as $q\uparrow := e, new(p), dispose(r)$, we use their equivalent forms: $L := upd(L, \subset q\supset, e)$ when $pnto(L, q)$, $L := L \cup \{\subset p\supset\}$ when $\neg pnto(L, p)$, $L := L - \{\subset r\supset\}$ when $pnto(L, r)$, respectively. Let us extend Pascal programs by a statement $q\uparrow.(k_1, ..., k_m) := (e_1, ..., e_m)$ which is defined when $pnto(L, q)$ and is equivalent to the statement $L := upd(L, \subset q\supset, (k_1, ..., k_m), (e_1, ..., e_m))$. This statement realizes the synchronous assignment of the values of expressions $e_1, ..., e_m$ to the corresponding fields $k_1, ..., k_m$ of the element $\subset q\supset$. In the case of $m = 1$, the statement has the form $q\uparrow.k := e$ which is equivalent to the statement $L := upd(L, \subset q\supset, k, e)$.

In the rest of this paper we assume that the set L consists of records with the fields $key, count$ and $next$. The key field contains the identification name for an element, and, therefore, the names are different for different elements. The $count$ field containing a positive integer is used for calculation of the number of identical elements belonging to input data. The $count$ field can be omitted. The $next$ field contains a pointer or nil.

The predicate $reach(L, p, q)$ means that the element $\subset q\supset$ is reached from the element $\subset p\supset$ in the set L [10]. Let $p = root(L)$ be a pointer to a head element of the set L, i.e. such an element from which other elements of the set L can be reached. Thus, the relation $p = root(L)$ is defined by the formula $pnto(L, p) \wedge \forall q(pnto(L, q) \wedge \subset q\supset \neq \subset p\supset \to reach(L, p, q))$. Let $l = last(L)$ be such an element of the set L that the field $l.next$ contains nil or a pointer to an element which does not belong to the set L. The predicate $linset(L)$ means that the set L is linear, i.e. L is a nonempty set for which there exists a pointer $p = root(L)$ and an element $l = last(L)$. Notice that there exists the only pointer $root(L)$ and the only element $last(L)$ for the linear set L.

Let us define several useful operations over linear sets. A linear set which contains the only element l is denoted by $set(l)$. Let us assume that L_1 and L_2 are disjoint linear sets such that if the field $last(L_2).next$ contains a pointer p, then $\neg pnto(L_1, p)$. We define their concatenation as a linear set $L = con(L_1, L_2)$ such that $L = L_1 \cup L_2$, $root(L) = root(L_1)$, $last(L) = last(L_2)$, and the pointer $root(L_2)$ is in the field $last(L_1).next$. We consider $con(L, l)$ and $con(l, L)$ to be a short form for $con(L, set(l))$ and $con(set(l), L)$, respectively. A linear set $con(con(L_1, L_2), L_3)$ is denoted by $con(L_1, L_2, L_3)$. A sequence which is the projection of the linear set L on the key field is denoted by $L.key$. Let $mset(L)$ be the multiset $\cup l.count \cdot l.key$ which consists of elements $l.key$ for $l \in L$, and the element $l.key$ appears in the multiset $l.count$ times.

The predicate $linlist(L)$ means that a set L is a linear list, i.e. L is a linear set and $last(L).next = nil$. For a linear list L presented by a data structure, we define a hierarchical data structure $pn(L)$ which represents a sequence of pointers to consecutive elements of the linear list L as

$(choo(pn(L)), rest(pn(L))) =$ **if** $empty(L)$ **then** (ω, ω) **else if** $empty(rest(L))$ **then**

$(root(set(choo(L))), pn(rest(L)))$ **else** $(root(head(L)), pn(rest(L)))$.

Notice that this definition corresponds to the definition of hierarchical structures from Section 2 which forbids the use of the notion $root(L)$, although in this case the definition of $pn(L)$ can be simplified.

6. Examples

Example 1. Reversal of a linear list.

To specify a program for an in-situ reversal of a linear list, we introduce a reversal function rev which is defined for nonempty sequences. Let $rev([a]) = a$, $rev(con(seq, a)) = con(a, rev(seq))$, where $[a]$ is a sequence which consists of the only element a, and also $con(a, seq)$ and $con(seq, a)$ are the concatenation operations for the sequence seq and the element a.

The following annotated program inverts an initial value L_0 of a linear list L by the change of $next$ fields of its elements.

$$\{P\}\ y := nil;\ \textbf{for}\ x\ \textbf{in}\ pn(L)\ \textbf{do}\ x\uparrow.next := y;\ \ y := x\ \textbf{end}\ \{Q\}$$

where $P : linlist(L_0) \wedge L = L_0$, $Q : linlist(L) \wedge L.key = rev(L_0.key)$.

The iteration body is represented as $(L, y) := body(L, y, x)$, where $body(L, y, x) = (upd(L, \subset x \supset, next, y), x)$. Let $S = pn(L)$ and $vec(S) = [s_1, ..., s_n]$. Notice that the iteration body changes the only element $L\subset x\supset$ of the linear list L for $x = s_i$, and, therefore, does not change the structure $rest^i(S)$ $(i = 1, ..., n-1)$. Thus, this iteration satisfies the definition of iteration semantics from Section 2. Projections of pairs $body(L, y, x)$ and $rep((L, y), S, body)$ on the i-th element are denoted by $body_i(L, y, x)$ and $rep_i((L, y), S, body)$, respectively $(i = 1, 2)$.

The following verification condition is generated with the help of the proof rule $rl1$ [14].

$$VC.P \rightarrow Q(L \leftarrow rep_1((L, nil), S, body)).$$

To prove VC, we connect L and S. Let $L\subset S\supset$ be a set of such elements of L to which pointers from $memb(S)$ refer. In the case of $empty(S)$ we assume that $L\subset S\supset$ is empty. It follows from this that $L = L\subset S\supset$ for $S = pn(L)$. We consider $rep_i(S)$ to be a short form for $rep_i((L\subset S\supset, nil), S, body)$ $(i = 1, 2)$.

Claim 1. In the case of $\neg empty(S)$ the following properties hold :
1.1. $rep_2(S) = last(S)$,
1.2. $rep_2((L\subset S\supset, nil), head(S), body) = rep_2(head(S))$.

Proof. By Theorem 5 [14], property 1.1 follows from $body_2(L, y, x) = x$. In the case of $empty(head(S))$ both parts of the equality 1.2 are equal to nil. Let us consider the case $\neg empty(head(S))$. Then $rep_2(head(S)) = last(head(S))$. It remains to notice that, by Theorem 5 [14],

$$rep_2((L\subset S\supset, nil), head(S), body) = last(head(S)).$$

Claim 2. In the case of $\neg empty(S)$,

$$rep_1((L\subset S\supset, nil), head(S), body) = rep_1(head(S)) \cup \{L\subset last(S)\supset\}.$$

Proof. Notice that $L\subset S\supset = L\subset head(S)\supset \cup \{L\subset last(S)\supset\}$. If $empty(head(S))$, then

$$rep_1((L\subset S\supset, nil), head(S), body) = L\subset S\supset = \{L\subset last(S)\supset\}$$

and the set $rep_1(head(S))$ is empty. Claim 2 follows from this.

Let us consider the case $\neg empty(head(S))$. By definition, the set $rep_1((L\subset S\supset, nil), head(S), body)$ is calculated with the help of $body_1$. Among the elements of $L\subset S\supset$, $body_1$ changes the elements of the form $L\subset x\supset$ for $x \in head(S)$. It remains to notice that, by Claim 1, the result of the change is defined by the structure $head(S)$.

The verification condition VC immediately follows from the property

$$\begin{aligned} prop(rep_1(S)) &= (linset(L\subset S\supset) \to linlist(rep_1(S)) \wedge root(rep_1(S)) \\ &= last(S) \wedge rep_1(S).key = rev(L\subset S\supset .key)). \end{aligned}$$

Claim 3. The property $prop(rep_1(S))$ holds.

Proof. We apply induction principle 2 for c=1 and $T(S) = head(S)$. When the set $L\subset S\supset$ consists of the only element, $\neg empty(S)$ and $empty(head(S))$ hold. By Theorem 5 [14], $rep_1(S) = body_1(L\subset S\supset, nil, last(S)) = upd(\{L\subset last(S)\supset\}, \subset last(S)\supset, next, nil)$. Therefore, the property $prop(rep_1(S))$ holds. Let us suppose $\neg empty(head(S))$ and $linset(L \subset S \supset)$. From the inductive hypothesis for $head(S)$, $linset(L\subset head(S)\supset)$, Claims 1, 2 and Theorem 5 [14] it follows that

$$\begin{aligned} rep_1(S) &= body_1(rep_1(head(S)) \cup \{L\subset last(S)\supset\}, last(head(S)), last(S)) \\ &= upd(rep_1(head(S)) \cup \{L\subset last(S)\supset\}, \subset last(S)\supset, next, last(head(S))) \\ &= rep_1(head(S)) \cup upd(\{L\subset last(S)\supset\}, \subset last(S)\supset, next, last(head(S))) \\ &= con(L\subset last(S)\supset, rep_1(head(S))). \end{aligned}$$

Therefore, $linlist(rep_1(S))$ and $root(rep_1(S)) = last(S)$. It remains to notice that

$$\begin{aligned} rep_1(S).key &= con(L\subset last(S)\supset.key, rep_1(head(S)).key) \\ &= con(L\subset last(S)\supset.key, rev(L\subset head(S)\supset.key)) \\ &= rev(con(L\subset head(S)\supset.key, L\subset last(S)\supset.key)) \\ &= rev(con(L\subset head(S)\supset, L\subset last(S)\supset).key) = rev(L\subset S\supset.key). \end{aligned}$$

Example 2. Search in a linear list with reordering.

Let us consider a program for a search of a key k in a linear list L with reordering. The program scans elements of the linear list L and stores the previous element. Two cases are possible. If the key k has been detected, the *count* field of the corresponding element is increased by 1. When this element is not first, it is transfered to the head of the list L by changing *next* fields. If the key k has not been detected, a new element with the key k and 1 in the *count* field is added to the head of the list L. To specify the program, we introduce a function seq/a which denotes a sequence resulted from the sequence seq by elimination of the first occurrence of the element a. If a does not belong to seq, then $seq/a = seq$.

The annotated program $prog1$ is represented in the form:

$\{P\}\, y := nil;\ r := root(L);$ **for** x **in** $pn(L)$ **do**
$body_1(L, y, x);$ **if** $x\uparrow.key = k$ **then** $EXIT;\ body_2(L, y, x)$ **end** $\{Q\}$,

where

$body_1(L, y, x)$: **if** $x\uparrow.key = k$ **then begin** $x\uparrow.count := x\uparrow.count + 1;$
if $y \neq nil$ **then begin** $y\uparrow.next := x\uparrow.next;\ x\uparrow.next := r$ **end end**,
$body_2(L, y, x)$: **if** $x\uparrow.next = nil$ **then begin** $new(z);\ z\uparrow.(key, count, next) := (k, 1, r)$ **end**
else $y := x$,
$P : L = L_0 \wedge linlist(L_0),\ Q : linlist(L) \wedge L.key = con(k, L_0.key/k) \wedge mset(L) = mset(L_0) \cup \{k\}$.

Let $S = pn(L)$ and $vec(S) = [s_1, ..., s_n]$. Notice that when $s_i\uparrow.key \neq k$, the statement $body_2$ can change the only variable y. Therefore, the iteration body does not change the structure $rest^i(S)$ $(i = 1, ..., n-1)$. Thus, this iteration satisfies the definition of iteration semantics from Section 2.

We apply Theorem 1 to eliminate the exit statement EXIT. Conditions of Theorem 1 hold since the statement $body_1(L, y, x)$ does not change the field $x\uparrow.key$, and when $x' < x$, the statement

$body_2(L, y, x')$ does not change this field because $x'\uparrow.next \neq nil$. By Theorem 1, program $prog1$ with an initial value L_0 of the variable L is equivalent to the following program $prog2$:

$\{P\}\, y := nil;\ r := root(L);$ **for** x **in** $ET(S)$ **do** $body_1(L, y, x);$
$body_2(L, y, x)$ **end**; **if** $ET(S) \neq S$ **then** $body_1(L, y, s_t)\ \{Q\}$

where $t = |memb(ET(S))| + 1$ and $ET(S)$ is defined from S, L_0, $cond(L_0, x) = (L_0 \subset x \supset .key = k)$.

By Lemma 2.2, $L_0 \subset x \supset .key \neq k$ for all $x \in ET(S)$. Therefore, the statement $body_1$ does not change $L = L_0$ in the iteration body. The statement $body_2$ can change L for $x = last(ET(S))$ only. Hence, the statement $body_1$ does not change the values of the variables in the iteration body from which $body_1$ can be eliminated. Thus, program $prog2$ is equivalent to the following program $prog3$:

$\{P\}\, y := nil;\ r := root(L);$ **for** x **in** $ET(S)$ **do** $body_2(L, y, x)$ **end**;
if $ET(S) \neq S$ **then** $body_1(L, y, s_t)\ \{Q\}$.

To simplify verification conditions, we consider two cases. When $ET(S) = S$, $\neg empty(ET(S))$ and program $prog3$ is equivalent to the following program $prog4$:

$\{P\}\, y := nil;\ r = root(L);$ **for** x **in** $head(S)$ **do** $body_2(L, y, x)$ **end**; $body_2(L, y, last(S))\ \{Q\}$.

From $L_0 \subset x \supset .next \neq nil$ for all $x \in head(S)$ it follows that the iteration can change the variable y only. As $last(S)\uparrow.next = nil$, the statement $body_2(L, y, last(S))$ has the following form:

$new(z); z\uparrow.(key, count, next) := (k, 1, r)$.

Thus, verification of the program $prog4$ is reduced to proving the following verification condition:

$VC1.\ P \wedge ET(S) = S \rightarrow Q(L \leftarrow upd(L \cup \{\subset z \supset\}, \subset z \supset, (key, count, next), (k, 1, root(L))))$.

When $ET(S) \neq S$, $L_0 \subset x \supset .next \neq nil$ and $L \subset x \supset = L_0 \subset x \supset$ for all $x \in ET(S)$. Therefore, the statement $body_2(L, y, x)$ has the form $y := x$ in program $prog3$. If $\neg empty(ET(S))$, then the loop from $prog3$ is represented as iteration over the structure $head(ET(S))$ with the body $y := x$, followed by the statement $y := last(ET(S))$. This iteration can be eliminated. Notice that by Lemma 2.3, $L_0 \subset s_t \supset .key = k$. It follows from this that $s_t\uparrow.key = k$, and $body_1(L, y, s_t)$ can be simplified in $prog3$. Thus, program $prog3$ is equivalent to the following program $prog5$:

$\{P\}\, y := nil;\ r := root(L);$ **if** $\neg empty(ET(S))$ **then** $y := last(ET(S));\ s_t\uparrow.count := s_t\uparrow.count + 1;$
if $y \neq nil$ **then begin** $y\uparrow.next := s_t\uparrow.next;\ s_t\uparrow.next := r$ **end** $\{Q\}$.

If $empty(ET(S))$, then $t = 1$. Otherwise, $t > 1\ \ last(ET(S)) = s_{t-1}$. Verification of the program $prog5$ is reduced to proving the following verification conditions:

$VC2.\ P \wedge empty(ET(S)) \rightarrow Q(L \leftarrow upd(L, \subset s_1 \supset, count, \subset s_1 \supset .count + 1))$,
$VC3.\ P \wedge \neg empty(ET(S)) \wedge ET(S) \neq S \rightarrow Q(L \leftarrow L')$

where

$L' = upd(upd(upd(L, \subset s_t \supset, count, \subset s_t \supset .count + 1), \subset s_{t-1} \supset, next,\ \subset s_t \supset .next), \subset s_t \supset, next, root(L))$.

Claim 4. The verification condition $VC1$ holds.

Proof. Let $L' = upd(L \cup \{\subset z \supset\}, \subset z \supset, (key, count, next), (k, 1, root(L)))$. Then $L' = con(\subset z \supset, L)$ since $\subset z \supset .next = root(L)$. It follows from this that $linlist(L')$. By Lemma 2.2, $L_0 \subset x \supset .key \neq k$ for all $x \in S$. Therefore, $k \notin L_0.key$. It follows from the condition P that $L = L_0$. Hence,

$L'.key = con(\subset z \supset .key, L.key) = con(k, L_0.key) = con(k, L_0.key/k)$ and $mset(L') = mset(L_0) \cup \{k\}$.

Claim 5. The verification condition $VC2$ holds.

Proof. Let $L' = upd(L, \subset s_1 \supset, count, \subset s_1 \supset .count + 1)$. Then $linlist(L')$. Two cases are possible. If $empty(head(S))$, then L consists of the only element $L \subset s_1 \supset$. By Lemma 2.3, $L_0 \subset s_1 \supset .key = k$. Therefore, $L_0.key/k$ is an empty sequence and $L'.key = L.key = con(k, L_0.key/k)$. It is evident that $mset(L') = \{L' \subset s_1 \supset .key\} \cdot L' \subset s_1 \supset .count = \{k\} \cdot (L \subset s_1 \supset .count + 1) = mset(L) \cup \{k\}$, where $\{b\} \cdot m$ denotes a multiset consisting of the element b which occurs m times. When $\neg empty(head(S))$, the linear list L is represented as $L = con(L\subset s_1\supset, L_1)$ where $L_1 = rest(L)$. It follows from this that $L' = con(L'\subset s_1\supset, L_1)$. Therefore, $L'.key = con(L'\subset s_1\supset.key, L_1.key) = con(k, L_1.key)$ and $L_0.key/k = con(L\subset s_1\supset.key, L_1.key)/k = L_1.key$. It remains to notice that

$$\begin{aligned} mset(L') &= \{L'\subset s_1\supset.key\}\cdot L'\subset s_1\supset.count \cup mset(L_1) \\ &= \{L\subset s_1\supset.key\}\cdot (L\subset s_1\supset.count + 1) \cup mset(L_1) \\ &= mset(L) \cup \{k\}. \end{aligned}$$

Claim 6. The verification condition $VC3$ holds.

Proof. Two cases are possible: $t = n$ or $1 < t < n$. In the case of $t = n$, the linear list L is represented as $L = con(L_1, L\subset s_{t-1}\supset, L\subset s_t\supset)$ for a suitable linear set L_1. If $empty(L_1)$, then similar reasoning can be developed. Notice that the set L' is represented as $L' = con(L'\subset s_t\supset, L_1, L'\subset s_{t-1}\supset)$, since $L'\subset s_t\supset.next = root(L) = root(L_1), L'\subset s_{t-1}\supset.next = L\subset s_t\supset.next = nil$. Therefore, $linlist(L')$.

By Lemma 2.3, $L'\subset s_t\supset.key = L\subset s_t\supset.key = L_0\subset s_t\supset.key = k$. By Lemma 2.2, $k \notin con(L_1.key, L\subset s_{t-1}\supset.key)$. Therefore,

$$\begin{aligned} L'.key &= con(L'\subset s_t\supset.key, L_1.key, L'\subset s_{t-1}\supset.key) \\ &= con(k, L_1.key, L\subset s_{t-1}\supset.key) \\ &= con(k, L.key/k) \end{aligned}$$

and

$$\begin{aligned} mset(L') &= \{L'\subset s_t\supset.key\}\cdot L'\subset s_t\supset.count \cup mset(L_1) \cup \{L'\subset s_{t-1}\supset.key\}\cdot L'\subset s_{t-1}\supset.count \\ &= \{L\subset s_t\supset.key\}\cdot (L\subset s_t\supset.count + 1) \cup mset(L_1) \cup \{L\subset s_{t-1}\supset.key\}\cdot L\subset s_{t-1}\supset.count \\ &= mset(L) \cup \{k\}. \end{aligned}$$

In the case of $1 < t < n$, the linear list L is represented as $L = con(L_1, L\subset s_t\supset, L_2)$ for a suitable linear set L_1 and a linear list L_2. Therefore, the set L' is represented as $L' = con(L'\subset s_t\supset, L_1, L_2)$. It follows from this that $linlist(L')$. By Lemma 2, $k \notin L_1.key$ and $L'\subset s_t\supset.key = L\subset s_t\supset.key = k$. Hence, $L'.key = con(k, L_1.key, L_2.key) = con(k, L.key/k)$. It remains to notice that

$$\begin{aligned} mset(L') &= \{L'\subset s_t\supset.key\}\cdot L'\subset s_t\supset.count \cup mset(L_1) \cup mset(L_2) \\ &= \{L\subset s_t\supset.key\}\cdot (L\subset s_t\supset.count + 1) \cup mset(L_1) \cup mset(L_2) \\ &= mset(L) \cup \{k\}. \end{aligned}$$

7. Conclusion

The development of the symbolic method for verification of definite iterations over hierarchical data structures aimed to apply it to pointer programs is described in the paper. When compared to [14, 15], the method is generalized in two aspects allowing for a restricted change of the structure by the iteration body and exit from the iteration body under a condition. This generalization substantially extends the field of application of the symbolic method since definite iterations with exit from their bodies allow us to represent important cases of while-loops.

In the first stage of verification, definite iterations with exit from their bodies are transformed to standard definite iterations over hierarchical data structures. Theorem 1 justifies correctness of this transformation, and Lemma 2 describes useful properties of hierarchical structures which are used by this transformation. In the second stage, verification conditions which can contain the replacement operation are generated. In the third stage, verification conditions are proved with the help of both a universal technique based on the induction principles and a problem-oriented technique based on notions related to the problem domain. The notions for programs over linear lists are described in Section 5.

Instead of loop invariants, the symbolic method uses properties of both hierarchical structures and the replacement operation. These properties, as a rule, are simpler than loop invariants, and new notions are not necessary for representation of the properties. The induction principles 1 and 2 are rather flexible and allow us to use different induction strategies for proving the properties. The use of properties of hierarchical data structures simplifies presentation of the properties of the replacement operation as well as proving verification conditions.

Partial verification of a program for reversal of a linear list has been described in [2] but the basic property of the program has not been proved in [2]. N. Wirth has considered a program for a search in a linear list with reordering as a challenge for verification [10]. This program has been considered in [10] where its partial verification has been described. It should be noted that the programs from [2] and [10] use while- and repeat-loops which are attended with invariants. The symbolic method allows us to perform the complete verification of such programs which are represented by definite iterations over hierarchical data structures. Verification of the program (see example 2) similar to that from [10] is performed without loop invariants and the replacement operation owing to Theorem 1 and elementary transformations for the loop elimination.

We suggest to extend the symbolic method to a new kind of definite iterations over tuples of data structures for the purpose of a natural representation of loops with several input data structures.

References

[1] S.K. Basu, J. Misra, *Some classes of naturally provable programs*, Proc. 2nd Intern. Conf. on Software Engineering, IEEE Press, 1976, 400–406.

[2] M. Benedikt, T. Reps, M. Sagiv, *A decidable logic for describing linked data structures*, Proc. ESOP/ETAPS'99, Lect. Notes Comput. Sci., **1576**, 1999, 2–19.

[3] P. Fradet, R. Gaugne, D. Le. Metayer, *Static detection of pointer errors: an axiomatisation and a checking algorithm*, Proc. ESOP'96, Lect. Notes Comput. Sci., **1058**, 1996, 125–140.

[4] D. Gries, N. Gehani, *Some ideas on data types in high-level languages*, Comm. ACM, **20**, No 6, 1977, 414–420.

[5] E.C.R. Hehner, A.M. Gravell, *Refinement semantics and loop rules*, Proc. FM'99, Lect. Notes Comput. Sci., **1709**, 1999, 1497–1510.

[6] C.A.R. Hoare, *An axiomatic basis of computer programming*, Comm. ACM, **12**, No 10, 1969 576–580.

[7] C.A.R. Hoare, *A note on the for statement*, BIT, **12**, No 3, 1972, 334–341.

[8] J.L. Jensen, M.E. Jorgensen, N. Klarlund, M.I. Schwartzbach, *Automatic verification of pointer programs using monadic second-order logic*, ACM SIGPLAN Notices, **32**, No 5, 1997, 226–234.

[9] R.C. Linger, H.D. Mills, B.I. Witt, *Structured Programming: Theory and Practice*, Addison-Wesley, Reading, MA, 1979.

[10] D.C. Luckham, N. Suzuki, *Verification of array, record and pointer operations in Pascal*, ACM Trans. on Programming Languages and Systems, **1**, No 2, 1979, 226–244.

[11] H.D. Mills, *Structured programming: Retrospect and prospect*, IEEE Software, **3**, No 6, 1986, 58–67.

[12] V.A. Nepomniashy, *Loop invariant elimination in program verification*, Programming and Comput. Software, No 3, 1985, 129–137 (English translation of Russian Journal "Programmirovanie").

[13] V.A. Nepomniaschy, *On problem-oriented program verification*, Programming and Comput. Software, No 1, 1986, 1–9.

[14] V.A. Nepomniaschy, *Symbolic verification method for definite iteration over data structures*, Information Processing Letters, No 69, 1999, 207–213.

[15] V.A. Nepomniaschy, *Verification of definite iteration over hierarchical data structures*, Proc. FASE/ETAPS'99, Lect. Notes Computer Sci., **1577**, 1999, 176–187.

[16] J. Stark, A. Ireland, *Invariant discovery via failed proof attempts*, Proc. LOPSTR'98, Lect. Notes Comput. Sci., **1559**, 1999, 271–288.

[17] A.M. Stavely, *Verifying definite iteration over data structures*, IEEE Trans. Software Engineering, **21**, No 6, 1995, 506–514.