

Rocq Assignment for Proving with Computer Assistance

Herman Geuvers

You make the assignment in **couples**; please write clearly both names and student numbers on all documents you hand in.

Grading

Your assignment will be evaluated on the following points

1. Correctness Rocq: The proofs should all be completed and finished, otherwise you will get at most a 5.
2. Correctness definitions: Do the definitions capture the notions correctly?
3. Effectivity of Rocq definitions and statements: Are the definitions and lemmas well-chosen? Proper break-down in sublemmas? Notation well-chosen?
4. Effectivity of Rocq proofs: concise proofs, proper abstractions.
5. Quality of the report: explanation of problem and definitions, explanation of the main line of the proof.
6. Independency (“zelfstandigheid”): how much help/suggestions were received?
7. Plus features: Remarkable use of tactics or Rocq features; remarkably smart definitions; additional proofs and properties.

There is one **standard assignment**, but you may also choose the alternative one, or you may choose one from the list at
<http://www.cs.ru.nl/~freek/courses/tt-2017/public/huiswerk.pdf>.

The standard assignment is split up in 2 parts:

- In case you finish only part I, you can at most receive a grade of 8.
- In case you finish all proofs for part I (with correct definitions) you will receive at least a 5.5.
- In case you finish both parts I and II you can receive at most a 10 (but then of course all evaluation points should be excellent).

Standard Assignment Part I: sorting of binary search trees

- We define the inductive type `tree` representing binary trees of natural numbers.

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

- Define a predicate `bst` on `tree` to express that a tree is sorted, i.e. it is a binary search tree (see http://en.wikipedia.org/wiki/Binary_search_tree for introduction to binary search trees). Experience of the past has shown that it works very well if you define `bst` as a recursive function:

```
Fixpoint bst (T : tree) : Prop := ...
```

If you are up for a **real** challenge you can try to make this exercise using some variant of self-balancing search trees, see http://en.wikipedia.org/wiki/Self-balancing_binary_search_tree; but before you do that make sure you can solve the basic version of this exercise!

- Define a function `insert` that takes a binary search tree and a natural number and inserts the number in the right place in the tree.
- Prove correctness of the `insert` function that is prove that:

```
bst t -> bst (insert n t) (for all t:tree, n:nat).
```

- Define a function `sort` that takes an arbitrary tree and sorts it, i.e. it transforms it into a binary search tree. Hint: you can define two auxiliary functions, one that stores the elements of a tree in a list and one that builds a binary search tree from the elements of a list.
- Prove that the result of the `sort` function is always a binary search tree.
- Given the predicate `occurs` expressing that an element belongs to a tree, prove that the sorted version of a tree contains the same elements as the original one, i.e. prove:

```
“occurs n t <-> occurs n (sort t)” (for all n:nat, t:tree)
```

Standard Assignment Part II: the minimum of a binary search tree

- Define a function `treeMin` that will return the value of the minimal node in a tree. You may want to use the function `Min` (minimum function) from `Arith`. Note that every function in `Rocq` needs to be total and you will need to decide what this function should return applied on an empty tree. To do this, use the `option` type. Check the definition of `option` by doing `Print option`.
- Given the predicate `occurs` expressing that an element belongs to a tree, prove correctness of the `treeMin` function, i.e. prove that:
 - the minimal element belongs to the tree and
 - that the values in all nodes are greater or equal than the minimal value.
- Define a function `leftmost` that given a tree will return a value of its leftmost node.
- Prove that the minimal element of a binary search tree (use the predicate `bst` on `tree`) is its leftmost node.
- Define a function `search` that given a binary search tree will check whether a given natural number occurs in the tree. It should use the fact that the tree is a binary search tree so it should look only on one branch of a tree, instead of on all of its nodes.
- Prove that the `search` function is correct, i.e. prove:

```
“bst t -> (occurs n t <-> search n t)” (for all n:nat, t:tree)
```

Alternative Assignment: satisfiability (assign_sat)

We study propositional formulas and check whether they are “satisfiable”. A formula f is satisfiable if there is a valuation ρ (a valuation is a map that assigns 0 or 1 to each of the proposition variables) such that $\rho(f) = 1$.

Here, $\rho(f)$ is computed using the well-known “truth table semantics”.

- Define the inductive type of “propositional expressions” `form` with the following constructors.

```
f_var : nat -> form
f_and : form -> form -> form
f_or : form -> form -> form
f_imp : form -> form -> form
f_neg : form -> form
```

`f_var` gives us infinitely many propositional variables, that are all indexed by a natural number.

- Define the notion of a “model” as a valuation ρ that assigns a boolean to each natural number. This can be done in various ways:

- `model : nat → bool`
- `model : list(nat * bool)`
- `model : listbool`

The last two assign a boolean to only finitely many numbers, but a proposition contains only finitely many variables anyway, so that’s no problem. Each of these choices has pros and cons; probably the second is easiest to work with.

- Define a function `find_model` that, given an `e:form`, computes a model ρ in which `e` is true (i.e. in which $\rho(e) = \text{true}$).

Let `find_model` give an “error” message if no such ρ exists, by making it of type `form -> option model`. Check the definition of `option` by doing `Print option`

NB. To define `find_model`, you will probably have to:

- First collect the list of proposition variables that occur in `e`.
- Then, by recursion over this list, try out all different valuations of `{true, false}` to the proposition variables occurring in `e`.

- Prove that `find_model` “works” : if `find_model e ≠ None` , then `find_model e` produces a model of `e`.