

# Theory exercises

Program verification with types and logic (NWI-IMC060)

Week 2 and 3

1. This exercise is about basic programming and proofs in Coq.

(a) Define a function:

```
Fixpoint leb (n m : nat) : bool :=  
  (* FILL OUT HERE *)
```

that returns `true` if a the natural number `n` is smaller or equal to `m`, and `false` otherwise.

(b) An alternative way to define inequality of natural number is:

```
Definition le (n m : nat) : Prop := exists k, m = n + k.
```

Explain the difference between `le` and `leb`. As part of your answer you should formally explain how `le` and `leb` are related by stating a Coq lemma. You **do not** have to give a proof for the lemma that you have stated.

(c) For the following lemmas, explain whether you would prove them by simplification (`simpl`), case analysis (`destruct`), or induction (`induction`). Also explain whether or not you would use `discriminate`.

1. `forall n m, leb (S n) (S m) = leb n m`
2. `forall n, leb 0 n = true`
3. `forall n, leb n 0 = true -> n = 0`
4. `forall n, leb n n = true`
5. `forall n m k, leb (k + n) (k + m) = leb n m`
6. `forall n m k, leb n m = true -> leb m k = true -> leb n k = true`

2. For a type `A` we have the following notions of equality in Coq:

- `eq : A -> A -> Prop`
- `eqb : A -> A -> bool`
- `dec : forall x1 x2 : A, { eq x1 x2 } + { ~eq x1 x2 }.`

This question is about these three notions of equality.

- (a) Explain the difference between `eq` and `eqb`. You should give the formal equivalence between `eq` and `eqb` as a Lemma in Coq syntax.
- (b) Explain the type `forall x1 x2 : A, { eq x1 x2 } + { ~eq x1 x2 }` of `dec`.
- (c) Define `eqb` in terms of `dec`. Explain how to prove the Lemma of Exercise a for your definition of `eqb`. You do not need to give a Coq proof script, but you should give a clear description of how the proof is carried out.

3. To express whether an element  $x$  is in a list  $xs$ , we define a recursive predicate in Coq:

```
Fixpoint In {A} (x : A) (xs : list A) : Prop :=
  match xs with
  | [] => False
  | x'::xs' => x = x' \/ In x xs'
  end.
```

- (a) Give a version of `In` with return type `bool` instead of return type `Prop`. You are allowed to restrict to certain kinds of types  $A$ , but you should clearly describe the class of types you consider.
- (b) Assume that you have the following function at your disposal:

```
Fixpoint app {A} (xs1 xs2 : list A) : list A :=
  match xs1 with
  | [] => xs2
  | x :: xs1 => x :: app xs1 xs2
  end.
```

Write a non-recursive definition for `In` using `app` that is logically equivalent to the definition of `In` above.

```
Definition In {A} (x : A) (xs : list A) : Prop :=
  (* YOUR ANSWER HERE *)
```

- (c) Consider the following function that transforms a list point-wise:

```
Fixpoint map {A B} (f : A -> B) (xs : list A) :=
  match xs with
  | [] => []
  | x::xs' => (f x)::(map f xs')
  end.
```

Can the following lemma be proved?

```
Lemma In_map {A B} (f : A -> B) (x : A) (xs : list A) :
  In (f x) (map f xs) <-> In x xs.
```

If the lemma can be proved, explain which tactics you would use to perform the proof (you do not have to give a proof script). If the lemma cannot be proved, give a counterexample.

- (d) Give a lemma that completely characterizes the elements of `map f xs`, *i.e.*, applies to all values  $y$  rather than only those of the form  $f x$ :

```
Lemma In_map {A B} (f : A -> B) (y : B) (xs : list A) :
  In y (map f xs) <-> (* YOUR ANSWER HERE *)
```