Proof Assistants – TP. 3

Bruno Barras

September 27, 2019

1 Basic inductive definitions

1.1 Definitions by case-analysis

We recall the definition of booleans in the prelude as an inductive type with two constant constructors:

```
Inductive bool : Type := true : bool | false : bool.
```

1- Define boolean negation `negb` and boolean conjunction `andb`.

2- Detail the normalisation steps of expressions `(fun x:bool => negb (andb false x))` and `(fun x:bool => negb (andb x false))`. Use the command `Eval compute in <term>.` to obtain the normal form and check your answers. What is remarkable ?

1.2 Strong elimination and dependent match

Let t_1 and t_2 be two arbitrary terms of type T_1 and T_2. We would like to write a function `g` such that `g true` yields t_1 and `g false` yields t_2.

Is the following function correct ? Why ?

```
Definition g (b:bool) := match b with true => t1 | false => t2 end.
```

The above definition can be fixed by writing a `return` clause:

```
Definition g (b:bool) := match b return <todo> with true => t1 | false => t2 end.
```

Now we want to write a variant of the above by reversing the true and false cases:

```
Definition g (b:bool) := match negb b with true => t2 | false => t1 end.
```

Of course, a `return` clause is needed, but since the destructed object is not just a variable, an `as` clause is also needed to assign a name to the destructed object. This name can be used in the `return` clause.

```
Definition g (b:bool) := match negb b as <todo> return <todo> with true => t1 | false => t2 end.
```

1.3 Logical connectives

Observe how the logical connectives `and`, `or`, `ex`, `eq` and their induction schemes are defined in the standard library of Coq.

1.4 Even numbers

Consider the definition of even numbers

```
Inductive even : nat -> Prop :=
| even0 : even 0
| evenS n : even n -> even (S (S n)).
```

Prove:

```
Lemma even_is_double : forall n, even n -> exists m, n=m+m.
```

by induction on (even n).

The above lemma can also be proven by induction on `n`. The proof is harder as the straightforward induction does not work. Observe how in the induction step, the induction cannot be used as it would require that the predecessor of an even number is also even.

One solution is to prove the property for `n` and `S n` at the same time:

```
Lemma even_is_double' : forall n,
  (even n -> exists m, n=m+m) /\ (even (S n) -> exists m, n=S(m+m)).
```

Prove this lemma.

2 Recursive types

Consider the definition of lists (already in the prelude):

```
Require Import List.
Inductive list (A : Type) : Type :=
  nil : list A | cons : A -> list A -> list A
```

2.1 Proofs by structural induction

1- Implement a function `belast : nat -> list nat -> list nat` that drops the last element of a list:

- `belast x nil = nil`
- `belast x (cons y l)= cons x (belast y l)`

2- Show the following statement:

```
Lemma length_belast (x : nat) (s : list nat) : length (belast x s) = length s.
```

3- Implement a function `skip : list nat -> list nat` such that removes from a list items at positions that are odd, e.g:

- `skip (cons x (cons y (conz z nil)))= cons y nil`

4- Show the following statement:

```
Lemma length_skip :
 forall l, 2 * length (skip l) <= length l.
```

Again, a straightforward induction on `l` does not work since `skip` makes recursive calls on the tail of the tail of the list (and not its tail). Another solution is to use the tactic `fix hyp n` that allows to prove a property (or inhabit a type) by adding an hypothesis `hyp` which has the same type as the original goal. The number `n` indicates which argument of the function is the inductive object which size decreases along the recursive calls. Here `n=1` because the list is the first argument. The hypothesis `hyp` can only be called on lists that are a suffix of list `l`.

2.2 Dependent types, recursively

The Coq prelude defines the binary product, the unit type and the type of natural numbers:

```
Inductive prod (A B : Type) : Type := pair : A -> B -> prod A B.
Inductive unit : Type := tt : unit.
Inductive nat : Type := O : nat | S : nat -> nat.
```

Construct an expression `prodn : Type -> nat -> Type` which builds the n-ary product of a given type A: (i.e. `prodn A n` is $A \times \ldots \times A$ (n times)). The definition will be by recursion on n.

Note that there exists a Gallina command `Fixpoint f (x:I): ty := def.` which is equivalent to `Definition f (x:I):= fix f (x:I): ty := def.`

Give an expression `length : forall A, list A -> nat` which computes the length of a list.

Give an expression `embed : forall A (l:list A), prodn A (length l)` which translates a list into a n-uple.