Verifying programs and proofs
part IV. Proofs about arithmetic programs

Yves Bertot

October 2019

1 Motivating introduction

Many proofs by induction just fall through without thinking, but some functions are more tricky than simple traversal of data-structure: they use accumulators, or they call themselves on other arguments than the immediate subterms of the input. For these function, proofs by induction is still the main tool, but such proofs must be planned carefully.

2 Advanced proof by induction for numbers

When proving statements by induction, it may be easier to prove stronger statements. This may seem paradoxal, because stronger statements will be harder to prove. However, in proofs by induction the fact to be proved is repeated in the induction hypothesis, which also becomes stronger.

2.1 Multi-step induction

The rule for defining a recursive function is that it can call itself on a different argument than an immediate subterm, as long as this argument can be traced back to the original argument through a descent in the structure. However, the induction principle associated to a given function is still usually restricted to only the immediate subterm. General solution exist, but for the moment we shall give a limited approach that shows how to overcome this hurdle. The idea is to prove a statement that covers the descent down to last but one step.

Here is an example to illustrate this idea. The first few lines of the following proof are used to show that plain induction fails.

```
Require Import Arith Psatz.

Fixpoint mod2 (n : nat) :=
  match n with S (S p) => mod2 p | a => a end.

Lemma mod2_lt_2 : forall n, mod2 n < 2.
```

```
Proof.
induction n; simpl; [lia | ]
  n : nat
  IHn : mod2 n < 2
  ============================
   match n with
   | 0 => S n
   | S p => mod2 p
   end < 2
destruct n;[lia | ].

  n : nat
  IHn : mod2 (S n) < 2
  ============================
   mod2 n < 2
```

Invoking `destruct` is natural because the goal's conclusion contains a pattern-matching construct. But the goal that we obtain is not easy to prove: there is a mismatch between the statement that concerns `n` and the hypothesis that concerns `S n`. We should abort this proof and start afresh, with a stronger statement.

2.2 Fixed multi-step induction

The recursive call of function `mod2` is made on a `p` such that the successor of `p` is not the initial argument, but the successor of `S p` is. So, we are only missing one step. To cope with the problem, we prove a conjunction of the fact that is interesting to us and the similar fact for `S p`. Here is the complete proof.

We use the tactic `assert` to introduce the stronger statement.

```
Lemma mod2_lt : forall n, mod2 n < 2.
intros n; assert (H : mod2 n < 2 /\ mod2 (S n) < 2);
 [ | destruct H; assumption].
  n : nat
  ============================
   mod2 n < 2 /\ mod2 (S n) < 2
induction n;[simpl; lia | ].
  n : nat
  IHn : mod2 n < 2 /\ mod2 (S n) < 2
  ============================
   mod2 (S n) < 2 /\ mod2 (S (S n)) < 2
```

The new goal is more complex: we now have to prove a conjunction instead of an equality. But the induction hypothesis is also a conjunction, and the right-hand side of the hypothesis coincides with the left hand-side of the goal's conclusion. Thus, we can destruct the hypothesis and solve the first part of the proof quickly.

```
destruct IHn as [H1 H2]; split;[assumption | ].
  n : nat
  H1 : mod2 n < 2
  H2 : mod2 (S n) < 2
  ============================
   mod2 (S (S n)) < 2
```

By computation, `mod2 (S (S n))` is the same as `mod2 n`. Thus, this goal is easily solved using the `assumption` tactic.

2.3 course-of-value induction

The example given in the previous section shows that we can have induction hypothesis not only on the predecessor of a number, but on the predecessor and the predecessor of the predecessor. Sometimes, we may want to have an induction hypothesis on all numbers smaller than a number.

There exists a theorem that covers this kind of need, but it applies in a more general setting. The name of the theorem is `well_founded_ind` and it applies to a variety of binary relations, as long as they satisfy the property to be "well founded". The relation `lt` satisfies this property. Thus we can combine two theorems together to obtain an induction principle that is stronger than the basic one we have already used.

```
Check well_founded_ind lt_wf.
well_founded_ind lt_wf
 : forall P : nat -> Prop,
   (forall x : nat, (forall y : nat, y < x -> P y) -> P x) ->
   forall a : nat, P a
```

Let's read carefully the statement of this theorem. It is universally quantified over a predicate `P`. It has only one case to cover (instead of two for a regular induction principle) but this case contains induction hypotheses for all numbers less than the number for which the property needs to be proved. If we manage to prove this single case, the predicate holds universally.

For the number 0, there is no other number that is less than 0, so there is no number for which induction hypotheses hold. So this is similar to the base case of the conventional induction principle.

This is illustrated in an example reasoning on an implementation of a division function on natural numbers. This function returns a pair containing the quotient and the remainder of the division. Note that this function is tuned to handle divisions by 0 gracefully. When dividing by 0, the quotient is set to 0 and the remainder is the number being divided.

```
Fixpoint div (n m : nat) :=
 match m, n with
   S m', S n' =>
   if leb m n then
```

```
      let (q, r) := div (n' - m') m in (S q, r)
    else
     (0, n)
 | _, _ => (0, n)
 end.
```

When this function computes it has recursive calls, where the second argument never decreases (the initial value is `m` the value of the second argument in the recursive call is also `m`). The first argument decreases by a variable amount. For instance, if `m` is 1, then `m'` is 0, and if we compute `div 7 1`, then the recursive calls are `div 6 1`, `div 5 1`, etc. On the other hand, if `m` is `3`, then `m'` is `2`, and if we compute `div 7 3`, then the recursive calls are `div 4 3` and `div 1 3`.

We can now prove a simple equality about this division function.

```
Lemma div_eq :
   forall n m, n = m * fst (div n m)  + snd (div n m).
intros n; induction n  as [n IHn] using (well_founded_ind lt_wf).
  n : nat
  IHn : forall y : nat,
        y < n -> forall m : nat,
                    y = m * fst (div y m) + snd (div y m)
  ============================
   forall m : nat, n = m * fst (div n m) + snd (div n m)
```

As we see here, the induction hypotheses states that every `y` smaller than `n` satisfies the property that we want to prove for `n`.

The rest of this proof is left as an exercise. Another exercise consists in proving that the remainder of the division is smaller than the divisor, when this divisor is not 0.

3 Strengthening a statement by universal quantification

For lists, the technique of strengthening induction also applies. Sometimes, the induction is actually disguised as an induction on natural numbers, by using the size of the list for example. Some other times, functions have several arguments and it is necessary to have universal quantification on some of the arguments instead of fixed values.

To illustrate this, let's look at a proof concerning reversed lists.

There are two ways to reverse lists, but only one is efficient.

```
Require Import List.

Fixpoint slow_rev_nat (l : list nat) : list nat :=
  match l with
    nil => nil
  | a::l' => slow_rev_nat l' ++ (a::nil)
  end.

Fixpoint pre_rev_nat (l l' : list nat) : list nat :=
  match l with
    nil => l'
  | a :: l' => pre_rev_nat l (a::l')
  end.

Definition rev_nat (l : list nat) : list nat :=
   pre_rev_nat l nil.
```

In the definition of `slow_rev_nat`, we use a function

`app`, with an infix notation `++` to concatenate two lists. It is easily to be convinced that `slow_rev_nat` reverses a list: when working on a list with a first element `a`, this element ends up in the end and the rest of the list is also reversed. So we can use it as a reference implementation, but it is slow, because concatenating takes a time proportional to the length of the first list being treated. In the end, the complexity of this function is quadratic.

The other function is more efficient: reversal takes place in linear time. We can see that by testing the functions on list of increasing length. So it is useful to have both functions available. The first one can be used in specifications, the second one in implementations.

If we want to express the correctness of the efficient function, we can just write the following statement.

```
Lemma rev_nat_correct : forall l, rev_nat l = slow_rev_nat l.
```

If we start our proof directly by an induction we discover that the statement loses its beautiful shape and the proof gets stuck in the recursive case.

```
induction l as [ | a l' IHl'].

  ======================
   rev_nat nil = slow_rev_nat nil
reflexivity.
simpl.

  IHl' : rev_nat l' = slow_rev_nat l'
  ======================
   rev_nat (a::l') = slow_rev_nat l'::(a::nil)
```

At this point, there is no simple way to show a correspondance between `rev_nat a::l'` and `rev_nat l'`; mainly because the latter is not a subterm of the former. This proof gets stuck.

To repair this proof, we can rely on a stronger statement, manipulating the function `pre_rev_nat` and quantifying over both its arguments.

```
Lemma rev_nat_correct : forall l, rev_nat l = slow_rev_nat l.
assert (forall l l', pre_rev_nat l l' = slow_rev_nat l ++ l').
 induction l.
  intros; reflexivity.
 intros l'; simpl.
  rewrite <- app_assoc.
 simpl; apply IHl.
intros l; unfold rev_nat; rewrite H, <- app_nil_end; reflexivity.
Qed.
```

4 Exercises

1. Here is a definition of division by 2 in natural numbers:

   ```
   Fixpoint div2 (n : nat) :=
    match n with S (S p) => S (div2 p) | _ => 0 end.
   ```

   Using the definition of `mod2` from these course notes, prove the following lemma:

   ```
   Lemma mod2_div2_eq n : n = mod2 n + 2 * div2 n.
   ```

2. Prove that `forall n, div2 (2 * n) = n`.

3. Prove that a number cannot be at the same time odd and even.