λCoq exercises for beginners

July 12, 2014 - Tagged as: coq, en.

Formalizing abstractions/data structures and proving theorems about them in Coq is so r
simple exercises that consist of encoding some abstractions and laws we know from alge
and then proving that some particular set + some operations on that set obeys the laws.

Using my amazing(!) JavaScript skills, I set up some "show/hide answer" buttons after eac
but the latter ones are relatively harder. Some abstractions/laws are inspired by Haskell.

Please note that I'm a beginner so my solutions probably have some flaws if you want to
programs :) I'm currently learning about typeclasses and records of Coq and I'm open to s

In exercises, when we talk that an abstraction should obey some laws, you need to enfor
need to make constructors in a way that user would have to prove that the data structure

```
Require Import List.
Import ListNotations.

Open Scope list_scope.
```

Exercise 1

A semigroup is a set together with an associative binary function. For example, natural n
form a semigroup, because we know/can prove that addition function is associative. Mor

```
forall (n1 n2 n3 : nat), n1 + (n2 + 3) = (n1 + n2) + n3.
```

Encode semigroups in Coq.

Show solution (ex. 1.1)

Now prove that lists together with append operation form a semigroup. Use standard Co

```
Theorem list_semigroup : forall A, semigroup (list A) (@app A).
Proof.
  intro. apply Semigroup_intro. intros.
  induction a1.
  + reflexivity.
  + simpl. f_equal. induction a2; auto.
Qed.
```

Hide solution (ex. 1.2)

Exercise 2

A monoid is a semigroup with an identity element. In our addition example, identity elem
to the monoid function(addition) as first or second argument, results is the other argume

```
forall (n : nat), 0 + n = n /\ n + 0 = n.
```

Encode monoids in Coq.

```
Inductive monoid A Op (sg : semigroup A Op) (U : A) : Prop :=
| Monoid_intro :
    semigroup A Op -> (forall (a : A), Op U a = Op a U /\ Op U a = a
```

Hide solution (ex. 2.1)

Now prove that lists with empty list as unit element together with the proof that lists are
previous exercise, form a monoid.

```
Theorem list_monoid : forall A, monoid (list A) (@app A) (@list_semi
Proof.
  intro. apply Monoid_intro. apply list_semigroup.
  intro. split.
  + rewrite app_nil_r. reflexivity.
  + reflexivity.
Qed.
```

Hide solution (ex. 2.2)

Exercise 3

In this exercise and exercise 4, we'll be talking about Haskell definitions of abstractions, ir
(although they may coincide)

A functor is a type with one argument(in Haskell terms, a type with kind `* -> *`) and a fu
If you're unfamiliar with functors of Haskell, you may want to skip this, or read Typeclasso

A Coq definition would use these to encode functors:

- Functor type: `F : Type -> Type`
- Functor operation: `forall t1 t2, (t1 -> t2) -> f t1 -> f t2` (let's call it fmap

A functor should obey these laws:

- `fmap id = id`
- `fmap (fun x => g (h x)) = fun x => (fmap g (fmap h x))`

Encode functors in Coq.

Show solution (ex. 3.1)

Now prove that lists with standard map function form a functor.

Show solution (ex. 3.2)

Exercise 4

A monad is a functor with two more operations; let's call `bind` and `lift` and some more l
functor type)

- bind: `forall t1 t2, F t1 -> (t1 -> F t2) -> F t2`
- lift: `forall t, t -> F t`

Laws:

- Left identity: `forall t1 t2 a f, bind t1 t2 (lift t1 a) f = f a`
- Right identity: `right_id : forall t m, bind t t m (lift t) = m`
- Associativity: `forall t1 t2 t3 m f g, bind t2 t3 (bind t1 t2 m f) g = bin`
`t3 (f x) g)`

Encode monads in Coq.

Show solution (ex. 4.1)

Now prove that lists form a monad. You need to figure out what functions to use for `lift`

Show solution (ex. 4.2)

Exercise 5

Prove that standard `option` type with some operations form a semigroup, monoid, funct
relevant operations.

What restrictions do you need on `options` type argument? (A in `option A`) Does it need t
form a monoid?

```
Definition map_option (A B : Type) (f : A -> B) (opt : option A) :=
  match opt with
  | None => None
  | Some t => Some (f t)
  end.

Definition append_option A OpA (sg : semigroup A OpA) (a b : option
```

```
  match a, b with
  | None, None => None
  | None, Some b' => Some b'
  | Some a', None => Some a'
  | Some a', Some b' => Some (OpA a' b')
  end.

Theorem option_semigroup : forall A OpA (sg : semigroup A OpA),
  semigroup (option A) (append_option A OpA sg).
Proof.
  intros. apply Semigroup_intro. intros. destruct a1.
  + destruct a2.
    - destruct a3.
      * simpl. f_equal. inversion sg. apply H.
      * simpl. reflexivity.
    - destruct a3; simpl; reflexivity.
  + destruct a2; destruct a3; auto.
Qed.

Theorem option_monoid : forall A OpA (sg : semigroup A OpA),
  monoid (option A) (append_option A OpA sg) (option_semigroup A OpA
Proof.
  intros. apply Monoid_intro. apply option_semigroup.
  intros. split. auto. destruct a; auto.
Qed.

Definition option_map A B (f : A -> B) (o : option A) : option B :=
  match o with
  | None => None
  | Some a => Some (f a)
  end.

Theorem option_functor : functor option option_map.
Proof.
  apply Functor_intro; intros; destruct f; auto.
Qed.

Definition option_bind A B (o1 : option A) (f : A -> option B) : opt
  match o1 with
  | None => None
  | Some a => f a
  end.

Theorem option_monad : monad option.
Proof.
  apply Monad_intro with (fmap := option_map) (lift := Some) (bind :
  + apply option_functor.
  + intros. auto.
  + intros. destruct m; auto.
  + intros. destruct m; auto.
Qed.
```

Hide solution (ex. 5.1)

Exercise 6

I only have a partial solution to this one and it's not strictly a Coq exercise, but it's still fun

A group is a monoid with inverse element of every element. In Coq syntax:

```
forall e, exists e_i -> op e e_1 = U
```

where `op` is monoid operation and `U` is unit of monoid.

Can you come up with a data structure that forms a group?

Show solution (ex. 6.1)