Designing, Testing and Formally Proving a Ternary Tree Flattening Algorithm

ShengYu Shen

shengyushen@icloud.com

Abstract. This paper describes my effort on designing, testing and formally proving a ternary tree flatten algorithm. I first propose a language to describe ternary tree, and design a scanner and a parser to analyze this language and construct corresponding data structure in memory. I then describe my implementation of ternary tree flatten algorithm in C, together with a close loop testing framework that continuously generating tree flattening it and verifying the correctness of its result. Furthermore, I also propose a generator mode for my program that can write out large tree description file. Finally, I formally verify this algorithm with Coq theorem prover.

Keywords: ternary tree; flattening; formal verification

1 Introduction

I first define a language to describe ternary tree in Section 2. And then the ternary tree flatten algorithm is presented in Section 3. After that I will show how to test this algorithm in Section 4. Finally, I will present how to formally prove the correctness of this algorithm in Section 5.

In the tgz package, t3.c contains the implementation of flatten algorithm, while ssy.v is the Coq[2] proof of this algorithm. Just run "make" to generate the t3.exe program. And running it without any parameter can show you its usage.

Enjoy it.

2 A language describing ternary tree

We propose a language to facilitate the job of describing ternary tree. Its syntax is shown below in yacc[1] format:

```
1 tree :
2   T0
3   | ( TN tree data tree data tree )
4 ;
5 data :
6   D0
```

```
    | ( DN [0-9]+ )
;
```

In this code segment, T0 and D0 respectively mean an empty TreeNode and an empty DataNode, while (TN tree data tree data tree) means a TreeNode with subtrees, and (DN [0-9]+) means a DataNode with associated *value*.

So the ternary tree shown in Fig. 1 is described with the following code segment:

```
(TN
  (TN T0 (DN 1) T0 (DN 2) T0)
  (DN 3)
  (TN
    (TN T0 (DN 4) T0 D0 T0)
    (DN 5)
    T0
    (DN 6)
    (TN T0 (DN 7) T0 D0 T0)
  )
  (DN 8)
  (TN T0 (DN 9) T0 D0 T0)
)
```

3 Ternary tree flatten algorithm

3.1 Problem definition

A ternary tree is an extension of a binary tree, where instead of a left and a right sub-tree pointer and a data value with value between the left and the right subtree, there are left, center and right subtree pointers, with two data values, one between the left and the center subtree, and one between the center and the right subtree.

Your task is to flatten a ternary tree into an array in a depth-first manner. Such a tree is shown in Fig.1:

In this case, the output array should come out as follows:

1 2 3 4 5 6 7 8 9

Note that some of the values and subtrees may not be present, and this is represented by a NULL pointer. In this case, they should not be traversed or be included in the output.

The data type definitions appear below:

```
/* Define the data type */
typedef unsigned int value_t;
/* Data structures used to define the tree. */
typedef struct DataNode {
```

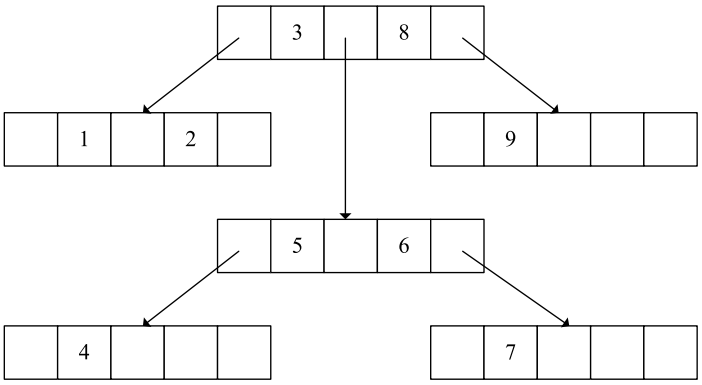

Fig. 1. Example of ternary tree

```
        value_t value;
} DataNode;

typedef struct TreeNode {
        struct TreeNode *left_tree;
        struct TreeNode *mid_tree;
        struct TreeNode *right_tree;
        struct DataNode *left_data;
        struct DataNode *right_data;
} TreeNode;

/* The interface to the function
flatten() that you're required to
 * implement is as follows: */
value_t * flatten (TreeNode * n, size_t * num_elements) ;
```

In this function: "TreeNode *n" is the pointer to the root node of the tree. "size_t num_elements" should be filled with the number of elements in the array on exit from the function. The return value "value_t *" should be a pointer to an array of elements containing the values of the flattened tree.

It is acceptable to solve this problem in either C or C++. Please describe any assumptions you've made, and the approach you've used to test your implementation.

3.2 The flatten algorithm

This algorithm is in the t3.c in the tgz package.

As shown intuitively in Fig. 2, at each *TreeNode*, my algorithm will recursively visit the three subtrees, and collect back their array of *value* items, and merge

them together with its own two *value* items to form a new array, which will finally be returned to its caller.

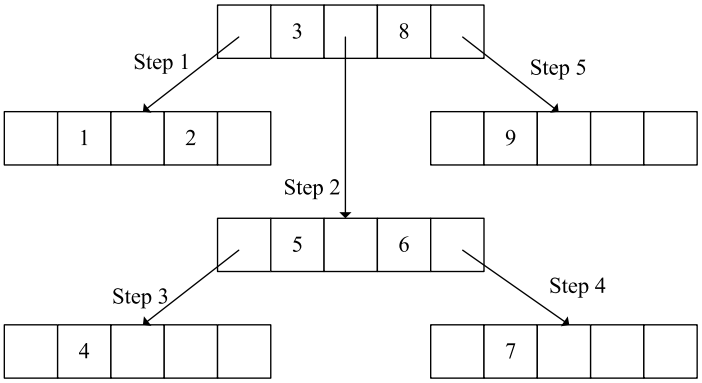

Fig. 2. Steps in recursively enumerate a ternary tree

The code that implement this idea is show below with embedded comments that explain my idea.

```
1  // copying a list of data from src to dst
2  // the size is num
3  void copyData (value_t *src, value_t *dst, size_t num) {
4    assert (num>=0);
5    int i;
6    for(i=0;i<num;i++) {
7      dst[i]=src[i];
8    }
9    return ;
10 }
11
12 value_t *flatten(TreeNode *n, size_t *num_elements)
13 {
14   //do nothing for an empty tree
15   if(n==(TreeNode*)NULL) {
16     *num_elements=0;
17     return (value_t*)NULL;
18   } else {
19     //recursively enumerate the three subtrees
20     //and get their list of data
21     size_t leftSize;
22     value_t *leftTreeArray=flatten(n->left_tree,&leftSize);
23     size_t midSize;
24     value_t *midTreeArray=flatten(n->mid_tree,&midSize);
```

```
25      size_t rightSize;
26      value_t *rightTreeArray=flatten(n->right_tree,&rightSize);
27
28      size_t leftDataSize;
29      if(n->left_data) {
30        leftDataSize = 1;
31      } else {
32        leftDataSize = 0;
33      }
34
35      size_t rightDataSize;
36      if(n->right_data) {
37        rightDataSize = 1;
38      } else {
39        rightDataSize = 0;
40      }
41
42      //the total size of new array
43      size_t allsize = leftSize + rightSize +
44      midSize + leftDataSize + rightDataSize;
45      *num_elements = allsize;
46
47      value_t *newArray = malloc(allsize*(sizeof(value_t)));
48      //copying the left tree
49      copyData(leftTreeArray,newArray,leftSize);
50      size_t newindex =leftSize;
51
52      //copying the left data and manipulate the new index
53      if(leftDataSize==1) {
54        newArray[newindex]=(n->left_data)->value;
55        newindex ++;
56      }
57
58      //copying the mid tree
59      copyData(midTreeArray,newArray+newindex,midSize);
60      newindex = newindex + midSize;
61
62      //copying the right data
63      if(rightDataSize ==1) {
64        newArray[newindex]=(n->right_data)->value;
65        newindex ++;
66      }
67
68      //copying the right tree
69      copyData(rightTreeArray,newArray+newindex,rightSize);
```

```
70
71      //free all mem alloced in flattening sub tree
72      free(leftTreeArray);
73      free(midTreeArray);
74      free(rightTreeArray);
75
76      return newArray;
77    }
78  }
```

4 Testing

4.1 Generating ternary tree automatically

Our tool *t3* have an additional mode that generate a large tree by:

```
1  Shell Prompt > t3 -gen <max depth> <density> > tree.txt
```

Here, <max depth> means the maximal depth of the generated tree, while <density> means the probability of generating a non-empty TreeNode or DataNode at each branch.

Of course, the user can also manually write a file according to the syntax described in Section 2.

We can flatten a ternary tree described in a particular file by running the following command in shell:

```
1  Shell Prompt > t3 <tree file name>
```

4.2 Close-loop testing framework

As shown intuitively in Fig. 3, my close-loop testing framework will continuously generate a ternary tree, put the list of *value* item been inserted in this tree into a list.

As shown in Fig. 4, After the flattening algorithm flattens the ternary tree, it can compare the resulting array with the list to confirm its result is correct.

This framework is called by running the following command in shell:

```
1  Shell Prompt > t3 -l
```

This framework will continuously increase the depth of the generated tree from 10, and run forever. The density for each generated tree is randomly determined before each iteration.

5 Formal proof of the flatten algorithm

This proof is in the ssy.v file in the tgz package.

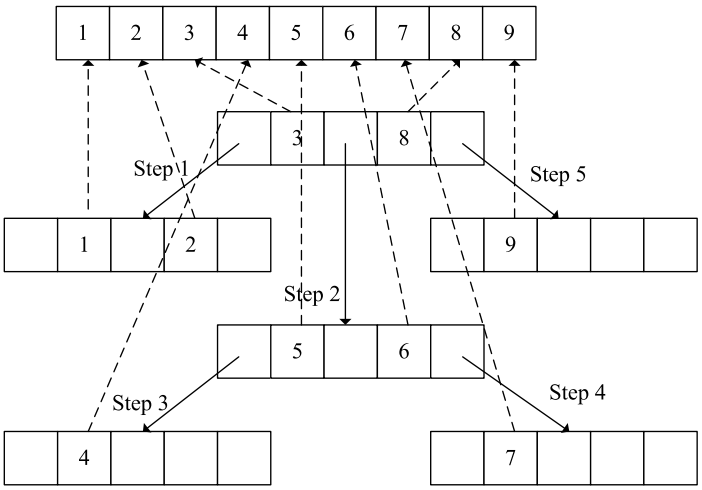

Fig. 3. Generating ternary tree with all its *value* item inserted into a list

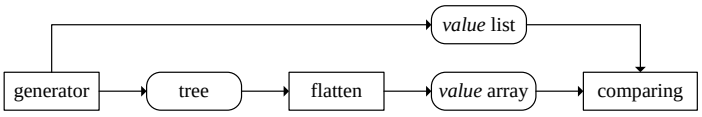

Fig. 4. Flattening a ternary tree and comparing the *value* array

5.1 subsecdef

I use the coq theorem prover[2] to prove the correctness of this algorithm. I will explain the proof step by step.

I first invoke all the related Coq library:

```
Require Import Arith.
Require Import List.
Require Import Bool.
Import ListNotations.
Open Scope nat_scope.
Open Scope list_scope.
```

I then define the two data structure DataNode and TreeNode:

```
(*definition of the two data structure of tenary tree*)
Inductive DataNode : Set :=
D0
| DN : nat -> DataNode.

Inductive TreeNode : Set :=
T0 : TreeNode
| TN : TreeNode->DataNode->TreeNode->
DataNode->TreeNode->TreeNode.
```

Then I define how to flatten a DataNode:

```
(*flattening the DataNode*)
Definition flatten_DataNode :=
fun dn:DataNode =>
  match dn with
  D0 => nil
  | DN n => cons n nil
  end.
```

I further define how to flatten a TreeNode, this is exactly the algorithm defined in Section3:

```
(*flattening the TreeNode*)
Fixpoint  flatten (tn:TreeNode) : list nat :=
  match tn with
  T0 => nil
  | TN lefttree leftdata midtree rightdata righttree =>
    let leftlist :=flatten lefttree in
    let midlist  :=flatten midtree in
    let rightlist:=flatten righttree in
    let leftdatalist := flatten_DataNode leftdata in
    let rightdatalist:= flatten_DataNode rightdata in
```

```
    leftlist++leftdatalist++midlist++
    rightdatalist++rightlist
  end.
```

At this point, you can take test this reformulated version on ternary tree by running:

```
Eval compute in flatten <tree>.
```

For example:

```
(*test drive on the flatten algorithm*)
Eval compute in flatten (TN T0 D0 T0 (DN 1) T0).
Eval compute in flatten
(TN
  (TN T0 (DN 1) T0 (DN 2) T0)
  (DN 3)
  (TN
    (TN T0 (DN 4) T0 D0 T0)
    (DN 5)
    T0
    (DN 6)
    (TN T0 (DN 7) T0 D0 T0)
  )
  (DN 8)
  (TN T0 (DN 9) T0 D0 T0)
).
```

I then define predicate stating that a value n appear in a DataNode dn:

```
(*a predicate meaning that n is in dn*)
Definition inDN (n:nat) (dn:DataNode) :Prop :=
  match dn with
  D0 => False
  | DN v => v=n
  end.
```

Similarly, I define a predicate stating that a value n appear in a TreeNode tn:

```
Fixpoint inTN (n:nat) (tn:TreeNode) :Prop :=
  match tn with
  T0 => False
  | TN lefttree leftdata midtree rightdata righttree =>
    (inTN n lefttree)\/
    (inDN n leftdata)\/
    (inTN n midtree)\/
```

```
    (inDN n rightdata)\/
    (inTN n righttree)
  end.
```

5.2 subsecproof

Next, I prove that the *inDN* operator is preserved by *flatten_DataNode*:

```
Lemma inDN_in_flattenDataNode :
  forall (n:nat) (dn:DataNode),
    (inDN n dn) -> (In n (flatten_DataNode dn)).
Proof.
intros.
induction dn.
elim H.
elim H.
simpl.
auto.
Qed.
```

Similarly, I prove that the *inTN* operator is preserved by *flatten*:

```
(*in operator is preserved by flatten
that is to say, some element in a
TreeNode is also in its flatten result*)
Theorem in_trans_flatten :
  forall (n:nat) (tn:TreeNode),
    (inTN n tn) -> (In n (flatten tn)).
Proof.
intros.
induction tn.
auto.
simpl in *.
repeat rewrite in_app_iff .
elim H.
intros.
left.
auto.
intros.
right.
elim H0.
intros.
left.
apply inDN_in_flattenDataNode.
auto.
intros.
```

```
right.
elim H1.
intros.
left.
apply IHtn2.
auto.
intros.
right.
elim H2.
intros.
left.
apply inDN_in_flattenDataNode.
auto.

intros.
right.
apply IHtn3.
auto.
Qed.
```

Again, I will prove the *inDN* operator is reversely preserved by *flatten_DataNode*:

```
Lemma in_flattenDataNode_inDN :
  forall (n:nat) (dn:DataNode),
    (In n (flatten_DataNode dn)) -> (inDN n dn).
Proof.
intros.
(*once again induction can expand dn on both hyp and goal
while case cab only expand goal*)
induction dn.
auto.
simpl.
elim H.
auto.
simpl.
intro.
elim H0.
Qed.
```

Similarly, I prove the *inTN* operator is reversely preserved by *flatten* with some lemmas:

```

Lemma in_3 :
  forall (a :nat) (l1 l2 l3: list nat),
    (In a (l1++l2++l3)) -> (In a l1)\/(In a l2)\/(In a l3).
Proof.
```

```
intros.
induction l1.
(*this is simply in all place*)
simpl in *.
right.
apply in_app_or.
assumption.
elim H.
intros.
left.
rewrite H0.
apply in_eq.
intros.
(*very useful in using hyps IHl0 with "A-> B" and H0 with "A"*)
destruct (IHl1 H0).
left.
apply in_cons.
assumption.
right.
assumption.
Qed.

Lemma in_4 :
  forall (a :nat) (l1 l2 l3 l4: list nat),
    (In a (l1++l2++l3++l4)) -> (In a l1)\/(In a l2)\/(In a l3)\/(In a l4).
Proof.
intros.
induction l1.
simpl in *.
right.
apply in_3.
assumption.
elim H.
intros.
left.
rewrite H0.
apply in_eq.
intros.
destruct (IHl1 H0).
left.
apply in_cons.
auto.
right.
auto.
Qed.
```

```
51
52
53 Lemma in_5 :
54   forall (a :nat) (l1 l2 l3 l4 l5: list nat),
55     (In a (l1++l2++l3++l4++l5)) -> (In a l1)\/(In a l2)\/(In a l3)\/(In a l4)
56 Proof.
57 intros.
58 induction l1.
59 simpl in *.
60 right.
61 apply in_4.
62 auto.
63 elim H.
64 intros.
65 left.
66 rewrite H0.
67 apply in_eq.
68 intros.
69 destruct (IHl1 H0).
70 left.
71 apply in_cons.
72 auto.
73 right.
74 auto.
75 Qed.
76
77 Theorem in_trans_flatten_rev :
78   forall (n:nat) (tn:TreeNode),
79     (In n (flatten tn)) -> (inTN n tn).
80 Proof.
81 intros.
82 induction tn.
83 auto.
84 simpl in *.
85 destruct ((in_5 n (flatten tn1) (flatten_DataNode d) (flatten tn2) (flatten_Da
   )H).
86 left.
87 apply IHtn1.
88 auto.
89 right.
90 (*another powerful trick that break A \/B\/... into A and B\/...*)
91 induction H0.
92 left.
93 apply in_flattenDataNode_inDN .
94 auto.
```

```
elim H0.
intros.
right.
left.
apply IHtn2.
auto.
intro.
right.
right.
elim H1.
intros.
left.
apply in_flattenDataNode_inDN .
auto.
intros.
right.
apply IHtn3.
auto.
Qed.
```

With Theorems *in_trans_flatten* and *in_trans_flatten_rev* presented above, we can be sure that *flatten* algorithm does not change the set of *value*.

6 REFERENCES

References

1. Johnson, Stephen C. (1975). "Yacc: Yet Another Compiler-Compiler". AT&T Bell Laboratories Technical Reports (AT&T Bell Laboratories Murray Hill, New Jersey 07974) (32).
2. Yves Bertot, Pierre Castran , "Interactive Theorem Proving and Program Development CoqArt: The Calculus of Inductive Constructions," in Texts in Theoretical Computer Science An EATCS Series,Springer 2004.