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