A Structural Approach to Pointer Data Types

TAIICHI YUASA*

Programs with pointers tend to be difficult to understand and error-prone. At the same time, verification of such programs is often quite complex. Thus it is desirable to restrict the use of pointers while maintaining efficiencies provided by pointer manipulation. To this end, a concept is introduced for tree structures with the mechanism of direct access to subtrees. It is proposed to embed this structuring concept into programming languages as an abstract data type basic to the languages. Examples are given, which suggests that this approach provides a clean view of pointer operations and eases verification of programs while reasonably maintaining efficiency of pointer manipulations.

1. Introduction

It has been recognized that use of pointers tends to increase difficulties of programs understanding, specifications, and especially verification, thus is error-prone. It is not easy to localize the effect of pointer operations. The simplest view is that a pointer operation affects the state which is determined by the contents of all cells accessible to the program. Program verification methods based on this view are inevitably accompanied by high complexities and are not applicable to actual programs though several strategies have been proposed to ease the difficulty [1, 2, 5]. Secondly it is often very difficult to give specifications to programs manipulating pointers. For instance, it is difficult to define explicitly what a 'linear list' is, which would be necessary to prove the simplest 'linear search' program.

These difficulties result from the arbitrary use of pointers. Therefore it is desirable to restrict the use of pointers while maintaining their merits of flexibility. For this purpose, we regard pointer data structures as objects of abstract data types. That is, we put a certain restriction on pointer data structures and introduce operations so that they preserve the restriction. With this approach, the effect of an operation is limited to the data objects which appear explicitly in the program text. Also the structure of data objects is characterized by the structural induction on the data types.

For most applications, data structures can be represented by some kinds of trees (including lists). Obviously there are two exceptions: cyclic structures and data sharing. Cyclic structures cause no problem as we will see later in this paper. As for data sharing, there are two cases: essential and inessential.

Fig. 1 illustrates a data structure which occurs frequently in pointer applications. Although the data object A is shared by two pointers P1 and P2, A can be regarded as a component of a larger data object B and the pointer

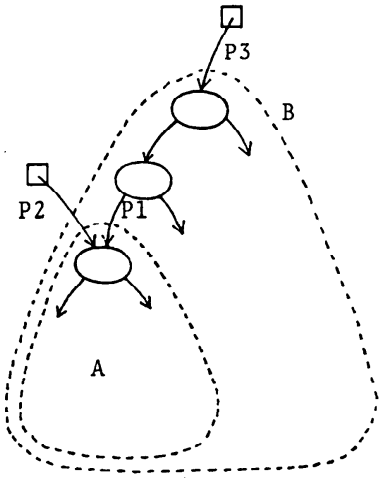

Fig. 1 Inessential sharing.

P2 is considered to point to a substructure of B. By denoting explicitly that modification through the pointer P2 affects B and information retrieved through P2 is contained in B, the ability to understand and verifiy programs will be increased. We call this kind of data sharing 'inessential'. Without this kind of sharing, i.e., without the mechanism to directly locate substructures, operations on trees would be quite inefficient. On the other hand, in case of 'essential' sharing in Fig. 2, the shared data object A is possibly a component of two (or more) data objects; thus it would be difficult in this case to locate the effect of pointer operations. Although the importance of essential sharing in system programming is not small, a fairly large portion of pointer manipulations can be covered by the inessential kind.

For the above reasons, we limit data sharing to the inessential kind. Here, the use of pointers falls into three classes:

Class 1. Those which constitute data structures such as P1 in Fig. 1. (Those pointers are accessed from other pointers, whereas those in the classes below are not.)

Class 2. Those which represent data objects such as

*Research Institute for Mathematical Sciences, Kyoto University.

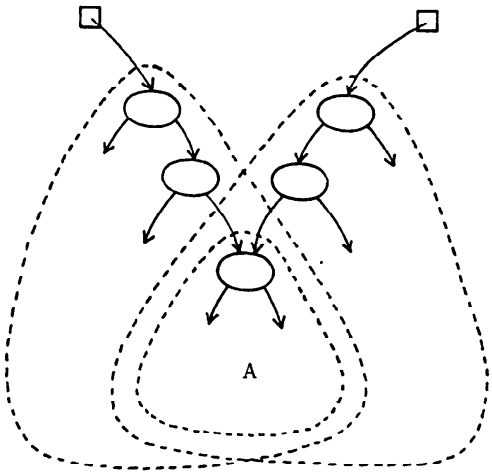

Fig. 2 Essential sharing.

P3 (e.g. the root pointer to a tree structure).

Class 3. Those which point to a location within a data structure such as P2. (They are used to access directly a portion of a data structure.)

In this paper, we introduce trees, as an abstract data type which has the facility of inessential sharing. There, the use of pointers in the first class are embedded in the language mechanism to build data structures, and those in the second to bind tree objects with variables. The third kind of pointers are regarded as another data type specially introduced to identify subtrees.

2. T-trees

In general, a tree may be EMPTY (an empty tree); or else a tree X is a tuple consisting of trees (called direct subtrees of X), and of a node which contains a value of a certain type called node type. A tree X is called a 'subtree' of a tree Y if either X is identical to Y or X is a subtree of one of the direct subtrees of non-EMPTY Y. In order to identify a subtree of a tree, we introduce the notion of T-trees (short for tagged trees). A T-tree is a tree, each subtree of which is attached with a tag unique to the subtree (see Fig. 3). Thus each subtree of a T-tree is also a T-tree.

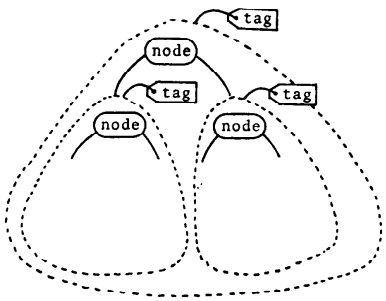

Fig. 3 A (binary) T-tree.

Each tag is an element of the data type TAG. TAG contains infinitely many elements to identify non-empty T-trees, as well as a distinguished element NIL to identify the empty T-tree.

(Remark: It is possible to realize cyclic structures with T-trees. For this purpose, we adopt as the node type a record type with as many fields of type TAG as the number of direct subtrees. When the TAG element in a TAG field is not NIL, we interpret that the direct subtree corresponding to the TAG field is actually the subtree identified by the tag. For example, in Fig. 4, 'tag 3' in the second TAG field of the T-tree with 'tag 4' means that its second direct subtree is actually the T-tree with 'tag 3'.)

In the following, we introduce operations by which T-trees are accessed through their identifying tags. As mentioned before, we need to denote explicitly what is affected by these operations and where the information is retrieved from by them. Let us call a T-tree X a supertree of Y when Y is a subtree of X and, in case X is not a subtree of any other T-tree, let us call X the root tree of Y. Since we have excluded the essential sharing, operations on a T-tree affect only its supertrees. In other words, the modification is bounded to the scope of the root tree. In order to make it possible to denote the root tree in program text, we require that there is a one-to-one correspondence between root trees and variables of type T-tree. The requirement is justified for the following reasons: Since a subtree is identified by a tag, we would rather define a variable of type TAG to hold subtrees. Besides, if two variables X and Y of type T-tree were bound to a single T-tree, then the effect of modification through X would be invisible for an operation to retrieve information through Y.

3. Operations on Trees

In the sequel, we introduce functions and procedures available to the programmer. These operations are basic in the sense that they are provided by the language system and are the only operations by which the programmer

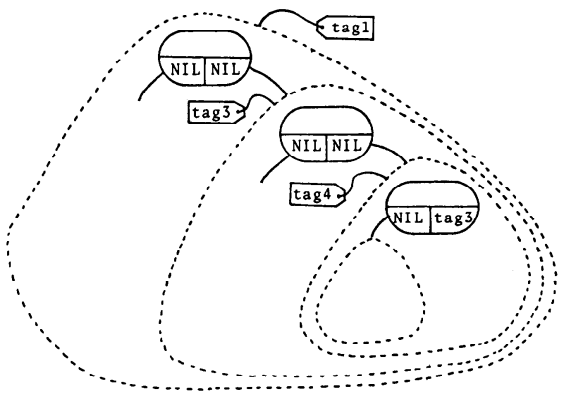

Fig. 4 Cyclic structures possible

can access data structures directly. They constitute a complete set of operations. That is, the programmer can construct any T-tree and retrieve any information stored in T-trees. For simplicity, we limit our discussion to the type B-TREE of T-trees with exactly two direct subtrees and with the node type integer. (In Fig. 3 was an object of B-TREE.)

GET1: B-TREE→B-TREE
GET2: B-TREE→B-TREE
GETN: B-TREE→integer
HEAD: B-TREE→TAG

These are functions to retrieve information from B-trees. GET1 and GET2 (collectively denoted as GET*i*) return the first and second direct subtrees, respectively. GETN(X) returns the node of X and HEAD(X) the tag attached to X.

TAGGET: (B-TREE, TAG)→B-TREE

This function corresponds to the dereferencing operation in conventional languages. That is, TAGGET(X, T) returns the subtree of X with the tag T.

Assignment to tree variables should be forbidden for the following reasons. In most languages, by executing an assignment statement 'X:=E', either a copy of E is generated to which X is bound, or X is bound to the object E itself. In the latter case, obviously our requirement is not satisfied. In the former case, on the other hand, the requirement would be met, but copying is just not what we want. Thus trees are modified through basic procedures by sending them as variable parameters. In the following procedure declarations, the types to the left of vertical bar '|' are those for variable parameters, and the types to the right for value parameters.

SETEMPTY: (B-TREE|)
SETNEW: (B-TREE|integer)

By SETEMPTY(X|), the B-TREE variable X is set empty. By SETNEW($X|I$), X is bound to a new tree with the node I and with direct subtrees empty. The new tree is attached with a tag not yet used.

TAGSETN: (B-TREE|TAG, integer)

TAGSETN($X|T$, I) replaces the node of the subtree of X attached with T by I.

We need operations to replace a B-tree by another B-tree in order to modify tree structures. For efficiency purpose, we want to have operations accomplished by simple pointer replacement rather than time-consuming copying. At the same time, we must avoid data sharing between two B-tree objects. As a compromise, when operations below replace a B-tree by a subtree A of Y, A in Y is as well replaced by the empty B-tree.

MOVE: (B-TREE, B-TREE|)
MOVE-*i*: (B-TREE, B-TREE|TAG)
MOVE*i*-: (B-TREE, B-TREE|TAG)
MOVE*i*-*j*: (B-TREE, B-TREE|TAG, TAG)

By MOVE(X, Y|), the variable X is bound to the B-tree to which Y was bound and, at the same time, Y is set empty. By MOVE-*i*(X, $Y|TY$), X is bound to the i-th direct subtree A of the Y's subtree to which the tag TY is attached, and A is replaced by EMPTY. MOVE*i*-(X, $Y|TX$) replaces the i-th direct subtree of the X's subtree with TX by the B-tree in Y and sets Y empty. By MOVE*i*-*j*(X, $Y|TX$, TY), the i-th direct subtree of the X's subtree with TX is replaced by the j-th direct subtrees A of the Y's subtree with TY and A is replaced by EMPTY.

In order to avoid data sharing between two B-tree objects, another restriction is necessary. Parameter passing to operations must be forbidden which may cause data sharing during the execution of the operations. For example, given a function F receiving two T-tree parameters X and Y, the call $F(Z$, GET2(Z)) would cause the second direct subtree of Z to be shared by X and Y during the execution of F.

In Fig. 5, a sample program to construct binary trees is presented. The procedure ADD receives a variable parameter X of B-TREE which is supposed to be a binary search tree, and a value parameter I of Integer. If I is not found in X, then X is expanded to contain I. We will come back to this program later for verification, when formal definitions of binary search trees etc. will be given.

4. Specifying B-trees and Operations on them

In the following, the language ι [8] (or its approgimation) is used to present data types and operations on them formally. This choice of language is not essential and any language which provides clear and rigorous data type definitions with abstraction mechanisms may substitute ι.

```
interface type TAG
  function
    NIL :     → @
    FIRST:    → @
    NEXT:  @  → @
end interface
```

In the interface part, operations on TAG are introduced with their domain and range. '@' presents the data type

```
procedure ADD (X:B-TREE | I:integer)
  var T,T1:TAG; DONE:boolean; J:integer; Y:B-TREE;
  T:=HEAD(X);
  if T=NIL then SETNEW(X|I)
   else
    DONE:=false;
    while ~DONE do
     J:=GETN(TAGGET(X,T));
     if J=I then DONE:=true
      else
       if I<J then T1:=HEAD(GET1(TAGGET(X,T)));
                   if T1=NIL then SETNEW(Y|I);
                                  MOVE1-(X,Y|T);
                                  DONE:=true
                              else T:=T1
                   end if
               else T1:=HEAD(GET2(TAGGET(X,T)));
                   if T1=NIL then SETNEW(Y|I);
                                  MOVE2-(X,Y|T);
                                  DONE:=true
                              else T:=T1
                   end if
       end if
     end if
    end while
  end if
end procedure
```

Fig. 5 A sample program.

being defined, i.e. TAG in this case. FIRST is supposed to be automatically attached to the new tree when the operation SETNEW is called for the first time during program execution. Then each time SETNEW is called, a tag not yet used is generated by calling NEXT.

Now we present the abstract data type B-TREE and operations on it. Some of the operations are not introduced in the previous section. This means that these operations are not executable in the user program and are used only for specification purposes. (Indeed some of the operations below would cause data sharing among two data objects of B-TREE, if they appeared in the program text. For example, by SET1($X|Y$), the B-tree bound to Y would be shared with X.) In the following, explanations are given for only such non-executable operations.

```
interface type B-TREE
  function
    EMPTY:  → @
    NEW: (integer, TAG) → @
    GETi: @ → @
    GETN: @ → integer
  procedure
    SETi: (@|@)
    SETN: (@|integer)
  function
    HEAD: @ → TAG
end interface
```

In the interface part of B-TREE, primitive operations on B-TREE are introduced. Other operations on B-TREE are supposed to be composed in terms of these primitive operations.

EMPTY is a constant function whose value is the empty tree. NEW(I, T) creates a new B-tree with a node I and with EMPTY subtrees. T is the tag attached to the new B-tree. SETi($X|Y$) replaces the i-th direct subtree of X by Y. There are two additional function and procedure (GETN and SETN) to get or set the node of a B-tree.

```
specification type TAG
  var T,S:@;
  axiom
    1.NEXT(NIL)=NIL
    2.NEXT(T)=NEXT(S) ⊃ T=S
    3.NEXT(T)≠FIRST
end specification
```

```
specification type B-TREE
  var X,Y:@; I:integer; T:TAG;
  domain
    NEW(I,T)   on T≠NIL
    GETi(X)    on X≠EMPTY
    GETN(X)    on X≠EMPTY
    SETi(X|Y)  on SETi-DOM(X,Y)
    SETN(X|I)  on X≠EMPTY
  axiom
    1.T≠NIL ⊃ HEAD(NEW(I,T)) = T
    2.T≠NIL ⊃ GETi(NEW(I,T)) = EMPTY
    3.T≠NIL ⊃ GETN(NEW(I,T)) = I
    4.X≠EMPTY ∧ SETi-DOM(X,Y) ⊃ HEAD(SETi(X|Y)) = HEAD(X)
    5.X≠EMPTY ∧ SETj-DOM(X,Y)
      ⊃ GETi(SETj(X|Y)) = Y (case i=j)
                        = GETi(X) (case i≠j)
    6.X≠EMPTY ∧ SETi-DOM(X,Y) ⊃ GETN(SETi(X|Y)) = GETN(X)
    7.X≠EMPTY ∧ SETi-DOM(X,Y) ⊃ HEAD(SETi(X|Y)) = HEAD(X)
    8.X≠EMPTY ⊃ GETi(SETN(X|I)) = GETi(X)
    9.X≠EMPTY ⊃ GETN(SETN(X|I)) = I
   10.X≠EMPTY ⊃ HEAD(SETN(X|I)) = HEAD(X)
   11.X=EMPTY ≡ HEAD(X)=NIL
end specification
```

The specification part consists of axioms in a many-sorted first-order logic (Free variables in the axioms are assumed to be universally quantified.) and domain specifications for some of the operations. (For the other operations, their domain specifications are assumed to be true.) Calls to an operation is valid if the arguments satisfy the condition in the domain specification for the operation. Otherwise, execution ends with abnormal condition. (Note: In the specification part, procedures are regarded as functions whose return values are those of the variable parameters. Thus HEAD (SETi($X|Y$)) in axiom 4 is the tag of the B-tree to which X is bound after SETi($X|Y$) is executed.)

In order to ensure that a subtree is uniquely identified by a pair of a B-tree and a tag, the domains for SET1 and SET2 are restricted so that, when, say, SET1($X|Y$) is executed for a non-empty B-tree X, a tag attached to a subtree of Y is identical neither to the tag of X nor any tag to a subtree of the X's second direct subtree.

Here, SETi-DOM(X, Y) is short for

```
X≠EMPTY ∧ ~CONT(Y,HEAD(X)) ∧ DISJOINT(GETj(X),Y),
```

where $i \neq j$. DISJOINT(X, Y) is short for

```
∀T.~(CONT(X,T) ∧ CONT(Y,T))
```

CONT is a predicate which determines whether a B-tree contains a subtree with the given TAG element. Fomally, CONT is defined as

```
CONT(X,T) =
   if X=EMPTY ∨ T=NIL then false
     else (HEAD(X)=T ∨ CONT(GET1(X),T) ∨ CONT(GET2(X),T))
```

Indeed, with the restrictions given in the domain specifications on SETi, the uniqueness of tags is guaranteed as asserted in the following lemma.

Lemma

For any X of type B-TREE,

UNIQUETAG(X)

holds, where UNIQUETAG is recursively defined as:

```
UNIQUETAG(X) = if X=EMPTY then true
                 else ~CONT(GET1(X), HEAD(X))
                      ∧ ~CONT(GET2(X), HEAD(X))
                      ∧ DISJOINT(GET1(X), GET2(X))
                      ∧ UNIQUETAG(GET1(X))
                      ∧ UNIQUETAG(GET2(X))
```

This lemma is proved by the generator induction rule (See [8]) on B-TREE, that is,

```
T≠NIL              ∀Y.(SETi-DOM(X,Y)      X≠EMPTY
⊃ P(NEW(I,T))      ⊃ P(SETi(X|Y)))        ⊃ P(SETN(X|I))
-------------------------------------------------------
                       P(X)
```

The lemma above is essential to prove properties of tag operations. If the same tag were possibly attached to more than one subtrees of a tree, then some properties we expect to hold would no longer be valid. For instance,

```
CONT(GET2(X),T) ⊃ TAGGET(GET2(X),T)=TAGGET(X,T)
```

would not hold.

Now we introduce supplementary operations on B-

TREE.

```
interface procedure TAGGET
  function TAGGET:(B-TREE,TAG) → B-TREE
end interface

specification procedure TAGGET
  var  X:B-TREE; T:TAG;
  domain
    TAGGET(X,T) on T=NIL ∨ CONT(X,T)
  axiom
    1. TAGGET(X,NIL) = EMPTY
    2. TAGGET(X,HEAD(X)) = X
    3. X≠EMPTY ∧ T≠NIL ∧ HEAD(X)≠T ∧ CONT(GETi(X),T)
       ⊃ TAGGET(X,T) = TAGGET(GETi(X),T)
end specification


interface procedure TAGSET
  procedure
    TAGSETi:(B-TREE|TAG,B-TREE)
    TAGSETN:(B-TREE|TAG,B-TREE)
end interface

specification procedure TAGSET
  var X,Y:B-TREE; T,T1:TAG; I:integer;
  domain
    TAGSETi(X|T,Y) on TAGSETi-DOM(X,T,Y)
    TAGSETN(X|T,I) on CONT(X,T)
  axiom
    1. T≠NIL ∧ HEAD(X)=T ∧ TAGSETi-DOM(X,T,Y)
       ⊃ TAGSETi(X|T,Y) = SETi(X|Y)
    2. CONT(GETj(X),T) ∧ DISJOINT(GETi(TAGGET(X,T)),Y)
                       ∧ TAGSETi(X,T,Y)
       ⊃ TAGSETi(X|T,Y) = SETj(X|TAGSETi(GETj(X)|T,Y))
    3. T≠NIL ∧ HEAD(X)=T ⊃ TAGSETN(X|T,Y) = SETN(X|Y)
    4. CONT(GETj(X),T)
       ⊃ TAGSETN(X|T,I) = SETj(X|TAGSETN(GETj(X)|T,I))
end specification
```

(where TAGSET*i*-DOM(X, T, Y) is short for

```
CONT(X,T)
∧ ∀T1.(CONT(Y,T1)
          ⊃ ~CONT(X,T1)
            ∨ (CONT(X,T1) ∧ CONT(GETi(TAGGET(X,T)),T1)))
```

TAGSET 1 and TAGSET 2 are non-executable procedures which replace the first subtree and the second subtree, respectively, of the subtree with a given tag.

Finally, we present MOVE operations. Notice the difference between SETNEW introduced here and that in the previous section. In a formal point of view, SETNEW must receive an additional parameter of type TAG to be attached to a new tree. In execution, on the other hand, SETNEW is assumed to be supplied with a new tag element automatically.

(Since MOVE operations receive two variable parameters, the value of X after an execution of MOVE(X, Y) is denoted as MOVE\$1(X, Y|), etc.)

```
interface procedure MOVEOPERATIONS
  procedure
     MOVE:      (B-TREE,B-TREE|)
     MOVE-i:    (B-TREE,B-TREE|TAG)
     MOVEi-:    (B-TREE,B-TREE|TAG)
     MOVEi-j:   (B-TREE,B-TREE|TAG,TAG)
     SETEMPTY:  (B-TREE|)
     SETNEW:    (B-TREE|TAG,integer)
end interface

specification procedure MOVEOPERATIONS
  var X,Y:B-TREE; T,TX,TY:TAG; I:integer;
  domain
     MOVE-i(X,Y|TY)      on CONT(Y,TY)
     MOVEi-(X,Y|TX)      on CONT(X,TX)
     MOVEi-j(X,Y|TX,TY) on CONT(X,TX) ∧ CONT(Y,TY)
  axiom
     1. MOVE$1(X,Y|)=Y
     2. MOVE$2(X,Y|)=EMPTY
     3. CONT(X,TX) ⊃ MOVEi-$1(X,Y|TX)=TAGSETi(X|TX,Y)
     4. CONT(X,TX) ⊃ MOVEi-$2(X,Y|TX)=EMPTY
     5. CONT(Y,TY) ⊃ MOVE-i$1(X,Y|TY)=GETi(TAGGET(Y,TY))
     6. CONT(Y,TY) ⊃ MOVE-i$2(X,Y|TY)=TAGSETi(Y|TY,EMPTY)
     7. CONT(X,TX) ∧ CONT(Y,TY)
        ⊃ MOVEi-j$1(X,Y|TX,TY)
                 = TAGSETi(X|TX,GETj(TAGGET(Y,TY)))
     8. CONT(X,TX) ∧ CONT(Y,TY)
        ⊃ MOVEi-j$2(X,Y|TX,TY) = TAGSETj(Y|TY,EMPTY)
     9. SETEMPTY(X)=EMPTY
    10. SETNEW(X|T,I)=NEW(T,I)
end specification
```

5. An Example of Proving a Program with Trees

Having formally presented the data type B-TREE and operations on it, now we prove the correctness of the procedure ADD in Fig. 5. Here, we attempt to prove the following properties of ADD:

```
property 1. SORTED(X) ⊃ SORTED(ADD(X|I))
property 2. SORTED(X) ⊃ INCLUDE(ADD(X|I),I)
property 3. SORTED(X) ∧ INCLUDE(X,J) ⊃ INCLUDE(ADD(X|I),J)
```

where SORTED is a predicate to determine whether or not a given B-tree is a binary search tree.

```
SORTED(X)
  =if X=EMPTY then TRUE
     else (GET1(X)≠EMPTY ⊃ GETN(GET1(X))<GETN(X))
                ∧ (GET2(X)≠EMPTY ⊃ GETN(X)<GETN(GET2(X)))
                ∧ SORTED(GET1(X))
                ∧ SORTED(GET2(X))
```

INCLUDE is a predicate to be defined as

```
INCLUDE(X,I)
  =if X=EMPTY then FALSE
     else GETN(X)=I ∨ INCLUDE(GET1(X),I)
          ∨ INCLUDE(GET2(X),I)
```

That is, INCLUDE(X, I) holds if and only if the B-tree X has a subtree whose node is equal to I.

We first attach inductive assertions to the while statement. For the property 2, for instance,

```
CONT(X,T) ∧ (DONE ⊃ INCLUDE(X,I))
```

is sufficient. Then the so-called verification conditions are generated in the well-known manner. Since these verification conditions can be proved straightforwardly, here, we simply provide the key formula

```
CONT(X,T) ∧ INCLUDE(TAGGET(X,T),I) ⊃ INCLUDE(X,I)
```

which may be used several times for proving the verification conditions. This formula is proved using the following structural induction on B-TREE:

$$\frac{P(\mathrm{EMPTY}) \qquad X\neq\mathrm{EMPTY} \wedge P(\mathrm{GET1}(X)) \wedge P(\mathrm{GET2}(X)) \supset P(X)}{P(X)}$$

6. Implementation

We discuss briefly how to implement B-trees efficiently. We have required that a subtree is identified by a pair of its root tree and the tag attached to the subtree. Thus a function call, say, TAGGET(X, T) is valid only if the B-tree to which the tag T is attached is a subtree of X. Since tags are implemented by pointers, as far as implementation is concerned, a subtree can be identified

only by the tag attached to the subtree. Thus we would rather avoid runtime tests for such a condition, since these tests include time-consuming tree traverse. The best solution to this problem would be to design a language processor which generates object programs in two modes depending on the user request. In one mode, redundant tests are detected and only necessary tests are inserted into object programs. For example, in the sample program in Fig. 5, the test for the expression TAGGET(X, T) will make it unnecessary to test for the statement MOVE1-(X, $Y|T$) immediately after the expression. In the other mode, no tests are inserted at all. (A similar method is often found for boundary check of array indexing.)

7. Generalization

The discussion so far made on B-trees can be extented to general cases. First of all, the choice of integers as the node type is not essential and can be replaced with any type. Adopting type-parameterization mechanisms to the language, it is even possible to define trees of arbitrary node type [6, 8]. Various kinds of trees with different arities (not necessarily binary) may be necessary in actual programming. Furthermore, a single tree may contain nodes with different types or subtrees of different arities. Note that it is also straightforward to design a language construct which allows the user to define data types of such non-uniform trees by extending the notion of T-trees as discussed in this paper.

Acknowledgement

The author wishes to express his appreciation to Professor Reiji Nakajima for patiently supervising this research.

References

1. BERRY, D. M., ERLICH, Z. and LUCENA, C. J. Pointers and data abstractions in high-level languages. *Computer Languages* 2 (1977), 135–148.
2. BURSTALL, R. M. Some techniques for proving correctness of programs which alter data structures. *Machine Intelligence* 7 (1972), 23–50.
3. HOARE, C. A. R. Recursive data structures. *Stanford AIM-223* (1973).
4. HOARE, C. A. R. and WIRTH, N. An axiomatic definition of the programming language PASCAL. *Acta Informatica* 2 (1973).
5. LAVENTHAL, M. S. Verification of programs operating on structured data. *MIT MAC TR-124* (1974).
6. LISKOV, B. et al. Abstraction mechanisms in CLU. *Comm. ACM.* 8 (1977), 567–576.
7. LUCKHAM, D. and SUZUKI, N. Verification-oriented proof rules for arrays, records, and pointers. Automatic Program Verification V, *Stanford AIM-278* (1976).
8. NAKAJIMA, R., HONDA, M. and NAKAHARA, H. Hierarchical program specification and verification—a many-sorted logical approach—*Acta Informatica* (1980).

(Received January 23, 1981: revised August 17~1981)