Coq projects for dIFP 2015

Robbert Krebbers

October 13, 2015

This file gives a list of suggestions for the Coq project of dIFP. You may also invent a project yourself or do a variation of one of the projects. If you invent a project yourself or do a variation of one of the suggested projects, you have to discuss it with the teacher.

Library file. As part of your Coq project you should make use of the following library file:

`http://robbertkrebbers.nl/teaching/dIFP-2015/project_lib.v`

This file contains the definition of the `Case` tactics, as well as some common Coq definitions, such as identifiers.

Extensions. Each project ends with an exercise that lists possible extensions, of which you have to choose at least one. You are advised to first complete the whole exercise before starting to work on the extension.

1 The simply typed λ-calculus

The simply typed λ-calculus is a subsystem of Coq that only has function types. The goal of this project is to formalize the simply typed λ-calculus in Coq and to implement a type checker for it.

The types and terms are given by the following grammar:

$$A, B ::= X \mid A \to B$$
$$t, r ::= x \mid t\, r \mid \lambda x : A\,.\, t$$

Here, X ranges over a set of type variables and x over a set of term variables. Examples of types are $X \to Y \to X$ and $(X \to X) \to (X \to X)$, and examples of terms are $\lambda x : X\,.\,\lambda y : Y\,.\,x$ and $\lambda f : X \to X\,.\,\lambda x : X\,.\,f\,(f\,x)$.

In order to define the set of well-typed terms, we define a typing judgment $\Gamma \vdash t : A$ that states that *a term t has type A in environment Γ*. The environment Γ associates types to term variables. The typing judgment $\Gamma \vdash t : A$ is defined by the following inference rules:

$$\frac{x : A \in \Gamma}{\Gamma \vdash x : A} \qquad \frac{\Gamma \vdash t : A \to B \qquad \Gamma \vdash r : A}{\Gamma \vdash t\, r : B} \qquad \frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x : A\,.\,t : A \to B}$$

For example, the term $\lambda x : X\,.\,\lambda y : Y\,.\,x$ has type $X \to Y \to X$, which is shown by the following derivation:

$$\frac{\dfrac{x : X, y : Y \vdash x : X}{x : X \vdash \lambda y : Y \,.\, x : Y \to X}}{\vdash \lambda x : X \,.\, \lambda y : Y \,.\, x : X \to Y \to X}$$

We formalize the types and terms of the simply typed λ-calculus using the following inductive definitions in Coq:

```
Inductive type :=
  | tvar : id -> type
  | tarr : type -> type -> type.

Inductive term :=
  | var : id -> term
  | app : term -> term -> term
  | lam : id -> type -> term -> term.
```

The type `id` of identifiers has been introduced in the chapter "Imp" of Software Foundations and is defined in the supplemented library file.

Exercise 1.1 *Give Coq definitions of type* `type` *corresponding to:*

$$X \to Y \to X \quad \text{and} \quad (X \to X) \to (X \to X).$$

Exercise 1.2 *Give Coq definitions of type* `term` *corresponding to:*

$$\lambda x : X \,.\, \lambda y : Y \,.\, x \quad \text{and} \quad \lambda f : X \to X \,.\, \lambda x : X \,.\, f\,(f\,x).$$

We will represent environments Γ as partial functions:

```
Definition env := id -> option type.
Definition empty_env : env := fun x => None.
Definition override (E : env) (x : id) (A : type) : env :=
  fun y => if beq_id x y then Some A else E y.
```

Exercise 1.3 *Represent the typing judgment $\Gamma \vdash t : A$ as an inductively defined proposition. Do so by completing the following definition:*

```
Inductive typed : env -> term -> type -> Prop :=
  | var_typed : forall E x A,
     E x = Some A ->
     typed E (var x) A
  (* fill out *).
```

Exercise 1.4 *Prove $\vdash \lambda x : X \,.\, \lambda y : Y \,.\, x \;:\; X \to Y \to X$. You have to create a lemma of the following shape:*

```
Lemma test : typed (*your environment*) (*term*) (*type*).
```

Exercise 1.5 *Implement a type checker by filling out the following definition:*

```
Fixpoint typecheck (E : env) (t : term) : option type :=
  (* fill out *).
```

Exercise 1.6 *Explain the difference between the typing judgment* `typed` *and the type checker* `typecheck`.

Exercise 1.7 *Write 2 positive tests and 2 negative tests of the type checker and prove these using the* `reflexivity` *tactic.*

Exercise 1.8 *Prove completeness and soundness of your type checker with respect to the typing rules:*

```
Lemma typecheck_complete : forall E t A,
  typed E t A ->
  typecheck E t = Some A.
Lemma typecheck_sound : forall E t A,
  typecheck E t = Some A ->
  typed E t A.
```

To prove completeness, you may want to use `induction` *on the hypothesis* `typed E t A`. *To prove soundness, you may want to vary the induction hypothesis.*

Exercise 1.9 *Extend the language with a feature of your choice. For example:*

- *Products/conjunctions and sums/disjunctions* ***(easy)***.
- *Natural numbers and recursion* ***(moderate)***.
- *Universal/existential quantification* ***(difficult)***.
- *A relation for β-reduction and a proof of type preservation* ***(very difficult)***. *See* `https://en.wikipedia.org/wiki/Subject_reduction`.

You have to extend the types, terms, typing rules, type checker and proofs.

2 SAT solver

The goal of this project is to implement a Boolean Satisfiability (SAT) solver in Coq and to prove soundness of your implementation.

Boolean formulas are given by the following grammar:

$$p, q ::= x \mid \mathsf{true} \mid \mathsf{false} \mid p \wedge q \mid p \vee q \mid p \to q \mid \neg p.$$

Here, x ranges over an infinite set of propositional variables. Examples of formulas are $(x \vee \neg y) \wedge (\neg x \vee y)$, $y \to (x \vee y)$ and $x \wedge \neg x \wedge \mathsf{true}$.

A Boolean formula p is said to be *satisfiable* in case its truth table contains at least one row whose outcome is 1. For example:

x	y	$(x \vee \neg y) \wedge (\neg x \vee y)$
0	0	1
0	1	0
1	0	0
1	1	1

satisfiable

x	y	$\neg y \to (x \vee y)$
0	0	0
0	1	1
1	0	1
1	1	1

satisfiable

x	$x \wedge \neg x \wedge \mathsf{true}$
0	0
1	0

unsatisfiable

The first part of this project is to represent Boolean formulas in Coq and to formally define the notion of satisfiability.

Exercise 2.1 *Define an inductive type* `form` *that represents Boolean formulas. Start by completing the following definition:*

```
Inductive form :=
  | var : id -> form
  (* fill out *).
```

The type `id` *of identifiers has been introduced in the chapter "Imp" of Software Foundations and is defined in the supplemented library file.*

Exercise 2.2 *Give Coq definitions of type* `form` *corresponding to:*

$$(x \vee \neg y) \wedge (\neg x \vee y) \qquad \neg y \to (x \vee y) \qquad \text{and} \qquad x \wedge \neg x \wedge \mathsf{true}.$$

In order to define satisfiability we introduce the notion of a *valuation*, which is a function that assigns true or false to each propositional variable.

```
Definition valuation := id -> bool.
Definition empty_valuation : valuation := fun x => false.
Definition override (V : valuation)
    (x : id) (b : bool) : valuation :=
  fun y => if beq_id x y then b else V y.
```

Exercise 2.3 *Define the interpretation of a formula (using the 'truth table semantics') by filling out the following definition:*

```
Fixpoint interp (V : valuation) (p : form) : bool :=
  (* fill out *).
```

A formula is said to be *satisfiable* if there exists a valuation that makes the formula true. This corresponds to the following definition in Coq:

```
Definition satisfiable (p : form) : Prop :=
  exists  V : valuation, interp V p = true.
```

You may want to read the section "Existential Quantification" of the chapter "MoreLogic" of Software Foundations.

Exercise 2.4 *Prove in Coq that* $(x \vee \neg y) \wedge (\neg x \vee y)$ *and* $\neg y \to (x \vee y)$ *are satisfiable. You should create two lemmas of the shape below:*

```
Lemma test1 : satisfiable (*formula 1*).
Lemma test2 : satisfiable (*formula 2*).
```

Exercise 2.5 *Define a function that given a formula computes a valuation in which the formula is true. You should implement this function by enumerating all possible valuations (that is generating its truth table).*

```
Definition find_valuation (p : form) : option valuation :=
  (* fill out *).
```

The function `find_valuation` *should yield* `None` *in case no such valuation exists (i.e. the formula is unsatisfiable).*

We can now define our SAT solver as follows:

```
Definition solver (p : form) : bool :=
  match find_valuation p with
  | Some _ => true
  | None => false
  end.
```

Exercise 2.6 *Explain the difference between* `satisfiable` *and* `solver`.

Exercise 2.7 *Write 2 positive and 2 negative tests of the solver and prove these tests using the* `reflexivity` *tactic.*

Exercise 2.8 *Prove that* `solver` *is sound. That means:*

```
Lemma solver_sound : forall p,
  solver p = true -> satisfiable p.
```

Exercise 2.9 *Extend your SAT solver in one of the following ways:*

- *Extend your development to* three-valued logic ***(moderate)***. *Three-valued logic has three truth values:* true, false *and some indeterminate third value. See* ***https://en.wikipedia.org/wiki/Three-valued_logic***.
- *Write an optimizer that simplifies a Boolean formula using the laws below, and prove correctness of your optimizer* ***(moderate)***.

  $x \wedge \mathsf{true} = x$ $\quad \mathsf{true} \wedge x = x$ $\quad x \wedge \mathsf{false} = \mathsf{false}$ $\quad \mathsf{false} \wedge x = \mathsf{false}$

  $x \vee \mathsf{true} = \mathsf{true}$ $\quad \mathsf{true} \vee x = \mathsf{true}$ $\quad x \vee \mathsf{false} = x$ $\quad \mathsf{false} \vee x = x$

  You should incoorporate this optimizer in `solver` *and its correctness proof.*

- *Write a converter that transforms Boolean formulas into negation normal form and prove correctness of your converter* ***(difficult)****. A Boolean formula is in negation normal form if the negation operator ¬ is only applied to variables. You should incoorporate this optimizer in* `solver` *and its correctness proof.*
- *Write a converter that transforms Boolean formulas into conjunctive normal form and prove correctness of your converter* ***(very difficult)****. See also* `https://en.wikipedia.org/wiki/Conjunctive_normal_form`*. You should incoorporate this optimizer in* `solver` *and its correctness proof.*
- *Prove completeness of your solver* ***(very difficult)****. That means:*

```
Lemma solver_complete : forall p,
  satisfiable p -> solver p = true.
```

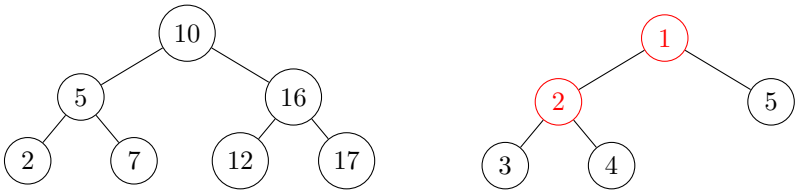
3 Binary search trees

The goal of this project is to implement some operations on binary search trees in Coq and to prove essential properties of the implementation.

A binary search tree is a data structure to efficiently store a finite set of natural numbers. Binary trees are inductively defined in Coq as follows:

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

A binary tree is said to be a binary search tree if it satisfies the *binary search tree property*, which states that the value in each node is greater (but not equal to) than all values in the left sub-tree, and smaller (but not equal to) than all values in the right sub-tree. For example, the tree on the left satisfies the property, but the one on the right does not (leafs are omitted):



The tree on the right does not satisfy the binary search tree property because there is a node numbered 2 as a left sub-tree of a node numbered 1.

Exercise 3.1 *Give Coq definitions of type* `tree` *corresponding to the above binary trees.*

Exercise 3.2 *Define an inductively defined proposition that describes whether a binary tree satisfies the binary search tree property. Do so by completing the following definition:*

```
Inductive sorted : tree -> Prop :=
```

```
  | leaf_sorted : sorted leaf
  (* fill out *).
```

In order to define the above function you may want to define helpers `greater : nat -> tree -> Prop` *and* `smaller : nat -> tree -> Prop` *or* `all : (nat -> Prop) -> tree -> Prop` *as inductively defined propositions.*

Exercise 3.3 *Prove that the first tree you have defined in Exercise 3.1 satisfies the binary search tree property, and that the second one does not. You may want to make use of composition* `;` *of tactics and the* `repeat` *tactical.*

Exercise 3.4 *Define a function that describes whether an element is contained in a binary search tree or not:*

```
Fixpoint elem_of (x : nat) (t : tree) : bool :=
  (* fill out *).
```

To define an efficient implementation of the `elem_of` *function, you should not traverse the whole tree, but make use of the binary search tree property.*

Exercise 3.5 *Write 2 positive tests and 2 negative tests of the* `elem_of` *function and prove these using the* `reflexivity` *tactic.*

The last part of this project is to define a function that inserts a natural number into a binary search tree. You have to prove that the resulting tree is indeed a binary search tree, and you have to prove that it is correct.

Exercise 3.6 *Define a function that inserts a natural number into a binary search tree:*

```
Fixpoint insert (x : nat) (t : tree) : tree :=
  (* fill out *).
```

If the number `x` *is already in the tree, it should return the original tree.*

Exercise 3.7 *Write 2 tests of the* `insert` *function and prove these using the* `reflexivity` *tactic.*

Exercise 3.8 *Prove that the* `insert` *function preserves the binary search tree property. That means:*

```
Lemma insert_sorted : forall t x,
  sorted t -> sorted (insert x t).
```

Exercise 3.9 *Prove that the* `insert` *function is correct. That means:*

```
Lemma insert_correct : forall t x y,
  sorted t ->
  elem_of y (insert x t) = orb (elem_of y t) (beq_nat x y).
```

Exercise 3.10 *Extend your development on binary search trees with a feature of your choice. For example:*

- *A converter from lists to binary search trees and vice versa. Prove a useful property about the interaction between these functions* ***(easy)****.*
- *A delete function. Prove that delete preserves the binary search tree property and prove a property about the interaction with* `elem_of` ***(moderate)****.*
- *Self-balancing search trees, such as black-red trees or AVL trees* ***(very difficult)****. See also* ***https://en.wikipedia.org/wiki/Self-balancing_binary_search_tree****.*

4 Arithmetic expression decompiler

The goal of this project is to formalize a compiler and decompiler for an arithmetical expression language. You have to prove that the compiler is correct and that the decompiler is an inverse of the compiler.

We consider an expression language of arithmetical expressions that is defined by the following inductive types in Coq:

```
Inductive unop :=
  | OSucc : unop
  | OPred : unop.

Inductive binop :=
  | OPlus : binop
  | OMult : binop.

Inductive exp :=
  | ENat : nat -> exp
  | EUnOp : unop -> exp -> exp
  | EBinOp : binop -> exp -> exp -> exp.
```

The expression `ENat` denotes a constant. The unary operator `OSucc` computes the successor and `OPred` computes the predecessor (which is defined to be 0 in case the argument is 0). The operators `OPlus` and `OMult` perform addition and multiplication as usual.

Exercise 4.1 *Define a function* `eval` *that gives a semantics to expressions. Fill out the following template:*

```
Fixpoint eval (e : exp) : nat :=
  (* fill out *).
```

Exercise 4.2 *Write 2 tests of the* `eval` *function and prove these tests using the* `reflexivity` *tactic.*

The first part of this project is to define a compiler from arithmetical expressions to a stack machine. The instructions of the stack machine are in reverse Polish notation and are given by the following Coq definitions:

```
Inductive instruction :=
  | IPush : nat -> instruction
  | IUnOp : unop -> instruction
  | IBinOp : binop -> instruction.
```

Exercise 4.3 *Implement a compiler:*

```
Fixpoint compile (e : exp) : list instruction :=
  (* fill out *).
```

Exercise 4.4 *Implement a virtual machine for stack instructions.*

```
Definition vm : list instruction -> option nat :=
  (* fill out *).
```

The virtual machine should yield `None` *in case of stack underflow. For example,* `vm [EBinOp OPlus; IPush 10] = None`.

Exercise 4.5 *Write 2 positive tests and 2 negative tests of the virtual machine and prove these using the* `reflexivity` *tactic.*

Exercise 4.6 *Prove correctness of your compiler:*

```
Lemma vm_correct : forall e,
   vm (compile e) = Some (eval e).
```

The next part of this project is to define a decompiler. The decompiler translates a list of machine instructions into a corresponding arithmetical expression. You have to show that the decompiler is an inverse of the compiler.

Exercise 4.7 *Define a decompiler:*

```
Definition decompile : list instruction -> option exp :=
  (* fill out *).
```

The decompiler should yield `None` *in case the list of instructions is ill-formed. For example,* `decompile [EBinOp OPlus; IPush 10] = None`.

The implementation of the decompiler is very similar to the implementation of the virtual machine. However, instead of evaluating a list of instructions to a value, you have to evaluate the list to an expression.

Exercise 4.8 *Prove that the decompiler is a left-inverse of the compiler:*

```
Lemma decompile_correct : forall e,
  decompile (compile e) = Some e.
```

Exercise 4.9 *Extend your project in one of the following ways:*

- *Extend the language with an operator that may fail (for example, division may fail in case of division by zero)* ***(moderate)****. You have to extend the syntax and semantics of the expression and machine language, as well as your proofs.*
- *Extend the language with a non-deterministic random number generator* ***(difficult)****. You have to extend the syntax and semantics of the expression and machine language, as well as your proofs.*
- *Prove that the decompiler is a right inverse of the compiler* ***(difficult)****:*

  ```
  Lemma compile_decompile : forall ins e,
    decompile ins = Some e -> compile e = ins.
  ```

- *As you may have noticed, the decompiler and the virtual machine, as well as its correctness proofs, are very similar. Try to figure out an abstraction so you can factor out similarities* ***(very difficult)****.*

5 Monads

The goal of this project is to obtain an understanding of monads, to be able to prove the laws of commonly used monads, and to program in monadic style.

Monads are an abstract way of capturing many common programming concepts such as partiality, error handling, statefulness, non-determinism and continuations in a functional programming language. A monad consists of a type constructor M : `Type` $\to$ `Type` and two operations:

- The `ret` operation takes a value $x : A$ from a plain type and injects it into the monad. The value `ret` x has type $M\ A$.
- The `bind` operation takes a monadic value $m : M\ A$ and applies a function $f : A \to M\ B$ to it. The resulting value `bind` $m\ f$ has type $M\ B$.

The purpose of these operations is best explained by an example. For that, we consider the `option` monad:

```
Definition option_ret {A} (x : A) : option A := Some x.
Definition option_bind {A B}
```

```
    (m : option A) (f : A -> option B) : option B :=
  match m with
  | Some x => f x
  | None => None
  end.
```

The `option` monad is used to model partiality: the constructor `Some` denotes that a computation has succeeded, and the constructor `None` denotes that a computation has failed. As shown in the above definition, `option_ret` models a successful computation, and `option_bind` propagates failures.

Recall the `index` function from the chapter "Poly" of Software Foundations that returns the nth element of a list, and yields `None` in case the element does not exist. We can use `option_bind` to use it multiple times, without having to perform explicit pattern matches on the results of `index`.

```
Fixpoint index {A} (n : nat) (l : list A) : option A :=
  match l with
  | [] => None
  | a :: l' =>
     if beq_nat n 0 then Some a else index (pred n) l'
  end.
Definition test (l : list nat) : option nat :=
  option_bind (index 1 l) (fun n1 =>
  option_bind (index 3 l) (fun n2 =>
  option_bind (index 5 l) (fun n3 =>
    option_ret (n1 + n2 + n3)))).
```

The function `test` retrieves the 1st, 3rd and 5th of the list `l` and sums these elements. If one of the elements does not exist, the whole function `test` fails by returning `None`. Let us try it out:

```
Eval compute in test [1;2;3;4;5;6;7;8;9;10]. // Some 12
Eval compute in test [1;2;3;4]. // None
```

Exercise 5.1 *Write 2 positive tests and 2 negative tests for multiple uses of* `index` *using* `option_ret` *and* `option_bind` *and prove these using the* `reflexivity` *tactic.*

An important part of a monad is that it satisfies the *monad laws*: `ret` is a neutral element for `bind` and binding two functions consecutively is the same as binding one function that is composed using a `bind`. These laws are expressed by the following Coq definition that is parametrized by a type constructor `M` and the operations `ret` and `bind`:

```
Definition monad
    {M : Type -> Type}
    (ret : forall {A}, A -> M A)
    (bind : forall {A B}, M A -> (A -> M B) -> M B) :=
  (forall A B (f : A -> M B) x, bind (ret x) f = f x) /\
  (forall A (m : M A), bind m ret = m) /\
  (forall A B C (f : A -> M B) (g : B -> M C) (m : M A),
    bind (bind m f) g = bind m (fun x => bind (f x) g)).
```

Exercise 5.2 *Prove that* `option` *satisfies the monad laws:*

```
Lemma option_monad : monad (@option_ret) (@option_bind).
```

Another example is the `list` monad which can be used to deal with functions that yield multiple values. The `ret` function yields the singleton list, and `bind` applies a function to each element of the list and flattens the result.

```
Definition list_ret {A} (x : A) : list A := [x].
Fixpoint list_bind {A B}
    (l : list A) (f : A -> list B) : list B :=
  match l with
  | [] => []
  | x :: l => f x ++ list_bind l f
  end.
```

Exercise 5.3 *Implement a function that computes all permutations of a list:*

```
Fixpoint permutations {A} (l : list A) : list (list A) :=
```

For example `permutations [1;2;3]` *may yield* `[[1;2;3]; [2;1;3]; [2;3;1]; [1;3;2]; [3;1;2]; [3;2;1]]` *(it does not matter in which order the permutations appear in the result). You should use* `list_ret` *and* `list_bind`*, and you may define at most one helper function using a* `Fixpoint`*.*

Exercise 5.4 *Prove that* `list` *satisfies the monad laws:*

```
Lemma list_monad : monad (@list_ret) (@list_bind).
```

Yet another example is the `state` monad, which attaches state information to a type. Given a type `A`, the corresponding type is a function that takes a state, and yields a resulting state and the return value of type `A`. The `ret` function produces a value without changing the state, and the `bind` function propagates changes to the state:

```
Definition state (A : Type) : Type := nat -> (nat * A).
```

We have represented states using natural numbers, but one could of course use any type, or make the definition polymorphic in the type of states. For the purpose of this exercise, this will not be necessary.

Exercise 5.5 *Implement the operations* `state_ret` *and* `state_bind`*. Prove that your operations satisfy the monad laws:*

```
Lemma state_monad : monad (@state_ret) (@state_bind).
```

In order to prove the above lemma, you need the functional extensionality axiom, which states that functions f and g are equal if $\forall x \,.\, f\, x = g\, x$. It can be obtained by adding the following to the beginning of your file:

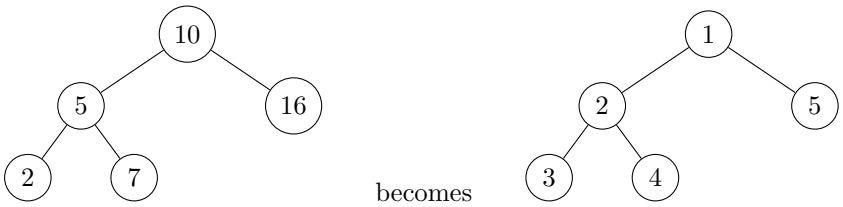
```
Require Import FunctionalExtensionality.
```

Use `Check @functional_extensionality` *to see the Coq statement of the imported functional extensionality axiom.*

In the remaining part of this exercise, we will use the state monad to implement a function that relabels binary trees. Binary trees are defined as:

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

The relabel function should preserve the tree structure, but change the labels such that they are numbed consecutively from left to right. For example:



In order to implement this function, we use the state monad to keep track of a counter. Each time we encounter a node, we increase the counter and relabel it. First we define a helper function to increase the counter.

Exercise 5.6 *Write a function that produces the value of the state and increases the value of the state by one:*

```
Definition state_inc : state nat := (* fill out *).
```

Exercise 5.7 *Write a function that relabels trees as described above:*

```
Fixpoint relabel (t : tree) : state tree :=
  (* fill out *)
```

You should use the monadic operations `state_ret`, `state_bind` *and* `state_inc`.

Exercise 5.8 *Write 2 tests of the* `relabel` *function and prove these using the* `reflexivity` *tactic.*

Exercise 5.9 *The size of a binary tree is defined as:*

```
Fixpoint size (t : tree) : nat :=
  match t with
  | leaf => 0
  | node x l r => S (size l + size r)
  end.
```

Write a lemma that states that the `relabel` *function preserves the size of a binary tree and prove this lemma.*

Exercise 5.10 *Extend your project in one of the following ways:*

- *Implement a non-trivial algorithm of your choice using one of the monads we have previously defined, and prove an interesting property about this algorithm* ***(easy)****.*
- *Implement another monad (for example the error monad, the reader monad or the continuation passing style monad) and prove that your implementation enjoys the monad laws* ***(easy)****. See* `https://en.wikipedia.org/wiki/Monad_(functional_programming)#Other_examples`*.*
- *Define an inductively defined relation that describes that two binary trees have the same shape. Write a lemma that states that the* `relabel` *function preserves the shape of a binary tree and prove this lemma* ***(moderate)****.*
- *Define the notion of an* additive monad *and show that the option and list monad are an instance of it* ***(moderate)****. See* `https://en.wikipedia.org/wiki/Monad_(functional_programming)#Additive_monads`*.*
- *Monads can also be defined in terms of the operations:*

$$\begin{aligned} \mathsf{ret} &: \forall A\,.\,A \to M\,A \\ \mathsf{fmap} &: \forall A\,B\,.\,(A \to B) \to (M\,A \to M\,B) \\ \mathsf{join} &: \forall A\,.\,M\,(M\,A) \to M\,A. \end{aligned}$$

  satisfying certain monadic laws (see `https://en.wikipedia.org/wiki/Monad_%28functional_programming%29#fmap_and_join`*). Prove that each* ret/bind *monad is a* ret/fmap/join *monad, and* vice versa ***(difficult)****. You are allowed to make use of the functional extensionality axiom.*