AVL Trees

Cláudio Amaral  Mário Florido  Vítor Santos Costa
{coa,amf,vsc}@dcc.fc.up.pt

AVL trees are implemented in a Yap module, avl.yap, making it interesting to show black-box testing with PrologCheck. Its interface is small, with predicates to create an empty tree, insert an element and look up an element, respectively avl_new/1, avl_insert/4 and avl_lookup/3. When performing this kind of test one does not simply test individual predicates, we test usages of the module. To do this we must be able to create sequences of interface calls and inspect intermediate results. Knowledge about the shape of data is gathered by manual testing if it is not known.

Generator

Creating a valid sequence of interface calls is not difficult, but requires attention to detail. First, we only want to generate valid sequences to save effort of checking validity and not suffer from sparse valid values. Usages of the module have two sets of important terms: key terms, and value terms. We decided to parametrise these sets as generators. We also parametrise a generator for failed look-ups that must be disjoint from the generator of values. In this generator we start by creating the tree, independently of the size parameter. This implies that when size is 0 an empty AVL-tree is still created. Thus, we always append the tree creation to a sequence of calls to insert and look-up values.

```
genAvl(KeyGen, ValGen, FGen, Calls, Size) :-
    Calls = [[avl:avl_new] || Cs1],
genAvlCmds(KeyGen, ValGen, FGen, [()], Cs1, Size).

genAvlCmds(KeyGen, ValGen, FGen, Lookups, Cs, Size) :-
genCmdHead(KeyGen, ValGen, FGen, Lookups, Cs, Size),
call(KeyGen, Key, Size),
    (X = i, Val),
    Cs = [[i, avl:avl_insert(Key, Val)] || CsS],
    NewLookups = [ (Key, Val) || Lookups ] ;
```

Choosing a single command follows a couple of restrictions. If the generator for failing look-ups is missing, these commands will not be generated. If the list of look-ups is empty, valid look-ups will not be generated. Valid look-ups can take two forms: checking if the key-value is in the tree and asking for the value corresponding to some key. All commands are branded by a command identifier, making it possible to distinguish between different kinds of look-up. The possible choices are bundled in a frequency generator. The relative probabilities are such that we get a big variety of commands within relatively small sequences.

Together with the generators for module usages we need the generators for keys and values. Keys are integers between 0 and 50000, a large enough interval to generate a fair amount of unique keys in smaller test cases. Values are non-negative integers bound by the size. To construct a disjoint generator we use singleton lists with elements given by the value generator.
Property

The avl property depends on several factors. It is parametrised by the tree to check and by operations to extract information from that tree. These operations consist of: current node’s key; key comparison; left and right sub-trees; empty tree test. A side effect of the property is returning the height of the tree. The empty tree test is a callable term that checks if the tree in the first argument is empty and unifies the second argument to the appropriate truth value. Key comparison is callable with three parameters: first key, second key and result. The result of the comparison reflects the relation between the given keys: lt, gt, eq, lte, gte.

```
pcprop({avl, (tree, T), (height, H),
        (curr_key, GetKey), (cmp_key, CmpKeys),
        (left, L), (right, R),
        (is_nil, IsNil)}) :-
  pcif(...).
```

A tree may be empty, in which case we just unify the correct height. In the case of a non-empty tree we retrieve its key and sub-trees. They are used in recursive checks of the property. The recursive calls are for a different predicate, accumulating lists of keys that should be greater of less then the keys of the tree. After checking the sub-trees, the returned heights are compared for balance and the current tree height computed.

```
pcif ( (call(IsNil, T, V), V = false)
    , ((call(GetKey, T, Key),
       call(L, T, LT), call(R, T, RT))
      pc_and
      pcprop({avl, (gt, [Key]), (lt, []),
              (tree, LT), (height, LH),
              (curr_key, GetKey), (cmp_key, CmpKeys),
              (left, L), (right, R), (is_nil, IsNil)})
      pc_and
      pcprop({avl, (gt, []), (lt, [Key]),
              (tree, RT), (height, RH),
              (curr_key, GetKey), (cmp_key, CmpKeys),
              (left, L), (right, R), (is_nil, IsNil)})
      pc_and
      ((abs(LH-RH) =< 1, !
        ; print(height_mismatch), nl, fail),
       H is 1+ max(LH, RH))
    ),
  (V=false, H = 0) )
```

The accumulating version of the property receives as extra arguments the lists of keys in the path up to the root that are greater and less then the current key. This order criterion is what differentiates the accumulating property. It is checked for the current key which is then added to the appropriate accumulators in the recursive calls.

```
pccprop({avl, (gt, GTS), (lt, LTS), ...}) :-
  pcif(
    (...),
    (call(GetKey, T, Key), call(L, T, LT),
     call(R, T, RT),
     (forall(member(X, LTS),
               call(CmpKeys, X, Key, lte)), !
      ; print(avl_key_not_inorder1), nl, fail),
     (forall(member(X, GTS),
               call(CmpKeys, X, Key, gt)), !
      ; print(avl_key_not_inorder2), nl, fail))
  pc_and (...)
  pc_and (...)
  pc_and (...)
  ),
```
To access the fields of trees generated and maintained by avl.yap we examine the product of some calls. After inspecting the resulting trees we implement the operations needed for the generic property. We complement this with the key comparison and empty tree check.

```prolog
get_key( avl( _Left , Key , _Val , _Balance , _Right ) , Key ).
cmp_keys( K1 , K2 , Cmp ) :-
   print({K1 , K2}),
   ( K1 < K2 , Cmp = lt
    ; K1 =< K2 , Cmp = lte
    ; K1 =:= K2 , Cmp = eq
    ; K1 >= K2 , Cmp = gte
    ; K1 > K2 , Cmp = gt ).
left( avl( Left , _Key , _Val , _Balance , _Right ), Left ).
right( avl( _Left , _Key , _Val , _Balance , Right ), Right ).
is_nil( avl( _Left , _Key , _Val , _Balance , _Right ) , false ).
is_nil([], true ).
```

With the generator, generic property, and access predicates we proceed to define the main test. We quantify over module uses, which are processed in its body. The body is composed by a start and a loop, dealing respectively with tree creation and insert/look-up commands. The start consumes the tree creation command and checks if the generated tree is an empty avl tree. The resulting empty tree is fed to the loop with any remaining interface calls. The loop consumes the command list a command at a time. An insert command requires that the resulting tree be checked for the avl property. Looking-up a key-value depends on the given label. The labels 11 and 12 indicate valid look-ups. In 11 the key-value look-up is performed directly, in contrast with 12 which looks-up the key and matches the returning value against the expected value. Invalid look-ups are validated by their execution failure.

```prolog
pcprop( avlUses ) :-
   pcforall( avlTest : genAvl ( avlTest : genKey , avlTest : genVal , avlTest : genFVal ), Calls , pcprop({ avlUses , Calls })).
```

```prolog
pcprop({ avlUses , [(avl:avl_new)||Calls]}) :-
call( avl:avl_new(Tree))
pc_and
   pcif(
      pcprop(({avl , (tree , Tree ), ( height , 0) ,
                (curr_key , avlTest:get_key) ,
                (cmp_key , avlTest:cmp_keys) ,
                (left , avlTest:left) ,
                (right , avlTest:right) ,
                (is_nil , avlTest:is_nil)})
      ,(pcprop({avlUses , Tree , Calls})))
   ,(print(avl_new_fail_invariant), nl, fail)).
```

```prolog
pcprop({avlUses , Tree , []}).
```

```prolog
pcprop({avlUses , Tree , [(i,avl:avl_insert(Key,Val))||Calls]}) :-
call( avl:avl_insert(Key,Val), Tree , ContinuationTree)
pc_and
   pcif(
      pcprop((avl, (tree, ContinuationTree), (...) )))
   ,(pcprop({avlUses , ContinuationTree , Calls})))
```
Note that the labelled property is connected with a `pc_and` operator where instead it could be in the place of the `true` Prolog built-in goal. This is due to the fact that PrologCheck does not inspect Prolog goals and connectives.

It is to be expected that the module satisfy the properties. We run 1000 tests with the test number option. In one of the testing suits, a counterexample popped up. We repeat the tests until we get enough counterexamples to examine. Inspecting the counterexamples we find the reason for the failure. We discover that the counterexamples found have two insertions under the same key, where the second value is discarded, and a look-up based on the second insertion. This means that the key generator is not sparse enough for 1000 test, where the size of generated values grows and the probability of such a combination appearing starts to show. We decide to do nothing about it since it is a very uncommon event. We could have increased the interval for the generation of keys, which would be a quick fix yet would not fix the specification bug, instead it would just make it less common. The best way to fix the specification is to not adding existing keys to the look-ups list in the command generator or enforcing the picking of the first insertion of a certain key. The latter is possible by choosing the last entry in the look-ups list with the key in question.

In order to assess the tool and the specification in broader terms we insert two bugs and test the two erroneous versions of the module. The first bug was introduced by the tester and the second by another programmer. This still fits the goal of black-box testing because the actual tests are still performed without knowledge of the module’s source. It is the same as testing different closed implementations.
To explore the tool potential we inserted a bug. We changed a line in the table that states how the tree will be re-balanced upon insertion of a new element. The first arguments of the table are input. The first is the state of the balance, and the second where we insert the new element. The other three columns state if the new state of the tree is balanced, if the tree increased in height, and if the tree needs further re-balancing. For example, the two first rows describe a situation where a totally balanced tree receives a new element. The tree will increase in size and will not be balanced, but we cannot improve on that, so no re-balancing will be needed.

```
% table (- , left , - , yes , no ).
table( - , left , - , yes , no ).
table( - , right , > , yes , no ).
table( < , left , - , no , yes ).
table( < , right , - , no , no ).
table( > , left , - , no , no ).
table( > , right , - , no , yes ).
```

Testing this version of the implementation shows a counterexample after less than 100 test cases on average. The counterexamples found are command sequences where three inserts are given in the key order 3, 1, 2. What happens is that after inserting the second value, the tree is unbalanced. This information is erased by the bug, which will allow any number of left insertions. Another failing sequence of insertions is in the key order 1, 3, 2. Here the last call will not even give a wrong result, it will simply fail at the last insertion. This is noticed in this example by the lack of an error message from the property. The only conclusion we can get from the counterexamples is that the balancing of the tree is probably failing when the tree is unbalanced to the right and the sub-tree to the left.

```
?- plqc:prologcheck(
    avlTest:(pcprop(avlUses))).
Failed: After 51 test(s).
Counterexample found:
    [[avl:avl_new,
      {i,avl:avl_insert(22340,0)},
      {i,avl:avl_insert(1488,1)},
      {i,avl:avl_insert(8952,0)}]],
    yes
?- plqc:prologcheck(
    avlTest:(pcprop(avlUses))).
Failed: After 66 test(s).
Counterexample found:
    [[avl:avl_new,
      {i,avl:avl_insert(9919,1)},
      {i,avl:avl_insert(39796,0)},
      {i,avl:avl_insert(18564,1)}]],
    yes
```

The second bug is purposely inserted by someone that is not involved with the tests. We examine and write about it as such, starting with the tests. Counterexamples are found in average after at least 200 test cases. The minimal shape found are sequences of 5 insertions. There are two families of counterexamples. One is such that the first three elements inserted are the keys 1, 2, 4, in any order, followed by key 5 and finally key 3. In the second the first keys are 2, 4, 5, also in no particular order, followed by 3 and 1.

```
?- plqc:prologcheck(
    avlTest:(pcprop(avlUses)),
    [numtests, 10000]).
Failed: After 213 test(s).
```
Counterexample found:
[[avl:avl_new,
 {i,avl:avl_insert(42290,9)},
 {i,avl:avl_insert(17481,5)},
 {i,avl:avl_insert(19130,5)},
 {i,avl:avl_insert(18001,3)},
 {i,avl:avl_insert(4284,1)}]]

yes

These sequences fail when trying to complete the second level of one of the root’s sub-trees. Furthermore, it happens only if the completion takes the order right-left.

There are no other details we could retrieve information from. As far as black-box testing is concerned the origin of the error cannot be pinpointed. Although, we can infer we are dealing with a balance issue, since it can appear with only insertions. We can go as far as saying that the problem appears to be in a left insertion, not noticeable in the nodes after the root of the tree.

Inspecting the source we can see that the error was caused by another change to the balance table changed for the first error. In this case, the following change was performed.

table( > , left , - , yes , no ).
% table( < , left , - , no , no ).

This forces a re-balance of an already balanced tree in the speculated situation - an insertion to the left filling where there is already a value to the right.