A formalization of monoids in Coq

Christopher Riis Bubeck Eriksen
Student number: 201206065

April 6, 2015

Contents

1 Introduction

Given a set 'S' and a binary operation 'Dot' over S, S is said to form a monoid with Dot if Dot is associative and there exists a left and right neutral element for Dot in S. I have expanded this definition to involve a type instead of a set and use an equality relation: Given a type T, which represents the type of elements in a set S, and a binary operation on T, T is said to form a monoid with the binary operation under some equality relation at T if the binary operation is associative under the relation and there exists a left and right neutral element of type T for the binary operation under the relation.

The goal of this project is to formalize monoids in Coq and prove 13 items to have or not have the properties of a monoid. Furthermore, all equality relations and types will be formalized without any dependencies i.e the project is as complete as possible. The only libraries used are the logic library, which contains essentials about conjunction, implication etc., and a little part of the Arith library. Coq's equality is rarely used.

2 Formalizing the specification of a monoid in Coq

When I started the project, I was formalizing the specification only with Coq's polymorphic equality in mind. A problem soon arose since the equality relation between lazy lists and the equality relation between functions are not that in-built equality. I realized I had to include the option of including a relation in the specification. Letting T be a type; eqT be a equality relation on type T; and Dot be a binary operation on T, this is my specification of a monoid in Coq:

```
Definition T_and_Dot_form_a_Monoid_under_eqT
          (T : Type)
          (eqT : T -> T -> Prop)
          (Dot : T -> T -> T) :=
  specification_of_an_equivalence_relation T eqT
  /\
  specification_of_preservation T eqT Dot
  /\
  (specification_of_associativity T eqT Dot)
  /\
  (exists e : T, specification_of_neutrality T eqT Dot e).
```

To prove that a type T with Dot form a monoid under some equality relation on T, one can formulate the lemma as

```
Lemma SomeT_and_SomeDot_form_a_Monoid_under_someRelation :
    T_and_Dot_form_a_Monoid_under_eqT SomeT someRelation someDot.
```

As one can see, four properties are present in the specification.

The first is that the equality relation at T must be an equivalence relation - it must be symmetric, reflexive and transitive. Furthermore, the binary operation needs to preserve the equality relation:

```
Definition specification_of_preservation (T : Type)
                          (eqT : T -> T -> Prop)
                          (Dot : T -> T -> T) :=
  forall a b c d : T,
    eqT a b ->
    eqT c d ->
    eqT (Dot a c) (Dot b d).
```

This property is very important, since, in layman's terms, it grants one the ability to replace equals by equals, i.e, to rewrite. The third property is that the binary operation needs to be associative for the equality relation. Denoting the equality relation as '$\sim$' and the binary operation as '.', the associativity means that (a . b) . c $\sim$ a . (b . c) for any a, b and c of type T. The last property is about the existence of a neutral element - there must exist some e of type T such that for any x of type T, x . e $\sim$ e . x $\sim$ x. Because of the transitive properties of the equality relation this means that the neutral element must be both right and left neutral.

3 Natural numbers and an equality relation on their set

I want to prove that natural numbers form a monoid with each of the binary operations addition, multiplication, minimum and maximum. However it makes sense to define the natural numers and an equality relation on them first. A natural number is defined to be either zero or the successor of another natural number. Furthermore it must be possible to do induction on them i.e. if a property holds for zero and given that the property holds for any natural it holds for the successor of that natural, the property must hold for all naturals. This can be formalized in Coq as the following inductive:

```
Inductive Nat : Type :=
  | O : Nat
  | S : Nat -> Nat.
```

Coq will automatically define the induction property as Nat_ind - it will be possible to do induction on naturals with the induction tactic. I will define the equality relation on the natural numbers as a fixpoint called eq_Nat. It should be the case that zero is equivalent only to itself and not to any other number. Furthermore two naturals which are not zero will be equivalent if their predecessors are. In Coq this is

```
Fixpoint eq_Nat (m n : Nat) : Prop :=
  match m with
      | O => match n with
                | O => True
                | S n' => False
             end
      | S m' => match n with
                  | O => False
                  | S n' => eq_Nat m' n'
                end
  end.
```

From this fix-point one can make four unfolding lemmas, one of them being ∼(eq_Nat O (S n')). This equality relation can be proved to be an equivalence relation.

All of the three properties of being an equivalence relation can be proved through induction and contradiction. I remind the reader that the contradiction tactic in Coq clears any sub-goal if the property False can be derived from some hypothesis or earlier proof in the current context. Also, ∼A is notation for A → False. Reflexivity can be proved through induction since zero is equivalent to itself and the successor of a natural number is equivalent to itself if the natural number itself is. Symmetry follows from induction and the facts that if some natural is equivalent to Zero it must be Zero; if a natural is equivalent to the successor of a natural it must itself be the successor of a natural; the relation on Zero is symmetric; and if two successors are equivalent their predecessors are (and vice versa). Transitivity follows from the same facts; that the relation on only Zero is transitive; and induction.

It can be shown that any binary operation on natural numbers will preserve eq_Nat. This resembles the fact that every natural number is in its own private equivalence class i.e. if eq_Nat m n then m has exactly the same amount of successor applications as n has. If two different numbers x and y were in the same equivalence class one would be able to derive a function which yields different output on x and y even though x and y are equivalent. The preservation can be proved using double induction, contradiction and a good idea. The idea is to do the outer induction where only one number is introduced - the binary operation should be a part of the induction hypothesis. The inner induction, which is present in the base case of the outer induction, should follow the same principle. Since the successor constructor is a function, one can 'withdraw' it into the binary operation with the induction hypothesis. Half of the proof is to use contradiction to narrow the values of the four natural numbers and the other half is to use the induction hypothesis. A lot of destruction is also going on. Since the report should not only be about this lemma, I will here only give one example of the application of the induction hypothesis:

```
a' : Nat
IHa' : forall (f : Nat -> Nat -> Nat) (b c d : Nat),
       a' =N= b -> c =N= d -> f a' c =N= f b d
f : Nat -> Nat -> Nat
b' : Nat
c' : Nat
d' : Nat
H_ab : a' =N= b'
H_cd : c' =N= d'
============================
 f (S a') (S c') =N= f (S b') (S d')
```

This is the induction step of the outer induction where all of a, b, c and d must be the successors of some numbers, derived by using contradiction. Also, the unfolding lemma about eq_Nat (S m) (S n) ↔ eq_Nat m n has been applied in H_ab and H_cd. Checking the induction hypothesis with the four numbers and the two hypotheses, one sees it is assumed that for all binary operations f on natural numbers, eq_Nat (f a' c') (f b' d').

The function adding a successor to both of its input and applying f to these will yield the subgoal i.e. (fun x y → f (S x) (S y))! By assuming that applying any function on a' and c' is equivalent to applying the same function on b' and d', it can proven that applying any function on (S a') and (S c') is equivalent to applying the same function on (S b') and (S d')!

4 Addition of natural numbers

To add two natural numbers m and n, one can apply all the successors of m onto n. The specification of addition in Coq is

```
Definition specification_of_addition (NAdd : Nat -> Nat -> Nat) :=
  (forall n : Nat,
     NAdd O n =N= n)
  /\
  (forall m' n : Nat,
     NAdd (S m') n =N= S (NAdd m' n)).
```

If a binary operation on the natural numbers satisfies the specification, it will recursively apply all successors of m onto n. It can be proved that the specification is unique i.e. if two functions satisfying the specification is given related input, they yield related output. It also follows from the specification that zero is left neutral for addition under eq_Nat. To prove that zero is right neutral requires induction - and some thought since polymorphic equality isn't used.

If one uses polymorphic equality in Coq, one can use the rewrite tactic. For example, if one has the sub-goal m = n and a hypothesis which says that m = p, one can easily rewrite m = n to p = n. Polymorphic equality is an equivalence relation and preserved by ALL functions with any number of input and any involved types. Coq has designed the rewrite tactic to use this property without having to apply a load of transitivity and preservation lemmas. Since I have designed the equivalence relation, I have to apply that load of lemmas to rewrite something. This is a crucial part of the whole project, but I need only explain it once here since I will prove that any equality relation I use is an equivalence relation and that any binary operation I prove something about preserves the equivalence relation in the same domain. I will illustrate how to rewrite with a preserving binary operation and an equivalence relation through some of the proofs of the properties of addition.

I will prove that zero is right neutral for addition. The base case of the induction is trivial; is follows from the specification that O + O is O. Here is how the start of the induction step looks like in Coq:

```
NAdd : Nat -> Nat -> Nat
H_SNA_O : forall n : Nat, NAdd O n =N= n
H_SNA_S : forall m' n : Nat, NAdd (S m') n =N= S (NAdd m' n)
n' : Nat
IHn' : NAdd n' O =N= n'
============================
 NAdd (S n') O =N= S n'
```

According to the transivity of eq_Nat, for all natural numbers a, b and c, a =N= b and b =N= c implies that a =N= c. Let the left side of the sub-goal be a and the right side be c. Let (S (NAdd n' O)) be b. I want to 'rewrite' a into b such that I can apply the unfolding eq_Nat lemma about successors on both sides and then the induction hypothesis. To rewrite, I can apply the transitivity lemma:

```
apply (eq_Nat_is_transitive (NAdd (S n') O) (S (NAdd n' O))).
```

which will produce:

```
  NAdd : Nat -> Nat -> Nat
  H_SNA_O : forall n : Nat, NAdd O n =N= n
  H_SNA_S : forall m' n : Nat, NAdd (S m') n =N= S (NAdd m' n)
  n' : Nat
  IHn' : NAdd n' O =N= n'
  ============================
   NAdd (S n') O =N= S (NAdd n' O)

subgoal 2 (ID 602) is:
 S (NAdd n' O) =N= S n'
```

That is, if I can prove that a =N= b, I can focus on proving b =N= c instead of a =N= c. The first sub-goal is proved by applying H_SNA_S with n' and O, and I can now apply the successor lemma to the second subgoal and prove it by the induction hypothesis.

It has now been shown that addition preserves the equality relation (all binary operations does); that the equality relation is an equivalence relation; and that there exists a neutral element. To prove that natural numbers with addition form a monoid under eq_Nat requires one to prove the associativity of addition. The lemma follows from induction on the left-most input since zero is neutral in the base case and because of the fact that one can move the successor completely out in the induction case; apply the induction hypothesis; and move the successor back in. I can not prove this solely based on the transitivity of eq_Nat. Consider the following sub-goal:

```
NAdd : Nat -> Nat -> Nat
H_SNA_O : forall n : Nat, NAdd O n =N= n
H_SNA_S : forall m' n : Nat, NAdd (S m') n =N= S (NAdd m' n)
b : Nat
c : Nat
============================
 NAdd (NAdd O b) c =N= NAdd b c
```

Here I want to rewrite (NAdd O b) into b. This is not possible by applying the lemma about transitivity of eq_Nat. But I know that any binary operation preserves eq_Nat. I have proven two corollaries which makes it easier to apply a preservation lemma, one of them being:

```
Lemma Dot_Nat_preserves_eq_Nat_RR :
  forall f : Nat -> Nat -> Nat,
    forall a b c : Nat,
      a =N= b ->
      f a c =N= f b c.
```

Where RR stands for right reflexive. The other lemma is for left reflexive i.e. almost the same lemma but where c is on the left side of both f. Turning back to the sub-goal in the associativity proof I can apply Dot_Nat_preserves_eq_Nat_RR. This produces the sub-goal:

```
NAdd : Nat -> Nat -> Nat
H_SNA_O : forall n : Nat, NAdd O n =N= n
H_SNA_S : forall m' n : Nat, NAdd (S m') n =N= S (NAdd m' n)
b : Nat
c : Nat
============================
 NAdd O b =N= b
```

Which I can easily prove with H_SNA_O.

I have now illustrated how to rewrite with transitivity and preservation. I leave the reader to see the rest of the associativity proof in the v. file. Because of the properties of eq_Nat, the preservation of all binary operations, the existence of a neutral element and associativity, natural numbers and addition form a monoid under eq_Nat.

I have proved two other lemmas regarding addition. The first one is about moving the successor from the left side directly to the right side i.e. S m' + n =N= m' (S n). This lemma is useful when proving addition to be commutative since the induction hypothesis is that for all natural numbers n, m' + n =N= n + m', and the sub-goal is S m' + n =N= n + S m'. One can move the successor over to n; use the induction hypothesis; and move the successor back over to m'.

5 Multiplication of natural numbers

Multiplication of two natural numbers m yields n is n summed m times, although one can also prove that it is n summed m times. Here is a more formal specification in Coq:

```
Definition specification_of_multiplication (NMul : Nat -> Nat -> Nat) :=
  (forall n : Nat,
     NMul O n =N= O)
    /\
  (forall NAdd : Nat -> Nat -> Nat,
    specification_of_addition NAdd ->
    forall m' n : Nat,
      NMul (S m') n =N= NAdd n (NMul m' n)).
```

Recursively, n is added to the rest m times with the termination being that zero times n is zero. That is, zero is left absorbent. One can prove the specification to be unique. When looking at the specification, one can see that the successor of zero will probably be left neutral for multiplication since (S O) multiplied by n will be n + O and O is neutral for addition. This is also the case as proved by unfolding multiplication twice and applying the lemma about zero being right neutral for addition. It can be shown through induction that (S O) is also right neutral for multiplication.

The base case is trivial since O is left absorbent. In the induction case, one has to prove that (S n') * (S O) =N= (S n') given that n' * (S O) =N= n'. One can move one addition of (S O) out to get (S O) + (n' * (S O)), which can be rewritten to (S O) + n' because of addition's preservation of eq_Nat and the induction hypothesis. One can then move the successor on Zero over to n' and remove the zero afterwards since it is left neutral for addition. That O is right absorbent can be proved in nearly the same way - the base case is trivial and in the induction case one can move one addition of O out; apply the induction hypothesis; and apply the lemma about zero being neutral for addition.

It can also be shown through induction that multiplication distributes over addition. That is, for all numbers m, n and p, (m + n) * p =N= (m * p) + (n * p) and p * (m + n) =N= (p * m) + (p * n). The first one is that multiplication is right distributive over addition and the other that it is left distributive. Because of the distributive property of multiplication, it can be shown through induction that multiplication is associative. In the base case one has to show that (O * b) * c =N= O * (b * c). This follows from the fact that zero is absorb3nt for multiplication. The induction case can be proved like this:

(S a' * b) * c =N= (b + (a' * b)) * c	by definition and preservation
=N= (b * c) + ((a' * b) * c)	by right distributivity
=N= (b * c) + (a' * (b * c))	by ind. hyp. and preservation
=N= (S a' * (b * c))	by symmetry and definition

The property of transitivity has also been used in all rewrites. The approach of proving it in Coq is the same. Because of the preservation of multiplication, eq_Nat being an equivalence relation, (S O) being neutral for multiplication and multiplication being associative, natural numbers and multiplication form a monoid under eq_Nat. The reader may have noticed that I have not shown much about how to prove this item in Coq. The reason is that you have access to the .v file and the proof style is the same as with addition. Transitivity and preservation is used to rewrite under eq_Nat. However, I feel that it is important to not only show how to prove it in Coq but also how to prove it by hand - the connection is important for human understanding.

6 Booleans and the equality relation on their set

A boolean is an inductive that can be one of two values, TRUE or FALSE. The all caps of the names is to prevent collisions with the props True and False. In Coq the inductive is specified as:

```
Inductive Boolean : Set :=
  | FALSE : Boolean
  | TRUE : Boolean.
```

Two booleans should be equal if they are both FALSE or both TRUE. The definition of equality between booleans can be defined with matching as follows:

```
Definition eq_Bool (p q : Boolean) :=
  match p with
      | TRUE => match q with
                  | TRUE => True
                  | FALSE => False
                end
      | FALSE => match q with
                   | TRUE => False
                   | FALSE => True
                 end
  end.
```

Four unfolding lemmas have been made - two which shows that two booleans can not be equivalent if they are not the same inductive; and two which shows that FALSE and FALSE is equivalent and that TRUE and TRUE is. That the equality relation is reflexive can easily be shown because of the last two unfolding lemmas. That the equality relation is symmetric follows from the fact that it is a contradiction to assume that TRUE and FALSE are equivalent. Transitivity follows from the same reasoning - if one assumes that a =B= b and b =B= c for some booleans a, b and c, then all three must be TRUE if one is and all three must be FALSE if one is. Therefore a =B= c. Because of these three properties, eq_Bool is an equivalence relation between booleans. It can also be shown that any binary operation on booleans will preserve eq_Bool. If a =B= c and b =B= d, then for any binary boolean operation f, f a b =B= f c d, since if a is TRUE (FALSE) then c is TRUE (FALSE) and the same with b and d.

7 Less than or equal

A natural number m is less than or equal to a natural number n if m has at most the same amount of successor constructors as n. This can be defined recursively in Coq as:

```
Definition specification_of_LEQ (LEQ : Nat -> Nat -> Boolean) :=
  (forall n : Nat,
     LEQ O n =B= TRUE)
  /\
  (forall m' : Nat,
     LEQ (S m') O =B= FALSE)
  /\
  (forall m' n' : Nat,
     LEQ (S m') (S n') =B= LEQ m' n').
```

That is, O is less than or equal to any number; no number other than O is less than or equal to O; and if (S m) is less than or equal to (S n) than m is less than or equal to n and vice versa. I have here used booleans and the equivalence relation on their set. If LEQ returns TRUE, the left side is less than or equal to the right side - FALSE if not. LEQ preserves eq_Bool :

```
Lemma about_preservation_of_Natural_LEQ :
  forall LEQ : Nat -> Nat -> Boolean,
    specification_of_Natural_LEQ LEQ ->
    forall a b c d : Nat,
      a =N= b ->
      c =N= d ->
      LEQ a c =B= LEQ b d.
```

This follows from induction on a. In the base case, b also has to be O, and since O is less than or equal to any natural the base case can be proved. In the induction case b also has to be the successor of a number, and both c and d must be O or the successors of some numbers. That LEQ (S a') O =B= LEQ (S b') O follows from the definition of LEQ. The other case can be proved since one can strip the successors off both sides in both eq_Nat and LEQ and apply the induction hypothesis. LEQ can also be proved to be unique. It is also the case that LEQ a b is either true or false. This follows from induction on a. The most important lemma is this:

```
Lemma destruct_LEQ :
  forall LEQ : Nat -> Nat -> Boolean,
    specification_of_Natural_LEQ LEQ ->
    forall m n : Nat,
    forall P : Prop,
      ((LEQ m n =B= TRUE -> P) /\ (LEQ m n =B= FALSE -> P)) -> P.
```

That is, if a property holds under any case of LEQ, then the property must hold in general. This follows from the fact that LEQ returns either TRUE or FALSE. This is a very useful lemma indeed. When proving something which involves LEQ, one can prove it to hold when LEQ returns FALSE and when LEQ returns TRUE. The reason for making this is that the destruct tactic in Coq uses polymorphic equality and I have not designed my specifications to work with this.

8 Minimum of natural numbers

The minimum of two natural numbers is the one of the two with the smallest amount of successor constructors. One could define minimum recursively in Coq as:

```
Definition specification_of_Minimum (Min : Nat -> Nat -> Nat) :=
  (forall n : Nat,
     Min O n =N= O)
  /\
  (forall m : Nat,
     Min m O =N= O)
  /\
  (forall m' n' : Nat,
     Min (S m') (S n') =N= S (Min m' n'))
```

However, I have chosen a path which is more intuitive:

```
Definition specification_of_Minimum (Min : Nat -> Nat -> Nat) :=
  forall LEQ : Nat -> Nat -> Boolean,
    specification_of_LEQ LEQ ->
    (forall m n : Nat,
       LEQ m n =B= TRUE ->
       Min m n =N= m)
    /\
    (forall m n : Nat,
       LEQ m n =B= FALSE ->
       Min m n =N= n).
```

This makes a bit more sense since if m $\leq$ n, then m is the minimum - n is the minimum if not. The funny thing is, however, that it is still important to prove that the second specification has the same properties as the first one.

That is, zero should be absorbent for both sides and one should be able to move the successor out from both sides and do a recursive call.

That zero is left absorbent follows from the fact that zero is less than or equal to any natural number. That zero is right absorbent follows from the fact that if the left side is zero, the minimum is zero, and if the left side is the successor of some number, then that number is not less than the right side (zero) and zero is returned. That the second specification has the last property of the first specification can be proved by destructing LEQ on the two numbers. This yields two cases, the first being:

Case : Leq (S a) (S b) =B= TRUE	(1)
Leq a b =B= TRUE	by definition of leq - (2)
Min (S a) (S b) =N= (S a)	by definition of Min and (1) - (3)
Min a b =N= a	by definition of Min and (2) - (4)
S (Min a b) =N= (S a)	by definition of eq_Nat and (4) - (5)
Min (S a) (S b) =N= S (Min a b)	by symmetry, transitivity, (3) and (5)

The proof for the other case is similar. I will now briefly explain how I have destructed Leq on (S a) and (S b) in Coq. Consider the sub-goal:

```
LEQ : Nat -> Nat -> Boolean
H_SL : specification_of_LEQ LEQ
Min : Nat -> Nat -> Nat
a : Nat
b : Nat
H_SM_L : forall m n : Nat, LEQ m n =B= TRUE -> Min m n =N= m
H_SM_R : forall m n : Nat, LEQ m n =B= FALSE -> Min m n =N= n
============================
 Min (S a) (S b) =N= S (Min a b)
```

Here I can apply

```
apply (destruct_LEQ LEQ H_SL (S a) (S b)).
```

which will leave me with the following sub-goal:

```
LEQ : Nat -> Nat -> Boolean
H_SL : specification_of_LEQ LEQ
Min : Nat -> Nat -> Nat
a : Nat
b : Nat
H_SM_L : forall m n : Nat, LEQ m n =B= TRUE -> Min m n =N= m
H_SM_R : forall m n : Nat, LEQ m n =B= FALSE -> Min m n =N= n
============================
 (LEQ (S a) (S b) =B= TRUE -> Min (S a) (S b) =N= S (Min a b)) /\
 (LEQ (S a) (S b) =B= FALSE -> Min (S a) (S b) =N= S (Min a b))
```

Hereafter I can use the split tactic to get two sub-goals, starting out by introducing the case hypothesis in both. But there is a way to make it act like the normal destruct tactic in one line. I remind the reader that several commands will be executed in one go when separated by semi-colons. Furthermore, if the split or destruct tactic appears in a group of commands, the following commands will be executed in all of the generated sub-goals. Therefore I can use this group of commands to generate two sub-goals where the case hypotheses are already introduced:

```
apply (destruct_LEQ LEQ H_SL (S a) (S b)); split; intro H_LEQ.
```

But I digress. The associativity of minimum follows from induction. In the base case zero will absorb everything. In the induction case we can assume that the two other variables are also successors of naturals, since if they are zero, they will be absorbent for everything. That leaves us with all three numberss being non-zero and the induction hypothesis. But because of the lemma above, one can move out the successors; apply the induction hypothesis; and move the successors back in. Now stands left to prove that there exists a neutral element for minimum - but consider this. A natural can not be the successor of itself - this follows by definition of eq_Nat and induction. In Coq the lemma is:

```
Lemma a_natural_number_can_not_be_the_successor_of_itself :
  forall n : Nat,
    ~(n =N= S n).
```

To prove this, do induction on n. In the base case we know from unfolding eq_Nat that O is not equal to the successor of anything. This will be an excellent short example of how to apply the contradiction tactic. Here is the base case in Coq:

```
H_Sn_n : O =N= S O
============================
 False
```

Now, there are two ways to prove this. The first one is simply to apply the following:

```
contradiction (unfold_eq_Nat_O_S O H_Sn_n).
```

Since the argument given to contradiction computes to the prop False, contradiction will prove the sub-goal since False implies everything. The other way to prove it is by applying the following in the hypothesis:

```
apply (unfold_eq_Nat_O_S O) in H_Sn_n.
```

The hypothesis will now become False and the sub-goal will be completed by using a single exact. The induction step is proved by the fact that (S n) =N= (S (S n)) implies n =N= (S n) - one can just apply this and the induction hypothesis.

It follows easily from induction that the minimum of any number and its successor is the number itself, i.e., Min n (S n) =N= n. These two lemmas can be used to prove that there is no upper bound on the natural numbers. Assume that there is some number M such that for all natural numbers a, Min M a =N= a. That is, M is bigger than everything. However, M must also be bigger than the successor of itself i.e. Min M (S M) =N= (S M). But we know from the second lemma that Min M (S M) =N= M. Because of the transitivity and symmetry of eq_Nat, we have assumed that M =N= (S M), but this is a contradiction according to the first lemma. The exciting thing is that this also implies that there exists no left neutral element for minimum! The neutral element would have to be the upper bound, but there is no upper bound.

It can also be proven that there is no right neutral element, but the non-existence of a left neutral element is enough to assure that natural numbers and minimum do not form a monoid under eq_Nat.

9 Maximum of natural numbers

The maximum of two natural numbers is the one which has the biggest amount of successor constructors. Turning it around, the maximum of two natural numbers is not the one which has the least amount of successor constructors. This can be specified in Coq as:

```
Definition specification_of_Maximum (Max : Nat -> Nat -> Nat) :=
  forall LEQ : Nat -> Nat -> Boolean,
    specification_of_LEQ LEQ ->
    (forall m n : Nat,
       LEQ m n =B= TRUE ->
       Max m n =N= n)
    /\
    (forall m n : Nat,
       LEQ m n =B= FALSE ->
       Max m n =N= m).
```

This is very close to the specification of minimum. The only differences in the specifications are that minimum returns the left side (right) if LEQ returns true (false) and maximum returns the right side (left) if LEQ returns true (false). It can be shown that maximum is unique. Furthermore, for all natural numbers m and n, Max (S m) (S n) =N= S (Max m n) - much like with minimum. The proof is almost identical to the proof about Min (S m) (S n) =N= S (Min m n). However, there is a notable difference between maximum and minimum, namely that maximum has a neutral element: Zero. That O is left neutral follows from the facts that O is less than or equal to any number and that no number other than zero is less than or equal to zero. In other words, the natural numbers have a lower bound.

The proof for associativity of maximum is much like the the proof for associativity of minimum. The proof is done by induction. In the base case O will be neutral for everything. In the induction case one can assume that the other two variables are not zero since zero will be neutral for everything. But if all the variables are successors of numbers, one can move out the successors; apply the induction hypothesis; and move the successors back in. That natural numbers and maximum form a monoid under eq_Nat follows from the facts that any binary operation on naturals preserves eq_Nat, that eq_Nat is an equivalence relation, that there exists a neutral element for maximum, and that maximum is associative. That natural numbers and maximum form a monoid while the numbers and minimum don't is related to the facts that the natural numbers have a lower bound but no upper bound. One would probably be able to prove that integers and maximum do not form a monoid and that a finite subset of natural numbers and minimum form a monoid, but I digress.

10 Boolean Conjunction and Disjunction

I have earlier shown the definition of booleans and an equality relation on their set. The binary operation conjunction, also called 'and', is defined to return TRUE iff. both input booleans are TRUE while the binary operation disjunction, also called 'or', is defined to return TRUE iff. at least one input boolean is TRUE. If conjunction and disjunction do not return TRUE, they return FALSE. In Coq, the two specifications are:

```
Definition specification_of_conjunction
          (conj : Boolean -> Boolean -> Boolean) :=
  (forall b : Boolean,
     conj FALSE b =B= FALSE)
  /\
  (forall b : Boolean,
     conj TRUE b =B= b).
```

```
Definition specification_of_disjunction
          (disj : Boolean -> Boolean -> Boolean) :=
  (forall b : Boolean,
     disj FALSE b =B= b)
  /\
  (forall b : Boolean,
     disj TRUE b =B= TRUE).
```

Since booleans is a finite set containing only two values, the following properties can easily be proved by destructing the booleans and applying properties from the specification:

TRUE is left and right neutral for conjunction.
FALSE is left and right absorbant for conjunction.
FALSE is left and right neutral for disjunction.
TRUE is left and right absorbant for disjunction.

Both operations are also associative. The proofs are however a bit more involved since one has to use the properties of eq_Bool: it is symmetric, transitive and preserved by all binary boolean operations. However there is no 'good ideas' - the proofs consists of destructing, rewriting and using the specifications. It is seen in the .v file that booleans form monoids under eq_Bool with both conjunction and disjunction. However, I will give an example of how to do a rewrite. Let us consider the proof for associativity of conjunction. By destructing the left most boolean, one has to prove the sub-goal:

```
conj : Boolean -> Boolean -> Boolean
H_SC : specification_of_conjunction conj
b : Boolean
c : Boolean
============================
 conj (conj FALSE b) c =B= conj FALSE (conj b c)
```

Notice that (conj FALSE b) =B= FALSE because of FALSE being left absorbent. The first step will be to rewrite (conj (conj FALSE b) c) into (conj FALSE c). This can be done by first applying

```
apply (eq_Bool_is_transitive (conj (conj FALSE b) c) (conj FALSE c)).
```

which will generate two sub-goals:

```
conj (conj FALSE b) c =B= conj FALSE c

subgoal 2 (ID 2711) is:
 conj FALSE c =B= conj FALSE (conj b c)
```

Since it has been proven that eq_Bool is preserved by all binary boolean operations, the first sub-goal can be proved by applying

```
apply (Dot_Boolean_preserves_eq_Bool).
```

which will generate two sub-goals from the first sub-goal:

```
conj FALSE b =B= FALSE

subgoal 2 (ID 2713) is:
 c =B= c
```

The first of these follows from FALSE being left absorbent and the second follows from eq_Bool being reflexive. Now one is left to prove

```
conj FALSE c =B= conj FALSE (conj b c)
```

The left side has been rewritten.

11 2X2 Matrices with natural numbers

A 2X2 matrix with natural numbers is a constructor on four natural numbers. In Coq this can be defined as:

```
Inductive M22 : Set :=
  | m22 : Nat -> Nat -> Nat -> Nat -> M22.
```

Furthermore, two matrices are equal if the points in the first matrix are equivalent to the points in the second matrix in the right order:

```
Definition eq_Matrix (A B : M22) :=
  match A with
      | m22 a11 a12 a21 a22 => match B with
                                 | m22 b11 b12 b21 b22 => a11 =N= b11 /\
                                                          a12 =N= b12 /\
                                                          a21 =N= b21 /\
                                                          a22 =N= b22
                               end
  end.
```

That eq_Matrix is an equivalence relation follows easily from destructing matrices and eq_Nat being an equivalence relation. However I have been unable to prove that all binary 2x2 matrix operations preserve eq_Matrix, but I have an idea of how to do it. It would just require an immense amount of induction and destruction on the naturals in the matrices. The approach would be the same as when showing that any binary operation on natural numbers preserves eq_Nat - there would just be very many variables.

12 Addition of 2X2 Matrices with natural numbers

Adding two matrices should yield a new matrix where the natural numbers of the two matrices have been added in the right locations:

```
Definition specification_of_Matrix_Addition
          (NMAdd : M22 -> M22 -> M22) :=
  forall NAdd : Nat -> Nat -> Nat,
    specification_of_addition NAdd ->
    (forall a11 a12 a21 a22 b11 b12 b21 b22 : Nat,
       NMAdd (m22 a11 a12 a21 a22) (m22 b11 b12 b21 b22) =M=
       m22 (NAdd a11 b11) (NAdd a12 b12) (NAdd a21 b21) (NAdd a22 b22)).
```

This specification is unique. The first important thing to prove is that matrix addition preserves eq_Matrix. The first notable sub-goal of this proof is:

```
NAdd : Nat -> Nat -> Nat
  H_SA : specification_of_addition NAdd
  NMAdd : M22 -> M22 -> M22
  H_SMA' : forall a11 a12 a21 a22 b11 b12 b21 b22 : Nat,
           NMAdd (m22 a11 a12 a21 a22) (m22 b11 b12 b21 b22) =M=
           m22 (NAdd a11 b11) (NAdd a12 b12) (NAdd a21 b21) (NAdd a22 b22)
  ============================
   forall a b c d : M22, a =M= b -> c =M= d -> NMAdd a c =M= NMAdd b d
```

First I have introduced the variables and the hypotheses; then I have destructed all the matrices; and finally I have used H_SMA' to add all the variables together. This yields the following sub-goal:

```
NAdd : Nat -> Nat -> Nat
  H_SA : specification_of_addition NAdd
  NMAdd : M22 -> M22 -> M22
  H_SMA' : forall a11 a12 a21 a22 b11 b12 b21 b22 : Nat,
           NMAdd (m22 a11 a12 a21 a22) (m22 b11 b12 b21 b22) =M=
           m22 (NAdd a11 b11) (NAdd a12 b12) (NAdd a21 b21) (NAdd a22 b22)
  a11 : Nat
  a12 : Nat
  a21 : Nat
  a22 : Nat
  b11 : Nat
  b12 : Nat
  b21 : Nat
  b22 : Nat
  c11 : Nat
  c12 : Nat
  c21 : Nat
  c22 : Nat
  d11 : Nat
  d12 : Nat
  d21 : Nat
  d22 : Nat
  H_ab_11 : a11 =N= b11
  H_ab_12 : a12 =N= b12
  H_ab_21 : a21 =N= b21
  H_ab_22 : a22 =N= b22
  H_cd_11 : c11 =N= d11
  H_cd_12 : c12 =N= d12
  H_cd_21 : c21 =N= d21
  H_cd_22 : c22 =N= d22
  ============================
   m22 (NAdd a11 c11) (NAdd a12 c12) (NAdd a21 c21) (NAdd a22 c22) =M=
   m22 (NAdd b11 d11) (NAdd b12 d12) (NAdd b21 d21) (NAdd b22 d22)
```

There are many variables. The approach would be the same no matter how big or small the matrices would be, but the bigger the matrices, the more variables and destructs. Unfolding eq_Matrix in the sub-goal yields the sub-goal:

```
NAdd a11 c11 =N= NAdd b11 d11 /\
NAdd a12 c12 =N= NAdd b12 d12 /\
NAdd a21 c21 =N= NAdd b21 d21 /\
NAdd a22 c22 =N= NAdd b22 d22
```

The hypotheses stays the same. Since addition preserves eq_Nat, this sub-goal can be proved by splitting and using the hypotheses together with the preservation property. In general things can get a bit harder to prove when not using polymorphic equality and the rewrite tactic. If I had used polymorphic equality, I would not have had to prove that matrix addition preserved it since polymorphic equality is designed to be preserved by everything. Furthermore, every time I have to do a rewrite I have to apply transitivity and preservation lemmas en masse.

That the all-zero matrix is both left and right neutral for matrix addition follows by definition, destructs and zero being neutral for addition. Associativity of matrix addition can be proved through destructs, preservation, transitivity, symmetry and addition being associative. Induction is not necessary in any of the proofs since the properties are essentially corollaries of addition where one destructs matrices; rewrites using transitivity and preservation; and unfolds eq_Matrix. All in all, 2X2 matrices and 2X2 matrix addition form a monoid under eq_Matrix.

13 Multiplication of 2X2 matrices with natural numbers

Matrix multiplication can be defined as the following in Coq:

```
Definition specification_of_Matrix_Multiplication
          (NMMul : M22 -> M22 -> M22) :=
  forall NAdd : Nat -> Nat -> Nat,
    specification_of_addition NAdd ->
    forall NMul : Nat -> Nat -> Nat,
      specification_of_multiplication NMul ->
      forall a11 a12 a21 a22 b11 b12 b21 b22 : Nat,
        NMMul (m22 a11 a12 a21 a22) (m22 b11 b12 b21 b22) =M=
        (m22 (NAdd (NMul a11 b11) (NMul a12 b21))
             (NAdd (NMul a11 b12) (NMul a12 b22))
             (NAdd (NMul a21 b11) (NMul a22 b21))
             (NAdd (NMul a21 b12) (NMul a22 b22))).
```

One of the reasons for this definition can be found in many linear algebra books - it is about linear transformations, but this is out of scope for this project. Matrix multiplication is an unique specification. I have proved that it preserves eq_Matrix; that there exists a neutral element for it; and that it is associative.

The fact that it preserves eq_Matrix stems from the fact that both addition and multiplication preserves eq_Nat. The proof consists of destructing the four matrices, unfolding the matrix multiplication, splitting each point and applying the lemma about preservation of eq_Nat a good amount of times.

The neutral element is the 'identity matrix' (m22 (S O) O O (S O)) which one can realize by considering that

```
(m22 (S O) O O (S O)) *M (m22 a11 a12 a21 a22) =M=
(m22 ((S O) * a11 + O * a21) ((S O) * a12 + O * a22)
     (O * a11 + (S O) * a21) (O * a12 + (S O) * a22))
```

By unfolding eq_Matrix one has to prove that

```
((S O) * a11 + O * a21) =N= a11 /\
((S O) * a12 + O * a22) =N= a12 /\
(O * a11 + (S O) * a21) =N= a21 /\
(O * a12 + (S O) * a22) =N= a22
```

to prove that the identity matrix is left neutral. This can be proved since (S O) is neutral for multiplication and O is absorbent for multiplication and neutral for addition - of course one also has to rewrite a lot with transitivity and preservation. In the same way the identity matrix can be proved to be right neutral for matrix multiplication. To prove that matrix multiplication is associative, one can unfold all the matrices, rewrite using preservation and transitivity and apply the specification to get the following sub-goal:

```
m22
    (NAdd (NMul (NAdd (NMul a11 b11) (NMul a12 b21)) c11)
       (NMul (NAdd (NMul a11 b12) (NMul a12 b22)) c21))
    (NAdd (NMul (NAdd (NMul a11 b11) (NMul a12 b21)) c12)
       (NMul (NAdd (NMul a11 b12) (NMul a12 b22)) c22))
    (NAdd (NMul (NAdd (NMul a21 b11) (NMul a22 b21)) c11)
       (NMul (NAdd (NMul a21 b12) (NMul a22 b22)) c21))
    (NAdd (NMul (NAdd (NMul a21 b11) (NMul a22 b21)) c12)
       (NMul (NAdd (NMul a21 b12) (NMul a22 b22)) c22)) =M=
  m22
    (NAdd (NMul a11 (NAdd (NMul b11 c11) (NMul b12 c21)))
       (NMul a12 (NAdd (NMul b21 c11) (NMul b22 c21))))
    (NAdd (NMul a11 (NAdd (NMul b11 c12) (NMul b12 c22)))
       (NMul a12 (NAdd (NMul b21 c12) (NMul b22 c22))))
    (NAdd (NMul a21 (NAdd (NMul b11 c11) (NMul b12 c21)))
       (NMul a22 (NAdd (NMul b21 c11) (NMul b22 c21))))
    (NAdd (NMul a21 (NAdd (NMul b11 c12) (NMul b12 c22)))
       (NMul a22 (NAdd (NMul b21 c12) (NMul b22 c22))))
```

This can be very hard to read without infix notation, but I do not know how to make infix notations for the binary operations since they are not designed by fix-points/definitions but by specifications. Unfolding eq_Matrix yields the four sub-goals

```
(NAdd (NMul (NAdd (NMul a11 b11) (NMul a12 b21)) c11)
       (NMul (NAdd (NMul a11 b12) (NMul a12 b22)) c21)) =N=
(NAdd (NMul a11 (NAdd (NMul b11 c11) (NMul b12 c21)))
       (NMul a12 (NAdd (NMul b21 c11) (NMul b22 c21))))
/\
(NAdd (NMul (NAdd (NMul a11 b11) (NMul a12 b21)) c12)
       (NMul (NAdd (NMul a11 b12) (NMul a12 b22)) c22)) =N=
(NAdd (NMul a11 (NAdd (NMul b11 c12) (NMul b12 c22)))
       (NMul a12 (NAdd (NMul b21 c12) (NMul b22 c22))))
/\
(NAdd (NMul (NAdd (NMul a21 b11) (NMul a22 b21)) c11)
       (NMul (NAdd (NMul a21 b12) (NMul a22 b22)) c21)) =N=
(NAdd (NMul a21 (NAdd (NMul b11 c11) (NMul b12 c21)))
       (NMul a22 (NAdd (NMul b21 c11) (NMul b22 c21))))
/\
(NAdd (NMul (NAdd (NMul a21 b11) (NMul a22 b21)) c12)
       (NMul (NAdd (NMul a21 b12) (NMul a22 b22)) c22)) =N=
(NAdd (NMul a21 (NAdd (NMul b11 c12) (NMul b12 c22)))
       (NMul a22 (NAdd (NMul b21 c12) (NMul b22 c22))))
```

By taking a closer look, one can see that the equivalences have a common denominator - they would all be true if the following lemma could be proved:

```
Lemma about_distributivity_in_each_matrix_point :
  forall NAdd : Nat -> Nat -> Nat,
    specification_of_Natural_addition NAdd ->
    forall NMul : Nat -> Nat -> Nat,
      specification_of_Natural_multiplication NMul ->
      forall a11 b11 a12 b21 c11 b12 b22 c21 : Nat,
        NAdd (NMul (NAdd (NMul a11 b11) (NMul a12 b21)) c11)
             (NMul (NAdd (NMul a11 b12) (NMul a12 b22)) c21) =N=
        NAdd (NMul a11 (NAdd (NMul b11 c11) (NMul b12 c21)))
             (NMul a12 (NAdd (NMul b21 c11) (NMul b22 c21))).
```

Luckily this can be proven, and therefore matrix multiplication is associative. However the lemma above requires an immense amount of applications of the properties of eq_Nat and the properties of multiplication and addition. This lemma has the longest proof in the project. It really made me reconsider why I was not using polymorphic equality, but the result was satisfying. If I had used the in-built multiplication, addition and polymorphic equality, I would have been able to prove the lemma above with a single 'ring'. However, supposedly one should be able to define tactics and etc. in Coq - we have not seen much about this in the lectures except for the unfold tactic, but one could probably devise a tactic that would be able to rewrite and prove arithmetic using my specifications. Anyhow, since the matrix multiplication preserves eq_Matrix; since eq_Matrix is an equivalence relation; since the identity matrix is neutral for matrix multiplication; and since matrix multiplication is associative, 2X2 matrices and 2X2 matrix multiplication form a monoid under eq_Matrix.

14 Polymorphic endofunctions

An endofunction is a function which has the same domain as co-domain. Two endofunctions with the same domain are equal if they, under some equality relation, return related output on related input. This can be formalized in Coq as:

```
Definition eq_PE (T : Type)
                 (eqT : T -> T -> Prop)
                 (f g : T -> T) :=
    forall x y : T,
       eqT x y -> eqT (f x) (g y).
```

Furthermore, I will only consider endofunctions which are referentially transparent, i.e., if two elements are related, then the outputs yielded from an endofunction on the elements should be related. This has been specified as a more general property in Coq:

```
Property referential_transparency_for_functions :
  forall (T1 : Type)
         (T2 : Type)
         (eq_T1 : T1 -> T1 -> Prop)
         (eq_T2 : T2 -> T2 -> Prop),
    specification_of_an_equivalence_relation T1 eq_T1 ->
    specification_of_an_equivalence_relation T2 eq_T2 ->
    forall (f : T1 -> T2)
           (x1 y1 : T1),
      eq_T1 x1 y1 ->
      eq_T2 (f x1) (f y1).
Admitted.
```

I have not admitted this because I can not prove it - in fact, it can not be proven. I have just admitted it because I am assuming that I am only proving properties of functions which are referentially transparent. It can now be proved that for any type T and any equality relation eqT at T, (eq_PE T eqT) is an equivalence relation between referentially transparent endofunctions from T to T if eqT is an equivalence relation at type T i.e.

```
Lemma eq_PE_is_an_equivalence_relation :
  forall T : Type,
    forall (eqT : T -> T -> Prop),
      eqT_is_an_equivalence_relation T eqT ->
      eqT_is_an_equivalence_relation (T -> T) (eq_PE T eqT).
```

All three properties can be shown by unfolding eq_PE and using the hypotheses about eqT being an equivalence relation. To prove reflexivity requires the assumed referential transparency.

15 Polymorphic endofunctions and composition

Composition of two endofunctions is defined to return a function which first applies the second function to some input and then applies the first function to the result of that. My specification in Coq is:

```
Definition specification_of_Compose_Polymorphic
           (T : Type)
           (eqT : T -> T -> Prop)
           (PCom : (T -> T) -> (T -> T) -> (T -> T)) :=
  forall f g : T -> T,
    eq_PE T eqT (PCom f g) (fun x => f (g x)).
```

It can be proved that composition will preserve (eq_PE T eqT) if eqT is an equivalence relation, i.e.

```
Lemma Compose_Polymorphic_preserves_eq_PE :
  forall T : Type,
    forall eqT : T -> T -> Prop,
      specification_of_an_equivalence_relation T eqT ->
      forall PCom : (T -> T) -> (T -> T) -> (T -> T),
        specification_of_Compose_Polymorphic T eqT PCom ->
        specification_of_preservation (T -> T) (eq_PE T eqT) PCom.
```

By using the specification of composition and the properties of eq_PE, one arrives at the following sub-goal:

```
T : Type
eqT : T -> T -> Prop
H_eqT : specification_of_an_equivalence_relation T eqT
PCom : (T -> T) -> (T -> T) -> T -> T
H_PCom : forall f g : T -> T, eq_PE T eqT (PCom f g) (fun x : T => f (g x))
a : T -> T
b : T -> T
c : T -> T
d : T -> T
H_ab : forall x y : T, eqT x y -> eqT (a x) (b y)
H_cd : forall x y : T, eqT x y -> eqT (c x) (d y)
x : T
y : T
H_xy : eqT x y
H_refl : forall a : T, eqT a a
H_symm : forall a b : T, eqT a b -> eqT b a
H_trans : forall x y z : T, eqT x y -> eqT y z -> eqT x z
============================
 eqT (a (c x)) (b (d y))
```

However, since eqT is reflexive, (c x) must be equal to itself, and because of transitivity and H_ab, one can get to the sub-goal

```
eqT (b (c x)) (b (d y))
```

Since the functions are assumed to be referentially transparent, the above will be the case if

```
eqT (c x) (d x)
```

Since x is equal to y, H_cd implies the above.

That the identity function is left and right neutral for composition if eqT is an equivalence relation follows easily from using transitivity and the specification of composition to rewrite and unfolding eq_PE. E.g.

```
eq_PE T eqT (PCom (fun i : T => i) g) g
```

can be proved by rewriting the sub-goal to

```
eq_PE T eqT (fun x : T => g x) g
```

which by unfolding eq_PE yields

```
forall x y : T, eqT x y -> eqT (g x) (g y)
```

which is proved by assuming g to be referentially transparent. The proof is nearly the same for the identity function being right neutral. It can also be proved that composition is associative if eqT is an equivalence relation:

```
Lemma Compose_Polymorphic_is_associative :
  forall T : Type,
  forall eqT : T -> T -> Prop,
    specification_of_an_equivalence_relation T eqT ->
    forall PCom : (T -> T) -> (T -> T) -> (T -> T),
      specification_of_Compose_Polymorphic T eqT PCom ->
      specification_of_associativity (T -> T) (eq_PE T eqT) PCom.
```

Denoting (eq_PE T eqT) as =F= and composition as >>, this is true since

(f>>g)>>h =F= (fun x : T → f (g x)) >>h
by definition of composition and preservation of (eq_PE T eqT)
(fun x : T → f (g x)) >>h =F= (fun y : T → f (g (h y)))
by definition of composition
(fun y : T → f (g (h y))) =F= f>>(fun y : T → g (h y))
by definition of composition and symmetry of (eq_PE T eqT)
f>>(fun y : T → g (h y)) =F= f>>(g>>h)
by definition of composition, preservation and symmetry of (eq_PE T eqT)

Summing it up, for any type T and equality relation at T: if the relation is an equivalence relation, then referentially transparent endofunctions at T and composition with T and the equality relation form a monoid under (eq_PE T eqT).

16 Polymorphic lists

A list of some type T is either empty, also called nil, or an element of type T appended to another list. In Coq this can be described as the inductive

```
Inductive ListT (T : Type) :=
  | Nil : ListT T
  | Cons : T -> ListT T -> ListT T.
```

Given a type T and a equality relation at T, I define two lists to be equal if their elements are related in the right order - more implicitly it is also required that they have the same number of elements:

```
Fixpoint eq_ListT (T : Type)
                  (eqT : T -> T -> Prop)
                  (xs ys : ListT T) :=
  match xs with
      | Nil => match ys with
                  | Nil => True
                  | Cons y ys' => False
               end
      | Cons x xs' => match ys with
                        | Nil => False
                        | Cons y ys' => eqT x y /\ eq_ListT T eqT xs' ys'
                      end
  end.
```

Four unfolding lemmas can be deducted from this: nil and nil are equivalent; a list with at least one element is never equivalent to nil (from either side); and two lists are equal if their top element is equal according to the given equality relation and the rest of the lists are equal (the converse is also true).

If the given equality relation between the elements is an equivalence relation, the list relation can be shown to be an equivalence relation through induction. The induction principle for lists is to show that a property holds for nil and that the property holds for a list consisting of a element on top of a smaller list if the property holds for the smaller list. To give a better picture:

```
ListT_ind
     : forall (T : Type) (P : ListT T -> Prop),
       P (Nil T) ->
       (forall (t : T) (l : ListT T), P l -> P (Cons T t l)) ->
       forall l : ListT T, P l
```

To show how induction on lists can be used, consider the following sub-goal in the proof about the equality being reflexive given that the element relation is:

```
eq_ListT T eqT a a
```

Here I can do induction on a. The base case is trivial since nil is equal to nil. The induction case is:

```
IHa' : eq_ListT T eqT a' a'
============================
 eq_ListT T eqT (Cons T a1 a') (Cons T a1 a')
```

Using the unfolding lemmas of list equivalence, one has to prove that a1 is equivalent to a1 with the element relation and that a' and a' are list equivalent. The first part follows from the element relation being assumed to be reflexive and the second part follows from the induction hypothesis. The proofs for symmetry and transitivity have the same approach.

17 Polymorphic list append

Appending polymorphic lists is similar to addition. But instead of applying all the successors of the left argument to the right argument, one wants to append all the elements of the left argument to the right argument. In Coq:

```
Definition specification_of_List_Append (T : Type)
                                    (eqT : T -> T -> Prop)
                                    (App : ListT T -> ListT T -> ListT T) :=
  (forall ys : ListT T,
     eq_ListT T eqT (App (Nil T) ys) ys) /\
  (forall x : T,
     forall xs' ys : ListT T,
       eq_ListT T eqT (App (Cons T x xs') ys) (Cons T x (App xs' ys))).
```

This specification is unique and I have also shown through induction that it preserves list equality given that the element relation is an equivalence relation. That nil is left neutral for append follows from the specification. It can however also be proved that nil is right neutral through induction. It is trivial that appending nil and nil yields nil. In the induction case one has to prove

```
IHxs' : eq_ListT T eqT (App xs' (Nil T)) xs'
============================
 eq_ListT T eqT (App (Cons T x xs') (Nil T)) (Cons T x xs')
```

The specification of append and transitive properties of list equivalence can rewrite this to

```
eq_ListT T eqT (Cons T x (App xs' (Nil T))) (Cons T x xs')
```

which can easily be proven since the element relation is assumed to be reflexive and because of the induction hypothesis. The point of the proof is that all the elements appended from the list to nil will be equal to the elements in the list in the right order. That list append is associative follows from appends ability to move elements 'to the top', transitivity, preservation and induction. All in all, polymorphic lists form a monoid with list append if the underlying element relation is an equivalence relation.

18 Polymorphic lazy lists

A lazy list is much like a regular list - except it can be infinite. That is, a lazy list can consists of only cons and no nil. In Coq this is the co-inductive type:

```
CoInductive LazyListT (T : Type) :=
  | LNil : LazyListT T
  | LCons : T -> LazyListT T -> LazyListT T.
```

One of the key differences between inductive and co-inductive types is that inductive types are constructed from some base case while co-inductive types have properties that tells one about how to deconstruct them. In Coq another difference is that the constructors in a co-inductive type are guarded i.e. they are not forced to be executed instantly when given explicit parameters. Inductive types have induction because of their constructive properties, while co-inductive types has co-induction which relies on how the types can be deconstructed.

I will now define a co-inductive equality relation on two lazy lists. If two lazy lists are nil, they are equal. But here is something that is the dual of equality on inductive lists: if two elements are equal and two lazy lists are equal, then the two lazy lists with the respective elements on top are equal while if two inductive lists are equal, their top elements are equal and the rest of the lists are equal. The converse of both cases can in this case be proved to be true, but the point is that with induction and lists something which can be constructed is deconstructed to prove, while with co-induction and lazy lists something which has to be deconstructed is constructed to prove. Anyways, the equality relation between lazy lists is called bisimilarity. Here it is in Coq:

```
CoInductive bisimilar_LazyListT (T : Type)
                            (eqT : T -> T -> Prop) : LazyListT T ->
                                                     LazyListT T -> Prop :=
| Bisimilar_Nil : bisimilar_LazyListT T eqT (LNil T) (LNil T)
| Bisimilar_Cons :
   forall (x y : T) (xs ys : LazyListT T),
     eqT x y ->
     bisimilar_LazyListT T eqT xs ys ->
     bisimilar_LazyListT T eqT (LCons T x xs) (LCons T y ys).
```

To give a better look into co-induction, I will illustrate it by showing that bisimilarity is reflexive. Consider the sub-goal

```
T : Type
eqT : T -> T -> Prop
H_eqT : specification_of_an_equivalence_relation T eqT
============================
 forall a : LazyListT T, bisimilar_LazyListT T eqT a a
```

If the lists were inductive one would introduce the variable and do induction now. However a lazy list does not necessarily have a base case since it can consist solely of cons. One can do co-induction instead. All lemmas and the like in Coq are really functions – take a look at the Curry-Howard correspondance – and proofs using co-induction will involve co-fixpoints. Co-fixpoints are somewhat like normal fixpoints, but they are, like co-inductive constructors, guarded in the sense that each step of the co-recursive step of the co-fixpoint is not automatically computed - they have to be decomposed. One cool thing about co-fixpoints is that one can define something which involves an infinite amount of applications of functions to something without Coq going into an infinite loop. However, let us return. The cofix tactic in Coq will prepare the rest of a proof to be a co-fixpoint. Using the cofix now in this proof will yield the sub-goal

```
T : Type
eqT : T -> T -> Prop
H_eqT : specification_of_an_equivalence_relation T eqT
CoIH : forall a : LazyListT T, bisimilar_LazyListT T eqT a a
============================
 forall a : LazyListT T, bisimilar_LazyListT T eqT a a
```

The co-induction hypothesis looks ready to be applied, but alas, it cannot. Coq won't accept the proof. This is because I have started a co-recursive proof - the co-fixpoint would not be guarded if I applied the hypothesis immediately. To prove something by co-induction one has to transform the sub-goal into something where an application of the co-induction hypothesis will be guarded.

I can start by introducing the list. Hereafter I can consider that a and a are bisimilar if a is nil; or if the top elements of a and a are equal while the rest of a and a are bisimilar. By destructing a, I can easily prove that nil and nil are equal. I arrive at the following sub-goal:

```
bisimilar_LazyListT T eqT (LCons T a1 a') (LCons T a1 a')
```

This will be true if a1 and a1 are equal and a' is bisimilar with a' because of the definition of bisimilarity. By applying Bisimilar_Cons, I can easily prove that a1 is equal to a1 since eqT is assumed to be an equivalence relation. I then arrive at the sub-goal

```
CoIH : forall a : LazyListT T, bisimilar_LazyListT T eqT a a
============================
 bisimilar_LazyListT T eqT a' a'
```

I can now apply the co-induction hypothesis and the proof will go through since the hypothesis is guarded by the co-inductive constructor Bisimilar_Cons. I will not go into more detail about co-induction. The proof about bisimilarity being symmetric has the same approach as the reflexivity proof. However, let us consider the proof for bisimilarity being transitive. Here I will take use of 'remember' and 'discriminate' - two Coq tactics I haven't used yet.

Nil is not bisimilar to anything that is not nil and vice versa. By hand, this follows from the fact that two lists are bisimilar if they are both nil or the top elements are equal while the rest of the lists are bisimilar. If nil and not nil are bisimilar, then not nil must be nil, which is a contradiction; or nil must have a top element, which it do not. Consider the sub-goal

```
H_nil_and_cons : bisimilar_LazyListT T eqT (LNil T) (LCons T y ys')
============================
 False
```

Here I can 'remember' nil as xs and the cons as ys. That is, I replace all occurrences of nil with xs and all occurrences of cons with ys while remembering what they really are:

```
xs : LazyListT T
H_xs : xs = LNil T
ys : LazyListT T
H_ys : ys = LCons T y ys'
H_nil_and_cons : bisimilar_LazyListT T eqT xs ys
============================
 False
```

Since bisimilarity is co-inductive, I can now destruct it. Coq will know that I have assumed both xs and ys to be nil or both to have top elements which are equal while the rest of the two are bisimilar. This is the first case where I have to prove False:

```
H_xs : LNil T = LNil T
H_ys : LNil T = LCons T y ys'
============================
 False
```

ys is assumed to be nil, but I have remembered it as being not nil. The discriminate tactic on the hypothesis will solve the sub-goal. The discriminate tactic looks at the constructors on both sides of the Leibniz equality - if their constructors are not the same it is absurd to assume that they are equal. The second case can also be discriminated since one assumed xs to be not nil while it has also be assumed to be nil.

It can also be shown that if two lazy lists are bisimilar and not nil their top elements must be equal while the rest of the lists must be bisimilar. Here I will use another tactic, injection. Injection looks at the constructors of both sides of a Leibniz equality in a hypothesis. If both constructors are the same their arguments on both sides must be respectively Leibniz equal. In the converse lemma one can first use remember and destruct the assumed bisimilarity of the two non nil lists. Since they are both not nil, the first case, Bisimilar_Nil, can be discriminated. This leaves one with the sub-goal

```
T : Type
eqT : T -> T -> Prop
x : T
y : T
xs' : LazyListT T
ys' : LazyListT T
x' : T
xs'' : LazyListT T
H_xs : LCons T x' xs'' = LCons T x xs'
y' : T
ys'' : LazyListT T
H_ys : LCons T y' ys'' = LCons T y ys'
H_xy' : eqT x' y'
H_xsys' : bisimilar_LazyListT T eqT xs'' ys''
============================
 eqT x y /\ bisimilar_LazyListT T eqT xs' ys'
```

It is assumed that x' is equal to y' while xs" is bisimilar to ys". But because of injection, x' is x; y' is y; xs" is xs'; and ys" is ys'. By using injection, rewriting, and splitting, one can easily solve the sub-goal.

That bisimilarity is transitive follows from the two lemmas above. If three lists are bisimilar, they must all be nil or they must all be not nil. If they are all nil, then the first is indeed bisimilar to the third. If they are all not nil their top elements must be equal - the transitivity follows from co-induction guarded behind an application of Bisimilar_Cons. Finally, bisimilarity is an equivalence relation if the equality relation between the elements is an equivalence relation.

19 Polymorphic lazy append

Appending two lazy lists can almost be defined the same way as appending two inductive lists. However a lazy list can be infinite - appending two lazy lists completely could be impossible. Luckily the constructors are guarding. My initial idea was to make the following specification:

```
Definition specification_of_Lazy_Append
          (T : Type)
          (eqT : T -> T -> Prop)
          (LApp : LazyListT T -> LazyListT T -> LazyListT T) :=
  (forall ys : LazyListT T,
     bisimilar_LazyListT T eqT (LApp (LNil T) ys) ys)
  /\
  (forall x : T,
     forall xs' ys : LazyListT T,
       bisimilar_LazyListT T eqT (LApp (LCons T x xs') ys)
                                 (LCons T x (LApp xs' ys))).
```

It turned out that I would not be able to do co-induction with this specification. E.g. in the proof for right neutrality of nil, one gets to the following sub-goal after destructing the other list and applying Bisimilar_Nil:

```
T : Type
eqT : T -> T -> Prop
H_eqT : specification_of_an_equivalence_relation T eqT
LApp : LazyListT T -> LazyListT T -> LazyListT T
H_SA : specification_of_Lazy_Append T eqT LApp
CoIH : forall xs : LazyListT T,
        bisimilar_LazyListT T eqT (LApp xs (LNil T)) xs
x : T
xs' : LazyListT T
H_SA_cons : forall (x : T) (xs' ys : LazyListT T),
             bisimilar_LazyListT T eqT (LApp (LCons T x xs') ys)
               (LCons T x (LApp xs' ys))
============================
 bisimilar_LazyListT T eqT (LApp (LCons T x xs') (LNil T)) (LCons T x xs')
```

Here I would like to use transitivity to move cons x on xs' out of append on the left side such that the sub-goal would be

```
bisimilar_LazyListT T eqT (LCons T x (LApp xs' (LNil T))) (LCons T x xs')
```

Hereafter I would be able to apply Bisimilar_Cons, prove that the top elements are equal and use the coinduction hypothesis to prove for xs'. However, my transitivity lemma does not guard the coinduction hypothesis. I can therefore not prove this with my knowledge. For some reason, eq_ind, the lemma applied when using the rewrite tactic, will guard a coinduction hypothesis. Therefore I resorted to creating a concrete implementation of lazy append. This is lazy append as a co-fixpoint:

```
CoFixpoint LApp (T : Type) (xs ys : LazyListT T) :=
  match xs with
      | LNil => ys
      | LCons x xs' => LCons T x (LApp T xs' ys)
  end.
```

Since this is a co-fixpoint, I will not be able to unfold it directly in my proofs. Luckily there is a simple trick to unfold one step of any lazy list including the result of lazy append. The idea is to decompose a lazy list:

```
Definition decompose_LazyListT (T : Type) (xs : LazyListT T) :=
  match xs with
      | LNil => LNil T
      | LCons x xs' => LCons T x xs'
  end.
```

It can be shown easily that the result of decomposing a lazy list is Leibniz equal to the list itself. This can be used to make two unfolding lemmas about lazy append and Leibniz equality:

```
LApp T (LNil T) ys = ys

LCons T x (LApp T xs' ys) = LCons T x (LApp T xs' ys).
```

That lazy lists and this concrete implementation of lazy append form a monoid under bisimilarity need not to be shown here - I guide the reader to the .v file. The proofs are very similar to the ones involving inductive lists. However, co-induction is used instead of induction. Using the rewrite tactic with the unfolding lemmas will guard the co-inductive hypotheses.

20 Coinductive naturals and minimum

As an extra item I have formalized co-inductive naturals, an equivalence relation for them and a binary operation which finds the minimum of two co-inductive naturals. I will not go into details with the proofs here, but co-inductive naturals and minimum form a monoid. This is because one can represent 'infinite' with co-inductive naturals - it is the co-inductive natural which consists only of successors. I've explained before that inductive naturals and minimum does not form a monoid. It would require the existence of a neutral element for minimum, and the inductive naturals do not have an upper bound. However, the co-inductive naturals have: the infinite element.

21 Polymorphic 2x2 matrices and 'addition'

As another extra item I have defined polymorphic 2x2 matrices. Such a matrix contains four elements of some given type. Addition between these is defined a lot like normal matrix addition except for the fact that it takes some binary operation and applies this between each pair of elements. The specification is

```
Definition specification_of_polymorphic_matrix_addition
          (T : Type)
          (eqT : T -> T -> Prop)
          (Dot : T -> T -> T)
          (PAdd : PM22 T -> PM22 T -> PM22 T) :=
  forall a11 a12 a21 a22 b11 b12 b21 b22 : T,
    eq_PM22 T eqT
           (PAdd (pm22 T a11 a12 a21 a22) (pm22 T b11 b12 b21 b22))
           (pm22 T (Dot a11 b11) (Dot a12 b12) (Dot a21 b21) (Dot a22 b22)).
```

Polymorphic matrices and polymorphic matrix addition can be shown to form a monoid under the given equality relation if T and Dot form a monoid under eqT. I will not go much into details with the proofs here - they are a lot like proofs of properties of 2x2 matrices consisting of natural numbers and matrix addition.

22 Two other properties of monoids

It can be shown that the neutral element in a monoid is unique i.e. if two elements have that property they must be related with the provided equality relation. Let e1 and e2 be two such elements. Writing '.' as infix notation for the binary operation and '$\sim$' as infix notation for the equality relation, x . e1 $\sim$ x for any x and e2 . y $\sim$ y for any y. Therefore, e2 . e1 $\sim$ e2 and e2 . e1 $\sim$ e1. Since $\sim$ is symmetric and transitive, e1 and e2 must be related.

Our lecturer recommended the monoid provers to do the following proof - I was sceptical at first, but I think it was one of the biggest eye-openers of this project. The proof shows the importance of the binary operation preserving the equality relation. The problem is to show that for any neutral element e and two arbitrary elements x and y, x . e . y $\sim$ x . y. I found this to be the hardest problem in the project at first since I had not been thinking about preservation. However I had a conversation with the lecturer and I was enlightened.

I redesigned my monoid specification to require the binary operation to preserve the equality relation. Then the above will be true if x . e $\sim$ x and y $\sim$ y. The first follows from e being a neutral element and the second from $\sim$ being reflexive.
This situation can best be explained through a quote from one of my favorite movies where the protagonist is trying to unravel the mysteries of Christmas:

Of course, I've been too close to see!
The answer's right in front of me!
- Jack Skellington in the Nightmare Before Christmas

He was later shot down by a cannon, but don't worry - he survived. He even figured his death out afterwards. Failure is the first step towards success.

23 Conclusion

I have formalized the specification of monoids in Coq and proved 11 items to be forming monoids. One item did not. No libraries has been used - everything is stand-alone. An important part of the project has been about equivalence relations and preservation of them by binary operations, which has been used to rewrite during proofs. Underway it has also been explained why using the rewrite tactic can be easier to overcome than by applying lemmas all the time. The contradiction tactic has been used a lot in proving various properties of my defined equivalence relations. I have briefly explained induction and co-induction in connection to my defined types, and I have encountered some problems with co-inductions and guardedness.

All in all, Coq is a very powerful language. I have had fun in making everything myself - I know almost everything that is going on in my project, but it also feels like inventing the wheel. Anyhow, I feel like I have learned a lot - Coq has so much to offer, and I know so little.