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

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 will not use references to parent/sibling, we will use 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.
```

This functions takes 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 as follows:

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

As we see 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 on not empty tree return nonzero, so we have two wrong statements: $n = 0$ (when $n \neq 0$) and $Node = Nill\ T$, because from lies it follows anything - 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 have not 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. To begin with, 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 this 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. 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 the **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 must always 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 its 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. Propositional definition

2.2. Computational definition

2.3. Theorem about equivalence between two definitions

2.4. Insert operation