МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РФ

Федеральное государственное автономное образовательное
учреждение высшего образования
ЮЖНЫЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ

Институт математики, механики и компьютерных наук
имени И. И. Воровича

Направление подготовки
02.03.02 — Фундаментальная информатика
и информационные технологии

СПЕЦИФИКАЦИЯ СТРУКТУР ДАННЫХ НА ЯЗЫКЕ ПРОГРАММИРОВАНИЯ COQ

Выпускная квалификационная работа
на степень бакалавра

Студента 4 курса
А. И. Насека

Научный руководитель:
старший преподаватель В. Н. Брагилевский

Допущено к защите:
руководитель направления ФИИТ ________________ В. С. Пилиди

Ростов-на-Дону
2017

Contents

Introduction

The aim of the work is to study $ssreflect$ library for the Coq language. In the process of studying, several data structures will be formulated, which will depend on each other.

Also, several basic operations and properties for these structures will be introduced. Theorems of the correctness of their application will be proved for these operations and properties. The proof of the theorems will use the existing $ssreflect$ library statements.

The main data structures are: a binary tree and a binary search tree. The work consists of two chapters, the binary tree will be formalized in the first, and a binary search tree will be formalized in the second one, which is based on a binary tree.

1. Binary tree

1.1. Inductive definition

The first basic data structure will be a binary tree. To implement this data structure in Coq, we give an inductive definition $BinaryTree\ T$ for some type T:

```
Inductive BinaryTree (T : Type) : Type :=
    | Nill : BinaryTree T
    | Node : BinaryTree T -> BinaryTree T -> T -> BinaryTree T ->
             BinaryTree T.
```

Type $Nill$ is a bacic constructor, describing the empty tree. Type $Node$ is a tree node constructor, which includes four fields:

- The fisrt argument have a type $BinaryTree\ T$ and it serves to provide additional information (e.g., if we have the tree with parents, this argument can be the reference to the **parent** or if we have the tree with siblings - it can be the reference to the **sibling**).
- The second argument have a type $BinaryTree\ T$ and it's a reference to the **left child**.
- The third argument have a type T. This field is a **value** of a tree node (e.g., as a type T, we can take a standart type in Coq - num and then values of this type will be natural numbers).
- The fourth argument have a type $BinaryTree\ T$ and it's a reference to the **right child**.

We give a several examples of trees (as the type T - we select num):

```
Check (Nill nat).                                   (* Admission *)
Check (Node (Nill nat) (Nill nat) 5 (Nill nat)).    (* Admission *)
Check (Node (Nill nat) (Nill nat) 5
      (Node (Nill nat) (Nill nat) 7 (Nill nat))).   (* Admission *)
```

- In the first example we have the empty tree.
- In the second - binary tree with only one node, in which all references to parent/sibling and childs are empty.
- In the last example we have the tree with two nodes. Root of tree have a value 5 and its right child have a value 7.

1.2. Functions and properties

In the basic implementation of binary trees, we won't use references to parent/sibling, we will apply only left and right childs. Therefore, we will define the properties of the getter:

```
Definition Lst {T} (tree : BinaryTree T) : BinaryTree T :=
  match tree with
  | Nill => Nill T
  | Node _ Left _ _ => Left
  end.

Definition Rst {T} (tree : BinaryTree T) : BinaryTree T :=
  match tree with
  | Nill => Nill T
  | Node _ _ _ Right => Right
  end.
```

These functions take a some binary tree, and if this tree is $Nill$ both functions return $Nill\ T$ for some T, but if this tree is a $Node$ of tree then Lst will return left child of this node, and Rst will return right child.

For clarity of examples we introduce a notation for $BinaryTree\ T$, where the type T is num and field parent/sibling = (Nill nat):

```
Notation null          := (Nill nat).
Notation "l -| v |- r" := (Node (Nill nat) l v r)
                          (at level 43, left associativity).
Notation "@ v "        := (null -| v |- null)
                          (at level 43, left associativity).
```

Now we check the correctness of these properties in Coq:

```
Eval compute in (Lst null).                   (* =>  null *)
Eval compute in (Rst null).                   (* =>  null *)

Eval compute in (Lst (@ 5)).                  (* =>  null *)
Eval compute in (Rst (@ 5)).                  (* =>  null *)

Eval compute in (Lst ((@ 3) -| 5 |- (@ 7))). (* => (@ 3) *)
Eval compute in (Rst ((@ 3) -| 5 |- (@ 7))). (* => (@ 7) *)
```

Then we implement two recursive functions for a $BinaryTree\ T$. The first one - is a **height** of tree, and the second - **count** of nodes in a tree:

```
Fixpoint height {T} (tree : BinaryTree T) : nat :=
    match tree with
    | Nill                => 0
    | Node _ Left _ Right => 1 + maxn (height Left) (height Right)
    end.

Fixpoint count {T} (tree : BinaryTree T) : nat :=
    match tree with
    | Nill                => 0
    | Node _ Left _ Right => 1 + (count Left) + (count Right)
    end.
```

- Functions **height** - is a maximum of the height of the left and right subtrees plus one.
- Functions **count** - is a count of nodes in the left and right subtrees plus one.

And check the correction these both functions:

```
Eval compute in (height null).                        (* => 0 *)
Eval compute in (count  null).                        (* => 0 *)

Eval compute in (height (@ 5)).                       (* => 1 *)
Eval compute in (count  (@ 5)).                       (* => 1 *)
Eval compute in (height ((@ 3) -| 5 |- (@ 7))).  (* => 2 *)
Eval compute in (count  ((@ 3) -| 5 |- (@ 7))).  (* => 3 *)

Eval compute in (height ((@ 3) -| 5 |-
                        ((@ 6) -| 7 |- (@ 8)))). (* => 3 *)
Eval compute in (count  ((@ 3) -| 5 |-
                        ((@ 6) -| 7 |- (@ 8)))). (* => 5 *)
```

For these two functions we will proof several simple lemmas:

```
Lemma height_nill : forall {T}, height (Nill T) = 0.
Lemma count_nill  : forall {T}, count  (Nill T) = 0.
```

These lemmas tell us, that height and count of nodes of binary tree, which is empty, equals zero. Both of them are proved trivially, since for the proof, you simply need to calculate the value of the functions for the base case $Nill$. To prove this statement with $ssreflect$ it is enough to apply the tactic $done$.

```
Lemma height_empty_tree : forall {T} (tree : BinaryTree T),
  height tree = 0 <-> tree = Nill T.

Lemma count_empty_tree : forall {T} (tree : BinaryTree T),
  count tree = 0 <-> tree = Nill T.
```

These lemmas tell us, that for all binary trees, which height or count of nodes equals zero, It follows that this tree can only be empty and no more else. The proof of these lemmas is following:

```
Proof.
  by move => T; case.
Qed.
```

As described above in the proof we use parsing cases on the structure of a tree. In the base case, when $tree = Nill$, functions on empty tree return zero, so we have two equivalent statements: $0 = 0$ and $Nill\ \ T = Nill\ \ T$.

In the second case, when tree is a node, functions are not empty tree returns nonzero, so we have two wrong statements: $n = 0$ (when $n \neq 0$) and $Node = Nill\ T$, because from lies anything follows - we proved this case and the lemma as a whole.

The following two lemmas are proved in a similar way:

```
Lemma height_subtrees : forall {T} (tree : BinaryTree T),
  tree <> Nill T -> height tree
          = 1 + maxn (height (Lst tree)) (height (Rst tree)).

Lemma count_subtrees : forall {T} (tree : BinaryTree T),
  tree <> Nill T -> count tree
          = 1 + count (Lst tree) + count (Rst tree).
```

These lemmas are opposite to the previous two assertions. If we haven't empty tree, we calculate the height and count of nodes according to base formulas.

1.3. Theorem about connection between height and count

We want to prove theorem, which tell us, that for all binary trees, their height is less than or equal to the count of nodes. For beginning we state the theorem in Coq:

```
Theorem leq_height_count : forall {T} (tree : BinaryTree T),
  height tree <= count tree.
```

We will prove this theorem by induction on the structure of a binary tree. For empty trees is definitely true, because both of these functions on empty tree return zero (we proved this lemmas before), and it follows that their results are equal, which satisfies the condition of the theorem - and we proved the base of induction.

In inductive transition, after substituting the definition of height and the number of nodes functions, we have the following statement:

$$\frac{height\ L <= count\ L,\ height\ R <= count\ R}{1 + maxn\ (height\ L)\ (height\ R) <= 1 + count\ L + count\ R}$$

Where L is the left subtree, R is the right one. On the top of the statement $height\ L <= count\ L$ and $height\ R <= count\ R$ are hypothesis, and at the bottom of the statement we have the goal, which we need to proof. So we will prove this goal by the following algorithm:

1. The first step is to reduce plus number one on both sides of the inequality:

   $$\frac{1 + maxn\ (height\ L)\ (height\ R) <= 1 + count\ L + count\ R}{maxn\ (height\ L)\ (height\ R) <= count\ L + count\ R}$$

   We will implement this step using the standard lemma in $ssrnat$ of the $ssreflect$ library.

2. The second step is replace the plus operator on the right-hand side of the inequality by a maximum:

   $$\frac{maxn\ (height\ L)\ (height\ R) <= count\ L + count\ R}{maxn\ (height\ L)\ (height\ R) <= maxn\ (count\ L)\ (count\ R)}$$

   For this we formulate and prove an 2 additional lemmas:

   ```
   First : forall x y, maxn x y <= x + y.
   ```

   ```
   Second: forall a b c d, (maxn a b <= maxn c d)
   -> (maxn a b <= c + d).
   ```

3. The next step is replace on both sides of the inequality the function of the maximum per operation plus:

$$\frac{maxn\ (height\ L)\ (height\ R) <= maxn\ (count\ L)\ (count\ R)}{height\ L + height\ R <= count\ L + count\ R}$$

To implement such a transition, we need to prove the following statement using the available hypotheses:

```
forall a b c d, a <= c -> b <= d ->
a + b <= c + d -> maxn a b <= maxn c d.
```

4. And the last step we only need to prove this statement:

   $height\ L + height\ R <= count\ L + count\ R$

   It's possible with use standard lemma in $ssreflect$:

   ```
   forall m1 m2 n1 n2 : nat, m1 <= n1 -> m2 <= n2 ->
   m1 + m2 <= n1 + n2.
   ```

1.4. Canonical comparison

Now we define a comparison function for a binary tree. This function receives two input trees $t1$ and $t2$ of type $BinaryTree\ T$ for some type T, and values of type T can be compared with each other (e.g., this condition is satisfied by the type nat or type seq). This condition we impose by means of the following construction:

```
Variables T : eqType.
Implicit Type t : BinaryTree T.
```

This restriction on type T guarantees us that instead of type T we can only use values, that can be compared to each other and no others. So, we can define a recursive function for comparing two trees without worrying about the correctness of type T.

The idea of comparing trees will be as follows: If we have two empty trees, then they are obviously equal to each other. If both trees are not empty, then each node of the tree $t1$ must be equal to the corresponding node of the tree $t2$, which means that the **values** of these nodes, their **parents/siblings**, and their **left** and **right childs** should be equal. In all other cases, trees are not equal to each other.

The implementation of this function in Coq is as follows:

```
Fixpoint eqtree t1 t2 {struct t1} :=
  match t1, t2 with
  | Nill, Nill => true
  | Node P1 L1 v1 R1, Node P2 L2 v2 R2 =>
    (v1 == v2) && eqtree P1 P2 && eqtree L1 L2 && eqtree R1 R2
  | _, _ => false
  end.
```

The restriction on type T here is manifested in this comparison: $v1 == v2$. Since $t1$ and $t2$ are values of type T, and we restricted type T to the condition that all values of this type can be compared, it follows that this comparison will not cause any errors.

But we don't just want to have our own function for comparing trees, we want to compare trees with a standard comparison, which is defined in Coq, and use all the proven lemmas and statements, that will make our life much easier, and we will not have to prove many statements from scratch.

For this, we need to instantiate a standard comparison for type $BinaryTree\ T$. First of all we need to prove the following lemma:

```
Lemma eqtreeP : Equality.axiom eqtree.
```

This lemma has always to be proved, if we want to instantiate a standard comparison for some new type. The interior of this lemma is as follows:

```
forall t1 t2 : BinaryTree T, reflect (t1 = t2) (eqtree t1 t2)
```

This statement tells us about the following that we must establish a one-to-one correspondence between the propositional comparison of two trees of type $BinatyTree\ T$ and a new comparison $eqtree$ that we defined.

This lemma prove by double induction on the structure of the construction of trees $t1$ and $t2$.

Having proved this lemma, we can instantiate a standard comparison for the type $BinaryTree\,T$. This is done in Coq as follows:

```
Canonical tree_eqMixin := EqMixin eqtreeP.
Canonical tree_eqType  := Eval hnf in
                          EqType (BinaryTree T) tree_eqMixin.
```

Now we can use the standard comparison for $BinaryTree\,T$ (Provided that type T satisfies the condition that it's values can be compared with each other):

```
Eval compute in (null == null).                (* => true  *)
Eval compute in (null ==
                 (@ 4) -| 5 |- (@ 6)).         (* => false *)
Eval compute in ((@ 4) -| 5 |- (@ 6) ==
                 (@ 4) -| 5 |- (@ 6)).         (* => true  *)
Eval compute in ((@ 3) -| 5 |- (@ 6) ==
                 (@ 4) -| 5 |- (@ 6)).         (* => false *)
```

2. Binary search tree

2.1. Imposition of conditions on a binary tree

On the basis of a formalized binary tree, we want to implement a binary search tree by introducing a number of restrictions on the original $BinaryTree\,T$. Thus, we can use the already proved lemmas and properties for BST. We will implement binary search trees for natural numbers, so in a type of T we will use the standard type nat.

For this, we must impose the following conditions on the $BinaryTree\,nat$:

1. Both subtrees - left and right - must be a binary search trees.
2. For all nodes of the left subtree of an arbitrary node X, the values of the data keys are less than the value of the data key of the node X.
3. For all nodes of the right subtree of an arbitrary node X, the values of the data keys are greater than the value of the data key of the node X.

To begin with, we define two auxiliary functions that we need in realizing of these conditions:

- The first function will receive a binary tree of type $BinaryTree\,T$ at the input, and the output will return the sequence of all nodes of this tree. For the sequence, we use the standard type in $ssreflect$ - seq:

```
Fixpoint nodes {T} (tree : BinaryTree T) : seq T :=
  match tree with
  | Nill                    => [::]
  | Node _ Left value Right =>
    value :: (nodes Left ++ nodes Right)
  end.
```

- The second function will take the input of a sequence of tree nodes and a predicate, and it will check that for each node of the tree this predicate will be true:

```
Fixpoint Is_cond (s : seq nat) (P : nat -> bool) : bool :=
  match s with
  | [::]    => true
  | y :: ys => P y && Is_cond ys P
  end.
```

Now we implement two functions that will check whether condition 2 and 3 are satisfied for a $BinaryTree\, nat$:

- Check condition 2 (**left condition**):

```
Fixpoint Left_cond (tree : BinaryTree nat) : bool :=
  match tree with
  | Nill => true
  | Node _ Left value Right =>
    Is_cond (nodes (Lst tree)) (fun y => y < value)
    && Left_cond Left
    && Left_cond Right
  end.
```

- Check condition 3 (**right condition**):

```
Fixpoint Right_cond (tree : BinaryTree nat) : bool :=
  match tree with
  | Nill => true
  | Node _ Left value Right =>
    Is_cond (nodes (Rst tree)) (fun y => value < y)
    && Right_cond Left
    && Right_cond Right
  end.
```

In these functions, for each node of a binary tree, we pass the sequence of nodes in the left subtree and the predicate $(fun\, y => y < value)$ (in the function $Left\_cond$) and the sequence of nodes in the right subtree and the predicate $(fun\ y\ =>\ value\ <\ y)$ (in the function $Right\_cond$) to the function Is_cond.

Now make sure that these functions work correctly:

```
Eval compute in (Left_cond
                (((@ 1) -| 2 |- (@ 3))
                        -| 4 |-
                  (@ 6))).              (* => true  *)
Eval compute in (Left_cond
                (((@ 3) -| 2 |- (@ 1))
                        -| 4 |-
                  (@ 6))).              (* => false *)
Eval compute in (Right_cond
                (((@ 1) -| 2 |- (@ 3))
                        -| 4 |-
                  (@ 6))).              (* => true  *)
Eval compute in (Right_cond
                (((@ 3) -| 2 |- (@ 1))
                        -| 4 |-
                  (@ 6))).              (* => false *)
```

2.2. Propositional and computational definitions

Now we can impose conditions on a binary tree and formalize the definition for a binary search tree. We will give two variants of definition - one **propositional** and the other **computational**.

Advantages of the propositional definition are: when we prove the lemmas, we have very clear proof, which reflects all the stages of the usual mathematical proof that we can do on paper. The downside of this definition is that we sometimes we have to look for complex ways to prove easy statements.

This minus is resolved in the computational definition, because in the process of proving lemmas, we actually calculate the values of functions on different arguments, and this can be done automatically, which greatly simplifies the proof, but makes it less obvious, and it is more difficult to isolate the formal logical steps of the proof.

Therefore, we will give two definitions and compare their effectiveness. And then we will prove that they are both equivalent.

- Propositional definition:

```
Inductive BST : BinaryTree nat -> Prop :=
  | bst_nill : BST null
  | bst_node : forall tree,
    Left_cond tree ->
    Right_cond tree ->
    BST (Lst tree) ->
    BST (Rst tree)  ->
    BST tree.
```

- Computational definition:

```
Fixpoint BSTeq (tree : BinaryTree nat) : bool :=
  match tree with
  | Nill => true
  | Node _ Left value Right =>
    Left_cond tree &&
    Right_cond tree &&
    BSTeq Left &&
    BSTeq Right
end.
```

Both of these definitions are very similar. Both here and there, we impose the conditions described in the previous chapter on a binary tree. But in the propositional definition, we explicitly indicate which trees can be binary search trees. We have two constructors: base bst_nill, which states that the empty tree is a binary search tree, and the main constructor bst_node, which says that all the conditions are met for some binary tree, so the tree is a binary search tree.

To implement the computational definition, we use a recursive function, at the input of which is fed a specific tree, and already by the structure of this tree all conditions are checked that this tree is a binary search tree.

2.3. Lemmas for binary search tree

Now we prove several lemmas about pro binary search trees (the formalization of the lemmas will be given only for the propositional definition, since the conditions of the lemmas are the same, and the difference is unique in their proof):

- The first lemma tells us that an empty binary tree is a binary search tree. This lemma is proved by automotives for computational definition and using the basic constructor for propositional:

  ```
  Lemma BST_Nill : BST null.
  ```

- The following lemma states that a binary tree that contains a single node is a binary search tree. The proof of computational definition also occurs automatically, and for the propositional one we consistently apply bst_node and bst_nill constructors:

  ```
  Lemma BST_One : forall y, BST (@ y).
  ```

  For clarity, we give a proof of this lemma:

  1. For the **propositional** definition:

     ```
     Proof.
       move => y;
       apply: bst_node => //;
       do [apply: BST_Nill].
     Qed.
     ```

  2. For the **computational** definition:

     ```
     Proof.
       done.
     Qed.
     ```

- The following lemma tells us that if we have three subtrees and a value, then we can make a new tree from them. Also if this tree satisfies all the conditions of the binary search tree, then it will look like:

  ```
  Lemma BST_Combine : forall tp tl v tr,
        Left_cond (Node tp tl v tr)  ->
        Right_cond (Node tp tl v tr) ->
        BST tl -> BST tr ->
        BST (Node tp tl v tr).
  ```

  The proof of this lemma for a propositional definition is carried out with the help of a similar idea, as is the proof of the previous lemma. And the proof of the deduction definition is also automatically, but with the repeated use of additional tactics, which allows us to move from the computing operator *and* to the propositional - this is necessary for splitting the conditions of belonging to the binary search tree for cases - true or false. And after the decomposition of the original lemma into several independent goals, each of them is proved automatically, that the the idea is to break the original target into smaller ones and to prove each automatically:

  ```
  case: andP => //; case.
  ```

- The following two lemmas are formulated for trees, which consist of two nodes - the root and the child. If this child is left one and the value in this node is less than the root, or if this child is right and the value in this node is greater than the value at the root, then such trees are binary search trees:

  ```
  Lemma BST_Two_l : forall a b,
        b < a -> BST ((@ b) -| a |-  null).

  Lemma BST_Two_r : forall a b,
        a < b -> BST (null  -| a |- (@ b)).
  ```

  The idea of proving these lemmas is analogous to the previous one.

- And in the end we prove two lemmas-examples. They clearly show the advantage of the computational definition before propositional, because all such examples are computed automatically, if we use a computational definition. In addition, if we use a propositional definition, then in this case we must give a rigorous proof for every concrete example that we come up with:

```
Lemma BST_Example_1 : BST ((@3)  -| 5 |- (@ 7)).
Proof.
  apply: bst_node => //;
  do [apply: BST_One].
Qed.

Lemma BST_Example_2 : BST ((null -| 3 |- (@ 4))
                                 -| 5 |- (@ 7)).
Proof.
  by apply: bst_node => //;
  do [apply: BST_Two_r | apply: BST_One].
Qed.
```

  And the proof for both lemmas using the computational definition is the following:

```
  Proof.
    done.
  Qed.
```

2.4. Theorem about equivalence between two definitions

Now we prove the equivalence of the two definitions of the binary search tree. To this end, we state and prove the following theorem:

```
Theorem bstP (tree : BinaryTree nat) :
        reflect (BST tree) (BSTeq tree).
```

We will carry out the proof of this theorem by induction on the structure of a binary tree. In the process of proof we need to show two things:

1. If a tree is admissible by a propositional definition, then it is also admissible if a computational definition is used and vice versa.
2. If the tree is inadmissible in the propositional definition, then it can't be admissible in the computational definition and vice versa.

Now we can prove the theorem, for example, for a propositional definition. But for a computational one, it can be proved using $reflect$ between two definitions.For example, We don't have to give strict proof for the example lemma, which is formulated using a propositional definition, we just simply use the $reflect$ between definitions and will prove this theorem automatically using the computational definition:

```
Lemma BST_Example_3 : BST (((@ 2) -| 3 |- (@ 4))
                                -| 5 |-
                           ((@ 6) -| 7 |- (@ 9))).
Proof.
  by apply: bstP.
Qed.
```

2.5. Safety of insert operation

We will define the operation of inserting into the binary search tree. To do this, we will define a recursive function, which is fed with a binary tree of type $BinaryType\, nat$ and an element for insertion into the tree nat:

```
Fixpoint BST_insert (value : nat) (tree : BinaryTree nat) :=
    match tree with
    | Nill                => null -| value |- null
    | Node _ Left y Right =>
        if value < y
        then (BST_insert value Left) -| y |- Right
        else if y < value
        then Left -| y |- (BST_insert value Right)
        else tree
    end.
```

If the new value is less than the value in the current node, then we insert it into the left subtree. If the new value is greater, then it turns into the right subtree. Eventually the value will be added to the tree. After this it becomes a leaf.

We introduce a notation for insert:

```
Notation "v >> t" := (BST_insert v t)
    (at level 43, left associativity).
```

Now let's check the correctness of the insert operation:

```
Eval compute in (2 >> null).
    (* => @ 2 *)
Eval compute in (2 >> (@ 3)).
    (* => @ 2 -| 3 |- null *)
Eval compute in (2 >> (@ 1)).
    (* => null -| 1 |- (@ 2) *)
Eval compute in (1 >> ((@ 2) -| 4 |- (@ 6))).
    (* => @ 1 -| 2 |-   null -| 4 |- (@ 6) *)
Eval compute in (3 >> ((@ 2) -| 4 |- (@ 6))).
    (* => null -| 2 |- (@ 3) -| 4 |- (@ 6) *)
```

Now we want to prove the theorem that if we insert a new element into the binary search tree, then after inserting it will also remain a binary search tree:

```
Theorem Safety_BST_insert : forall tree y,
        BST tree -> BST (y >> tree).
```

This theorem is proved by induction on the structure of the construction of a tree. Also, to prove this theorem, we need to formulate the following statements:

- For function $nodes$, we need to define an axiom that will guarantee us that when we add a tree element to the sequence seq, we will always add it to the head of the list, because the order of the nodes of the tree in the sequence is not important to us at all. We need to go through all the nodes of the tree and make sure that all the conditions are fulfilled in order to say, that this binary tree is a binary search tree:

```
Axiom Insert_cond : forall value tree,
   nodes (value >> tree) = value :: (nodes tree).
```

- We need to prove the lemma that after the insertion operation into the binary tree for which the **left condition** is satisfied, this condition will also be satisfied:

```
Lemma Safety_Left_cond : forall tree y,
      Left_cond tree -> Left_cond (y >> tree).
```

- We need to prove the lemma that after the insertion operation into the binary tree for which the **right condition** is satisfied, this condition will also be satisfied:

```
Lemma Safety_Right_cond : forall tree y,
      Right_cond tree -> Right_cond (y >> tree).
```

The two preceding lemmas are proved by induction on the structure of the construction of a tree.

Conclusion

In the process of writing this work, binary trees and binary search trees were certified. In addition, several variants of definitions for binary search trees were given. So the theorem on the equivalence of these definitions was proved, which allows more flexible use of these trees in the work.

We also made instantiation of the standard equality in Coq for our binary trees, which allows us to use all available standard lemmas for them.

The full code of this work can be found at github [3]

The programming language Coq was studied using an electronic resource Software Foundations [4] and the books of Adam Chlipala [1].

The library $ssreflect$ was studied with the help of lectures of Ilya Sergey [2] and documentation for this library [5]

References

1. *Chlipala A.* Certified Programming with Dependent Types. — MIT Press, 2016.
2. Programs and Proofs. — URL: `http://ilyasergey.net/pnp/` (visited on 05/24/2017).
3. Repository of this work. — URL: `https://github.com/arrival3000/Specification-of-data-structures-in-Coq.git` (visited on 05/24/2017).
4. Software Foundations. — URL: `http://www.seas.upenn.edu/~bcpierce/sf/current/index.html` (visited on 05/24/2017).
5. Ssreflect Documentation. — URL: `http://ssr.msr-inria.inria.fr/doc/ssreflect-1.5/` (visited on 05/24/2017).