Standard front page

IT-Universitetet i København

Course exam with written work

Class code (Holdkode):
SASP
1007008U

Name of course:
Advanced Models and Programs

Course manager:
Jesper Bengtson

Course with e-portfolio, link:

Thesis/project exam with project agreement

Supervisor:

Thesis or project title:

Name(s):	Birthdate and year:	ITU-mail:
1. Hildur Uffe Flemberg	27-06-89	hufl@
2. Nicolai Skovvart	23-12-88	nbsk@
3. Søren Sønderby Nielsen	23-10-90	ssni@
4.		@
5.		@
6.		@
7.		@

Verification of AVL trees in Gallina

Hildur Uffe Flemberg, Nicolai Bo Skovvart and Søren Sønderby Nielsen
{hufl, nbsk, ssni}@itu.dk

IT University of Copenhagen

Abstract. Formal verification is useful to prove various properties of programs and is preferable to other validation methods such as testing or design by contract if correctness is important. It is however much more costly and time-consuming to prove that a program actually works as intended than making the program to begin with. However, efforts are being made to make verification of software easier, and the Coq interactive proof assistant is one such attempt. In this paper we implement AVL trees and prove various properties of our implementation using Coq in the Gallina language. Examples of things we prove include that our binary search tree add-method does not break the binary search tree invariant, that our AVL tree balancing function does not alter the in-order traversal of the tree and that our AVL tree insert-method does not break the binary search tree invariant.
The source-code for this project can be found at https://github.com/ssnielsen/SASP-AVL.

Keywords: AVL Trees, Formal Verification, Coq Interactive Proof Assistant, Gallina

1 Introduction

This project is conducted in the spring term 2013 at the IT University of Copenhagen in connection with the Advanced Model and Programs course. The project is supervised by Jesper Bengtson and Hannes Mehnert. The target audience of the report is people interested in Coq [2] and the field of formal verification in general.

Software verification is a widespread research area due to its ability to prove properties of programs and algorithms, unlike for example testing that can only verify that the programs and algorithms work as expected in certain cases. However, formally proving that software adheres to the specification is traditionally a time consuming task compared to testing, costs a lot of money and is therefore neglected in favor of the latter methodology. For that reason, we wish to promote the use of interactive tools like proof assistants to facilitate the process of verification.

During the project, we developed a small-scale piece of software in Gallina [3], the formal specification language of Coq, for verifying the correctness of basic operations on AVL trees, e.g. that inserting an element in a valid AVL tree does not destroy its properties as a valid tree etc.

In this paper, we will first cover the background of the project including Coq and AVL trees in particular, followed by an example of insertion in AVL trees. We then present our implementation and what aspects of it we have formally verified. Following that, we will reflect upon the project and discuss relevant future work. Finally, we come to a conclusion on the project.

2 Background

2.1 Coq

Coq is an interactive proof assistant. It can be seen as a combination of two things:

1. A simple functional specification language named Gallina
2. A set of tools for stating and proving logical assertions (including assertions about the behavior of programs).

It allows one to express propositions and then prove them step-by-step. It can do this using various forms of logic, for example propositional logic, boolean logic, separation logic etc. which we will not go into detail with in this report. In this project we have primarily been using propositional and boolean logic.

For example, consider the following proposition: *If a list appended to another list results in the empty list, both lists must be empty.* In standard logic this could be expressed as $\forall l1\, l2, [] = l1 ++ l2 \rightarrow l1 = [] \wedge l2 = []$. In Coq, the expression could look like Code Sample 1:

```
Lemma empty_app : forall (l1: list nat) (l2 : list nat),
    [] = l1 ++ l2 -> l1 = [] /\ l2 = [].
```

Code Sample 1: Coq example: Empty append - proposition

First of all, the lemma we are trying to prove is given a name, in this case empty_app. This makes us able to reference the proof in subsequent proofs. We then state, using the *forall* keyword that the following lemma talks about two arbitrary lists and name the lists *l1* and *l2*. A pre-condition is added using the implies operator stating that the lemma only holds for empty input lists. Given this precondition we are ready to state something about the input lists, namely that *l1* is the empty list and (written as /\ in Coq resembling $\wedge$ in classical logic) *l2* is the empty list.

Up to this point we have only specified what we want to prove, not how to actually prove it. Coq requires us to be very specific when proving propositions, but it does allow earlier proofs to be referenced. To prove our example, we specify the following set of tactics:

```
Proof.
   intros.
   split.
      destruct l1.
         reflexivity.
         inversion H.
      destruct l2.
         reflexivity.
         destruct l1.
            inversion H.
            inversion H.
Qed.
```

Code Sample 2: Coq example: Empty append - proof

First, the precondition is moved into the context using intros (introduction). We are then left with proving $l1 = [] \wedge l2 = []$. We now have to prove two cases, namely that $l1$ is the empty list and that $l2$ is the empty list. To prove that $l1$ is the empty-list we perform case analysis on $l1$. A list can either be the empty list ([]) or the cons-case ($a :: l1$). In the empty list case, we are left with $[] = []$ which is obviously true and can be solved with the reflexivity tactic. The cons case cannot be proven, but it can be dismissed as it breaks our precondition. The inversion tactic is smart enough to identify this broken preconception. We have now proven $l1 = []$. We are then left with proving $l2 = []$. We also perform cases analysis in this case, and can disregard the empty list case in the same way as $l1$. We then have to disregard the cons-case, but Coq is initially not smart enough to solve this using the inversion tactic as it does not know anything about the left-hand side of $[] = l1 ++ a :: l2$, even if this seems obviously false. The solution to this is to perform case-analysis on $l1$ again. When $l1$ is the empty list, we are left with $[] = a :: l2$ which inversion can disregarded. When $l1$ is the cons-case we are left with $[] = a :: l1 ++ a :: l2$, which can also be disregard by inversion.

We have now proven in Coq that if two lists appended to each other results in the empty list, both lists must be the empty list. □

2.2 AVL trees

An AVL tree is a variant of a binary search tree [1] where every operation performed on the tree that modifies its state is terminated with a re-balancing operation [1].

An AVL tree is balanced when its root and all sub-trees are balanced and a node is balanced whenever the difference between the heights of its sub-trees are at most 1.

Rebalancing of a tree T is done by rotating sub-trees according to well defined rules depending on where in T the insertion or deletion took place. Both insertion

and deletion works by making sure that no unbalances exist on the path from the inserted/deleted element w to the root of the tree.

If some node z on the path from w to the root of T is out of balance, this unbalance can be fixed in two different, but almost similar ways depending on the type of operation that caused it.

Insertion In case of an insertion, we fix the unbalance locally in the sub-tree rooted at z using a rotation over a trinode consisting of z, a child of z and a grandchild of z. Let y be the taller child of z and let x be the taller child of y. If the children of y are equally tall, pick the child that is also an ancestor of w.

The tree rooted at z can now be rebalanced using the algorithm shown in Figure 1. The structural relationship between nodes x, y, z and their sub-trees can be laid out in 4 different ways all illustrated in Figures 1 to 4. Depending on the layout, the type of rotation needed to balance the tree are often labelled as Left-Left, Left-Right, Right-Right, Right-Left or simply LL, LR, RR and LR in short.

Since the element was inserted in the sub-tree rooted at z and since the balance of z is now restored, all ancestors of z that were previously unbalanced are now balanced. Thus it takes only one rotation (single or double) to restore global balance of an AVL tree after insertion.

Deletion The main difference between insertion and deletion lies in the way x and y are chosen after z is found - more specifically, how ties are broken when identifying x. If the children of y are equally tall, x is chosen so that it has the same parental relationship to y as y has to z. For example, if y is a left child of z, x is the left child of y and if y is a right child of z, x is a right child of y.

Finally, due to the different strategy for choosing x, we might reduce the height of the sub-tree by 1, thus risking to cause unbalances further up in the tree. To make up for this, we need to traverse the rest of the path from z to the root of the tree and maybe do repeating rotations to restore global balance.

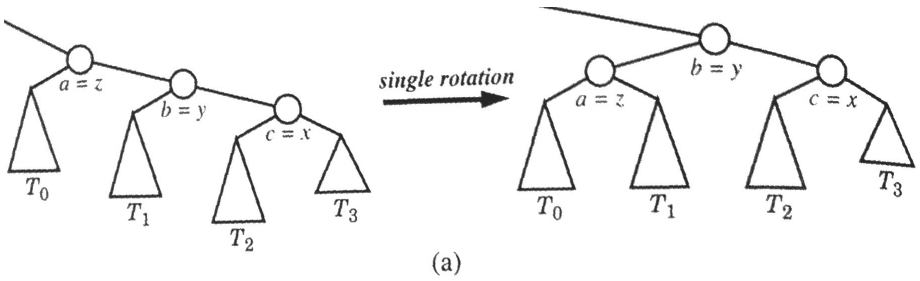
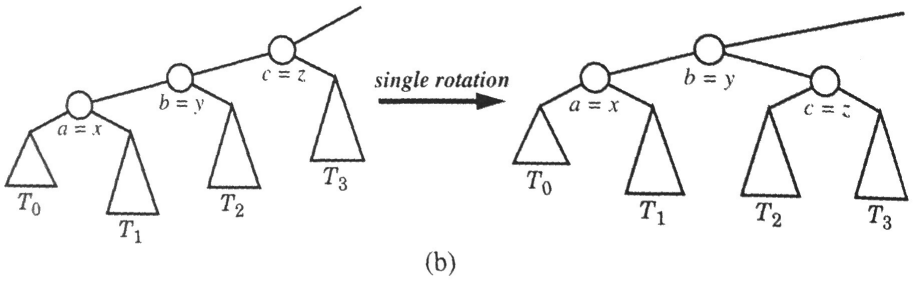
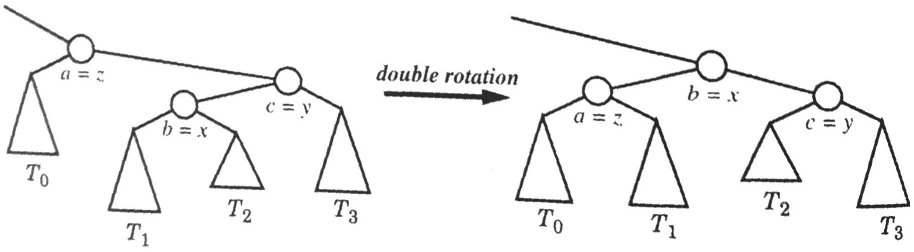


Fig. 1: RR-rotation [1]



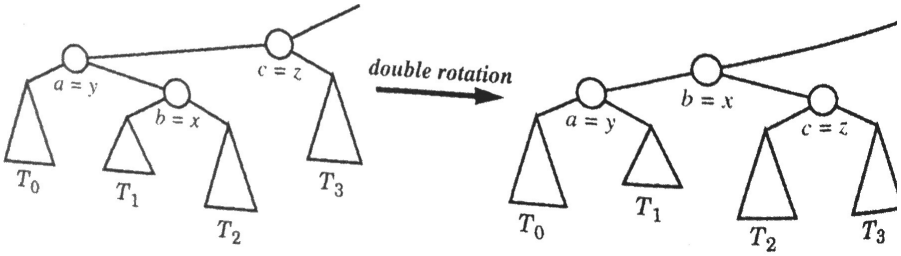
(b)

Fig. 2: LL-rotation [1]



(c)

Fig. 3: RL-rotation [1]



(d)

Fig. 4: LR-rotation [1]

2.3 AVL Insertion Example

As an example of how AVL trees operate, we show how to insert the element 54 in an existing AVL tree. Initially, the element is inserted as it would be in a regular binary search tree - that is, we compare the new element to the root, continue in the left sub-tree if the element is less-than or equal to the root value and in the right sub-tree otherwise (For our Coq-implementation of add see Code Sample 5). When a leaf is reached, it is replaced with a node with the value of 54 and leaves as its sub-trees. Figure 5.a shows the state of the tree after this step with heights in annotations above each node.

Next, we check the balance of the root checked. The root is out of balance as the difference in heights between the sub-trees exceeds 1. Let the 78-node be identified as z. The taller child of z is the one storing 50, let this be identified as y. Likewise, 62 is the taller child of y, let this be identified as x. The relationship between x, y and z is recognized as the case shown in Figure 4, suggesting that we fix the unbalance by a double rotation.

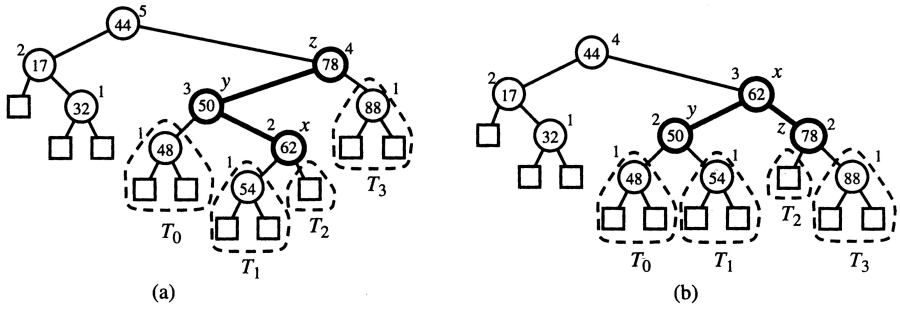


Fig. 5: Insertion in an AVL tree. Fig (a) shows the tree after 54 is inserted as it would be in a regular binary search tree. As can be seen, the insertion leaves the tree out of balance. Fig. (b) shows the same tree after the balancing operation [1]

Figure 5.b shows the state of the tree after the rotation where 62 has elevated its left sub-tree into its place and replaced the tree rooted at 78. The tree, previously rooted at 78, becomes the new right sub-tree of 62 while the old right sub-tree of 62 becomes the new left sub-tree of 78. As can be seen on the figure, this single operation restores global balance.

3 Solution

3.1 Implementation

To be able to prove the correctness of the AVL tree operations, we have modelled trees using Gallina which is the specification language in Coq [3]. The implementation consists of several functions and inductively defined types.

Types

Tree The entire implementation builds on the `tree` type (Code Sample 3). The type has two constructors, the leaf and the node. In our implementation, the leaves are empty (sub-)trees, and the node consists of a natural number (the value of the node) as well as the left and right sub-trees (which can either be leaves or nodes).

```
Inductive tree: Type :=
   | leaf : tree
   | node : nat -> tree -> tree -> tree.
```

Code Sample 3: Tree

Functions

Height One of things that make operations on AVL trees efficient is that the tree is balanced. Since balance is defined in terms of the height of the sub-trees it is valuable to have a function that defines the height of a tree. Our implementation is shown in Code Sample 4 and states that leaves have a height of zero while nodes have a height of 1 plus the height of the tallest sub-tree.

```
Fixpoint height (t: tree) :=
    match t with
    | leaf => 0
    | node _ leftChild rightChild => 1 + max (
       height leftChild) (height rightChild)
    end.
```

Code Sample 4: Fixpoint to compute the height of a tree

Add The `add` fixpoint is displayed in Code Sample 5. It takes a value, a tree to insert it in and returns the updated binary search tree.

```
Fixpoint add (val:nat) (t: tree) :=
    match t with
    | leaf => node val leaf leaf
    | node treeval lChild rChild =>
            match ble_nat val treeval with
            | true => node treeval (add val lChild)
               rChild
            | false => node treeval lChild (add val
               rChild)
            end
    end.
```

Code Sample 5: Fixpoint that performs a binary search tree add operation

Balance The `balance` fixpoint takes an unbalanced tree and returns a tree where one balancing rotation has been performed. The types of rotation are shown in Figures 1 to 4. Balancing is a key property of AVL trees, because it is the balancing operation(s) following the modification to the tree that ensure the height of the tree does not exceed $O(logn)$ where n is the amount of elements in the tree.

A balanced tree has a higher performance because it allows look-up of elements in $O(logn)$ time which is the dominating factor in all operations that modifies the tree. This is evident because an element has to be found before it can be changed. Likewise, when inserting a new element, the proper place to insert it has to be searched for just like if the tree had been a regular binary search tree.

Note that when we, in line five, match on the difference in heights of the left and right sub-tree, we add two. This makes sure that the result is always a natural number instead of an integer in the case where the right sub-tree is the tallest. As a result of this, we match the result of the expression with zero and four instead of minus two and two. The comments next to the match cases explain this behavior as well.

The implementation of the `balance` fixpoint is too long to show in its entirety, but can be found in Appendix A. A snippet is shown below in Code Sample 6.

```
Fixpoint balance (t : tree) :=
   match t with
   | leaf => leaf (* Leaves are balanced by definition *)
   | node val lChild rChild =>
      match (height lChild) + 2 - (height rChild) with
      | 0 => (* -2, right sub-tree is unbalanced *)
```

```
match rChild with
| leaf => leaf (* shouldn't happen *)
| node rval rlChild rrChild =>
   match blt_nat (height rlChild) (height
       rrChild) with
   | true => (* RR-rotation *)
      node rval
         (node val lChild rlChild)
         rrChild
   | false =>
      match rlChild with
      | leaf => leaf (* shouldn't happen*)
      | node rlval rllChild rlrChild =>
         (* RL-rotation *)
         node rlval
            (node val lChild rllChild)
            (node rval rlrChild rrChild)
      end
   end
end
...
```

Code Sample 6: Snippet of balance fixpoint that does one balancing rotation on a tree

Insert The `insert` operation takes a tree and an element to be inserted. The operation is implemented as a simple *Definition* in Gallina which adds the element to the tree using the add fixpoint and balances the result with the balance fixpoint.

```
Definition insert val t :=
    balance (add val t).
```

Code Sample 7: Definition of the meaning of insertion

3.2 Validity

To show that our implementation of AVL trees is valid we want to prove various properties about the behavior of the tree. To be able to express some of these properties, we rely on some helper functions, namely in-order tree traversal and what it means to be a sorted list.

Helper functions

In-order The `inorder` fixpoint takes a tree and computes a list containing the tree elements in the order they would be encountered in the tree during an in-order traversal [1]. This means that the root element of the tree lies between the elements from its left sub-tree and the elements from its right sub-tree in the list. The same property applies for all other nodes in the tree.

A consequence of the way the list is built is that the elements will appear in the list in sorted order, from smallest to largest, when the original tree is a binary search tree. This is immediate because all elements in the left sub-tree of a binary search tree node is exactly those that are smaller than or equal than the element at the node itself. This property comes in handy in the rest of the implementation where we use it to check if a random tree is a binary search tree. In other words, we make an in-order traversal of the tree and check if the resulting list is sorted.

Definition 1. *The binary search tree invariant can be expressed as the in-order tree traversal being sorted.*

The `inorder` fixpoint is shown in Code Sample 8.

```
Fixpoint inorder (t: tree) :=
    match t with
    | leaf => []
    | node val leftChild rightChild => (inorder
       leftChild)++val::(inorder rightChild)
    end.
```

Code Sample 8: Fixpoint that computes the in-order traversal of a tree

Sorted list For a list to be sorted, it either has to be empty, a single-element list or begin with an element x that is less-than or equal to the following element y followed by a list that is sorted when y is included as the head of the list. Our Coq implementation expressing this can be seen in Code Sample 9.

```
Inductive sorted : list nat -> Prop :=
   | s_nil : sorted []
   | s_single : forall n, sorted [n]
   | s_cons : forall n m xs, n <= m -> sorted (m::xs) ->
      sorted (n::m::xs).
```

Code Sample 9: Behavior of a sorted list

Proofs The properties of AVL trees we have proven are the following:

Add preserves binary search tree invariant We wanted to prove that our add function did not break the binary search tree invariant, namely that elements smaller than or equal to a node was contained in the left sub-tree, and elements larger than the node was contained in the right sub-tree. The binary search tree invariant can conveniently be expressed as the fact that the in-order traversal of the tree is a sorted list.

Our equivalent Coq expression can be seen in Code Sample 10.

```
Theorem add_preserves_bst : forall t n,
    sorted (inorder t) -> sorted (inorder (add n t)).
```

Code Sample 10: Add preserves binary search tree invariant

Non-empty after add An important thing to prove about the add function is that it always produce non-empty trees. This expression is seen in Code Sample 11.

```
Lemma inorder_add_nonempty : forall n t,
    inorder (add n t) <> [].
```

Code Sample 11: Adding an element to a tree results in a non-empty tree

Balance does not alter in-order traversal We also wanted to prove that our balance function did not change the in-order traversal of a tree. This is a theorem that was essential to proving the insert preserves binary search tree invariant. Our Coq expression can be seen in Code Sample 12

```
Theorem balance_inorder : forall t,
    inorder t = inorder (balance t).
```

Code Sample 12: Balance does not alter in-order traversal

Insert preserves binary search tree invariant The most essential proof we have done is that our insert method does not break the binary search tree invariant. Due to the formerly proven theorems it is fairly short, and can be seen in its entirety in Code Sample 13.

```
Theorem insert_preserves_bst : forall t n,
    sorted (inorder t) -> sorted (inorder (insert n t)).
Proof.
    intros. unfold insert. rewrite <- balance_inorder.
        apply add_preserves_bst. assumption.
Qed.
```

Code Sample 13: Insert preserves binary search tree invariant

4 Reflection and Future Work

4.1 Reflection

Formally proving the properties of our AVL tree implementation has been both interesting and challenging.

We initially assumed we would be able to prove more aspects of AVL trees, but fell into many pitfalls and overlooked some smart tricks that could reduce the complexity of proofs significantly. We also got to experience times where the things we tried to prove were false which as an example led to the discovery of a false statement in the balance fixpoint. This discovery was made when we got to a point where we were unable to prove it in Coq.

This is a good example of why an interactive theorem prover is an incredibly useful tool, as false assumptions can easily be overlooked when writing proofs by hand. It also shows the validity of formal verification, as it helped us spot our invalid implementation of the balance function even though we had already tested it on concrete example trees in Coq. As written by Edsger Dijkstra, 1969: *"Testing shows the presence, not the absence of bugs"*.

4.2 Future Work

In this project, we verified a couple of operations on AVL trees in Gallina, but there are still more things that could both be implemented and proven due to time limitations. We present the following suggestions for future work.

4.3 Prove that balancing has the intended effect

The purpose of the balancing is to prove that after insertion, the height of the tree is at most $log_2 n$ where n is the number of elements in the tree. Formally, the proof could look something along the lines of what is presented in Code Sample 14.

```
Theorem balance_property, forall n t t',
    balanced_tree(t) ->
    t' = insert n t ->
```

```
balanced_tree(t') /\
height t' <= log2 (list_length (inorder t')).
```

Code Sample 14: Proof that balancing has the intended effect

For this to be proven, a *balanced_tree* inductively defined data type would have to be implemented, checking that the difference in sub-tree height is never greater than 1 for all nodes in the tree. The *balanced_tree*(t') could also potentially be proven in a separate theorem. Due to time constraints we were unable to do this, but as it is a key property of AVL trees it should be formally verified.

4.4 Deletion

Due to time constraints we were limited to verifying the insertion operation, but it would be just as relevant to implement and verify certain properties about deletion. An example of this could be: *Given a tree that contains at least one occurrence of an element, after deletion of the occurence, the resulting tree will have one less occurrence of the element and the size of the tree will be decreased by one.* One could think of several other equally interesting properties to verify but we leave those up to the imagination of the reader.

4.5 Store tree heights in the node

For the look-up, insertion and deletion operations to perform in $O(logn)$ time, as they should in a proper AVL tree implementation, the height of the tree needs to be stored in the nodes rather than calculated on the fly as the height-function itself takes $O(logn)$ time to run and is called a number of times dependent on the tree size. It should mainly affect the `inorder_add_nonempty` and `add_preserves_best` proofs, but may also influence `balance_inorder`.

5 Conclusion

In this project we have implemented a binary search tree data structure, a binary search tree add operation as well as an AVL tree balancing function and an insert operation in Coq that when exclusively used will produce AVL trees. We have formally proven the following properties about our implementation in Coq:

- That our add operation does not break the binary search tree invariant
- That our add operation never produces empty trees
- That our balance method does not alter in-order traversal of trees
- That our insert method does not break the binary search tree invariant

5.1 Metrics

Here are some metrics regarding proofs about our fixpoints. The fixpoints are mentioned in the categories column. The amount of lines in the proofs vary, but are generally dependant on the amount of cases they handle. The **Total** row is not merely a summation of the other rows as the previous categories overlap, and some lemmas fall outside the categories.

Categories	**Lemmas**	**Lines**
`sorted`	12	74
`add`	6	41
`inorder`	10	86
`last`	8	43
`first`	6	39
`balance`	1	35
Total	27	177

Table 1: Code metrics about proofs.

References

1. M. Goodrich and R. Tamassia. *Algorithm Design: Foundations, Analysis and Internet Examples*. John Wiley & Sons Australia, Limited, 2002.
2. Inria.fr. The Coq Proof Assistant. `http://coq.inria.fr/`.
3. Inria.fr. Chapter 1 The Gallina specification language, 2013. `http://coq.inria.fr/doc/Reference-Manual003.html`.

A Balance fixpoint

```
Fixpoint balance (t : tree) :=
   match t with
   | leaf => leaf (*Leaf is balanced*)
   | node val lChild rChild =>
      match (height lChild) + 2 - (height rChild) with
      | 0 => (*-2, right-unbalance*)
         match rChild with
         | leaf => leaf (* shouldn't happen, right-
            unbalance *)
         | node rval rlChild rrChild =>
            match blt_nat (height rlChild) (height
               rrChild) with
            | true => (* RR-rotation *)
               node rval (node val lChild rlChild)
                  rrChild
            | false =>
               match rlChild with
               | leaf => leaf (* shouldn't happen*)
               | node rlval rllChild rlrChild => (* RL-
                  rotation *)
                  node rlval (node val lChild rllChild) (
                     node rval rlrChild rrChild)
               end
            end
        end
      | 4 => (*+2, left-unbalance*)
         match lChild with
         | leaf => leaf (* shouldn't happen, left-
            unbalance *)
         | node lval llChild lrChild =>
            match blt_nat (height llChild) (height
               lrChild) with
            | true =>
               match lrChild with
               | leaf => leaf (* shouldn't happen *)
               |  node lrval lrlChild lrrChild => (* LR-
                  rotation *)
                     node lrval (node lval llChild
                        lrlChild) (node val lrrChild
                        rChild)
               end
            | false => (* LL-rotation *)
```

```
                node lval llChild (node val lrChild rChild
                    )
            end
        end
      | _ =>
        (*Rest should be balanced at +-1 or 0*)
        (*Check the sub-trees for imbalance*)
        node val (balance lChild) (balance rChild)
   end
end.
```