Verifying programs and proofs
part VI. A glimpse at dependent types

Yves Bertot

November 2019

1 Motivating introduction

Proofs and programs are handled in the same way in the langage of the Coq system (this language is also called the gallina language). This is a central aspect of Type Theory, the theory that underlies the implementation of the system. Studying this theory for its expressive power is the subject of an entire teaching module, (*Type Theory*), but we wish to give a glimpse of it in this chapter.

2 The *Curry-Howard* isomorphism

The Coq system provides both a small purely functional programming language and a logical system. The key point of its design is that the logical system and the programming language are only one language.

2.1 Intuitive understanding of proofs-as-programs

From a practical point of view, theorems are tools. A theorem with the statement $A \Rightarrow B$ is a tool we can use to produce proofs of B whenever we already have a proof of A. In this sense, it is a function that takes as arguments proofs of A and produces proofs of B. The idea of Type Theory is to push this idea to the extreme so that any object of type $A \to B$ is actually a proof of the statement $A \Rightarrow B$. So the type is the statement, or the other way round, the statement is the type. This is known as the *proposition-as-types* correspondence.

If we turn attention to proofs, the statement $A \Rightarrow B$ holds when we are able to produce an object p whose type is $A \to B$. So when we write $p : A \to B$, we state that $A \to B$ is a proposition and p is a proof of this proposition.

The key insight is that this applies to other type constructions than just the arrow type. Every construct from the logical system corresponds to a construct from the programming language and vice-versa.

When considering the programming language as a way to perform proofs, the actual value of proofs is less important than the fact that they exist or not. Part of the Coq system relies on a distinction between types that are supposed to represent propositions and types that are supposed to represent datatypes. The type `Prop` is used as type of types, with the intent that the elements of `Prop` are used to represent propositions (statements). For other types, there exist another type of types, called `Type`. There is a form of subtype relation here: a function that expects an element `Type` as argument can be applied to an element of `Prop`, but a function that expects an element of `Prop` as argument cannot be applied to arbitrary elements of `Type`.

Beyond implication, the expressive power of the logical system relies on universal quantification. When we mirror the behavior of universal quantification in the programming language, we discover a feature that is usually not present in conventional programming languages. We have a statically typed language, but a single function can return values in different types.

We can illustrate this with a simple predicate on natural numbers, for example the predicate `even`, which expresses that a number is a double. Using plain functional application, we have that `even 0`, `even 1` is a proposition, `even 2` is a proposition, and so on. If we denote `Prop` a type for propositions, then `even` must have type `nat -> Prop`. At this point, we should note that `even 0` and `even 1` should be distinct propositions, because the former should be provable (it should contain a proof), and the second one should not.

When we perform a proof by induction for some property $P(n)$ where n is a natural number, the proof by induction actually leads to verifying that $P(0)$ holds (we can call this information the base fact) and that for every number n, $P(n+1)$ holds under the assumption that $P(n)$ already holds (we can call this information the step fact). Intuitively, if we combine the base fact and the step fact once, we get a proof of $P(1)$, then combining this proof of $P(1)$ with the step fact again, we can get a proof of $P(2)$. Obviously, for every natural number k, we only need to repeat this process k times to obtain a proof of $P(k)$. However, any finite proof document would only produce $P(k)$ up to a certain bound. The induction principle gives a way to describe this repetitive process and we shall see that the whole idea can be described as a recursive function. When we have the recursive function, we have a stronger fact: the statement holds for any natural number. This is expressed by a statement $\forall x : \texttt{nat}, P(x)$ and the correspondence between propositions and types implies that this statement should also be the type.

3 dependent types

In the same manner that a theorem that proves $A \Rightarrow B$ is a tool that constructs proofs of B from proofs of A, a theorem that proves $\forall x : A, P(x)$ is a tool that constructs proofs of $P(a_1)$, $P(a_2)$, and so on for any any elements a_1, a_2 in A. In this sense, theorems proving universally quantified statements are also functions mapping objects to proofs, except that the input objects may not be proofs, but regular data.

If the theorem th has the statement $forall x : A$, $P(x)$ and a_1 is an element in A, we can write $th\ a_1$ (the application of th to a_1) and this gives us a proof of $P(a_1)$. When applied to another argument a_2, this gives us a proof of $P(a_2)$, this is really a different type.

The notation $\forall (x : A), P(x)$ is often called a *product type*. This is a reference to cartesian products. When using a cartesian product, or repeated cartesian products to obtain a tuple, we have values in a family of types. For instance, if the cartesian product type $A_1 * A_2$, then an element of this type provides a guarantee to return an element of A_i for any i in $\{1, 2\}$. A product type generalizes this situation to the case where the indices are taken from any type instead of the finite set $\{1, 2\}$. If we have a function of type $\forall x : A, P(x)$ we also guarantee the existence of elements in $P(x)$ for any possible index x.

3.1 Functions with dependent types

In practice, this means that we have two markers for function types. The one that we already know is the "arrow" type, which have been using for all functions in previous lessons and which we saw in the previous section that it can also be read as type for implication. The new function type is the universal quantification, sometimes also called a product type. Understanding how these are used in well formed formulas revolves around the following two sentences.

- if a function f has type $A \to B$ and a value e has type A, the the expression $f\ e$ is well formed and has type B.
- if a function f has type $\forall x : A, B$ and a value e has type A then the expression $f\ e$ is well formed and has type B where all free occurences of x are replaced by e.

In reality, the Coq system does not have two rules for typing function calls, it only knows the rule for functions with a type of the form $forall x : A, B$. The trick that Coq uses is that universally quantified formulas such that x does not occur in B are automatically printed as arrow types.

```
Check (forall x: nat, nat).
nat -> nat : Set
```

In the Coq system, proofs are always represented as programs in the same programming language that we used for our simple programs. The goal-directed machinery and the tactics that we learned to use in the previous lessons are only provided to help us construct these proofs, but we can play the game of constructing proofs without using tactics. Actually an expert user of Coq will regularly have small fragments of proofs built directly by applying theorems to values in the middle of their tactic scripts.

A very simple example of proposition that is always true and can be described using implication and universal quantification is the proposition *for every proposition A, if A holds then A holds*. A proof of this proposition can be written directly in the Coq system without using tactics in the following manner.

```
Definition basic_truth : forall A : Prop, A -> A :=
  fun (E : Prop)(x : E) => x.
```

This is an example of a tautology. As often, this tautology is not very interesting, it does not teach us much. We shall only use it when we need a provable proposition somewhere.

A very simple example of a proposition that cannot be proved is the following one:

```
Check forall A : Prop, A.
forall A : Prop, A
   : Prop
```

That this formula cannot be proved is the central fact guaranteeing that the Coq system is consistent.

3.2 Constructing proofs directly

If we have a theorem *th* whose statement is $\forall x : A, P\ x \rightarrow Q\ x$, this means that P has type $A \rightarrow$ `Prop`, Q has type $A \rightarrow$ `Prop`, and *th* is actually a function accepting two arguments. To illustrate this, we shall use an example with two theorems that are always loaded at the beginning of a Coq session.

These statements are concerned with the two-place predicate $. \leq .$

```
Check le.
le : nat -> nat -> Prop
```

This predicate is about two natural numbers. When these numbers are given, a specific notation is triggered.

```
Check le 3 1.
3 <= 1
```

Among the basic theorems that are provided for `le`, here are two that we will use in this illustration.

```
Check le_n.
le_n : forall n : nat, n <= n

Check le_S.
le_n : forall n m : nat, n <= m -> n <= S m
```

Theorem `le_n` is a function that takes only one argument. Theorem `le_S` is a function that takes three arguments, two natural numbers and a proof that these numbers respect `le`.

For instance, we wish to construct a proof of `3 <= 5`. We will proceed in the following manner.

- First we note that `le_n 3` is a proof of `3 <= 3`.
- Then `le_S 3` is a proof of `forall m, 3 <= m -> 3 <= S m`
- `le_S 3 3` is a proof of `3 <= 3 -> 3 <= 4`
- This is a function, it expects an element of type `3 <= 3` as argument, but we know how to produce one: `le_n 3`, so we can continue
- `le_S 3 3 (le_n 3)` is a proof of `3 <= 4`
- `le_S 3 4` is a proof of `3 <= 4 -> 3 <= 5`
- This is a function, it expects an element of type `3 <= 4` as argument so we can continue
- `le_S 3 4 (le_S 3 3 (le_n 3))` is a proof of `3 <= 5`.
- We can use `Check le_S 3 4 (le_S 3 3 (le_n 3)).` to verify this with the Coq system.

If we wanted to prove that 10 is larger than 0, it would be feasible by this approach. If we wanted to prove that 100 is larger than 0, it would be feasible but tedious. We need to find a way to avoid this kind of proof expansion.

4 dependently typed pattern-matching

Functions with dependent types are rarely provided by conventional programming languages.

In the programming language of the Coq system, we can construct a function that returns a dependent type in a principled way. Even for a function acting on natural numbers and in charge of returning values in different types. The main tool is *dependent pattern-matching*, where the user can instruct the Coq system to construct explicitely objects of different types in each of the branches.

A general form of recursive function on the type of natural numbers would have the following shape.

```
Fixpoint gen_rec ... (n : nat) : ... :=
  match n with
  | O => ...
  | S p => ...
  end.
```

First we would like to state that the returned value will be in a type written as `B n` for some `B`. The first trick we will use is to make this `B` an argument of the function. What we want is that `B n` should be a type whenever `n` is a natural number. To say this, we state that `gen_rec` has a first argument `B : nat -> Type`. Our definition takes the following form:

```
Fixpoint gen_rec (B : nat -> Type) ... (n : nat) : B n :=
  match n with
  | O => ...
  | S p => ...
  end.
```

Now we need to instruct the Coq system so that it understands that the branches of the pattern matching construct should return a different type, this is done by the following syntax:

```
Fixpoint gen_rec (B : nat -> Type) ... (n : nat) : B n :=
  match n return B n with
  | O => ...
  | S p => ...
  end.
```

This says that the first branch should return a value in type `B 0`, and the second branch should return a value in `B (S p)`.

For the branch on 0, we know that the returned value should have type `B 0`. Since `B` is completely unknown, we don't know how to construct such a value, so we will simply assume it is given as an argument to the function `gen_rec`.

```
Fixpoint gen_rec (B : nat -> Type) (V : B 0)... (n : nat) : B n :=
  match n return B n with
  | 0 => V
  | S p => ...
  end.
```

For the second branch, we know the returned value should have type `B (S p)`, but we also know that this value can be built using `p` and the result of a recusive call on `p`. This recusive call will produce a value of type `B p`. So all the computation happening in the second branch can be modeled by some function `F` that takes `p` as argument, a value in type `B p` and has to produce a result in type `S p`. This function `F` and must have type `forall p : nat, B p -> B (S p)`. As for `V` we don't know enough to invent this `F` so we assume it is given as argument to the `gen_rec` function. In all the complete code for the function looks as follows:

```
Fixpoint gen_rec (B : nat -> Type) (V : B 0)
  (F : forall p : nat, B p -> B (S p)) (n : nat) : B n :=
  match n return B n with
  | 0 => V
  | S p => F p (gen_rec B V F p)
  end.
```

We shall now illustrate how such a function could be used, for instance to construct proofs. For instance, we may want to prove that every natural number is larger than 0, as was mentioned in section 3.2. We will use the function `gen_rec` with the type family `B` fixed to `fun n : nat => 0 <= n`. For value `V`, we need a value of type `0 <= 0`. We can construct such a value using `le_n`. For `F`, we need to have the type `forall p, 0 <= p -> 0 <= S p`. It turns out that `le_S 0` has exactly this type. So we can instantiate `gen_rec` with these 3 arguments.

```
Definition proof_0_le : forall x, 0 <= x :=
  gen_rec (fun x => 0 <= x) (le_n 0) (le_S 0).
```

Two more remarks need to be done about this proof. First, users the `gen_rec` function already exists inside the Coq system, but under a different name. It was generated automatically when the type `nat` was defined inductively. The name of the function is `nat_rect` (the suffix `rect` should be broken in two parts: `rec` because it is the generic recursive function on natural numbers, and `t` because it is defined for any type). In general, this generic recursive function are named *induction principles*. Second, this function is interesting because it exists, but it rarely useful to execute it, otherwise it will construct the long sequence of nested theorems that were mentioned in Section 3.2.

```
Check proof_0_le 10.
proof_0_le 10 : 0 <= 10

Compute proof_0_le 10.
le_S 0 9 (le_S 0 8 (le_S 0 7 (le_S 0 6 (le_S 0 5
    (le_S 0 4 (le_S 0 3 (le_S 0 2 (le_S 0 1 (le_S 0 0 (le_n 0))))))))))
```

When it comes to proofs, the command `Check` is more important than `Compute`.

5 Dependent inductive types

The inductive types that we saw in previous sections have all their constructors with simple types. It is interesting to see what happens if we use inductive definitions for *families* of types, with constructors that have dependent types.

As an illustration, we shall consider a few types whose elements are indexed by a natural number. These types will thus have type `nat -> Type`.

5.1 A finite set inductive type

The first example is a family of types for all natural numbers, but only a few of these types have elements.

```
Inductive lt3 : nat -> Type :=
| lt3_0 : lt3 0
| lt3_1 : lt3 1
| lt3_2 : lt3 2.
```

With such a definition, the type `lt3 3` exists, but it does not contain any element. On the other side the type `lt3 0` exists, but it contains only element. This is an example of a type that could be used to represent proofs, but for now will keep it as a datatype.

If we want to define a function on `lt3` that covers all possible case, we need this function to first take an natural number argument `n`, which will be used as an index for the type `lt3 n`.

```
Definition lt3_gen (B : forall n : nat, lt3 n -> Type) ...
  (x : nat) (t : lt3 x) :=
  match x with
  | lt3_0 => ...
  | lt3_1 => ...
  | lt3_2 => ...
  end.
```

We then need to express that each of the branches of the pattern-matching construct will return a different type. This is done by modifying the pattern-matching construct in the following manner:

```
Definition lt3_gen (B : forall n : nat, lt3 n -> Type) ...
  (x : nat) (t : lt3 x) :=
  match x in lt3 a return B a x with
  | lt3_0 => ...
  | lt3_1 => ...
  | lt3_2 => ...
  end.
```

Because the constructor in the first matching clause has type `lt3 0`, this means that the first branch will have to return a value of type `B 0 lt3_0`. For the second clause, the value will have to be in `B 1 lt3_1`, and so on

```
Definition lt3_gen (B : forall n : nat, lt3 n -> Type)
  (V0 : B 0 lt3_0) (V1 : B 1 lt3_1) (V2 : B 2 lt3_2)
  (x : nat) (t : lt3 x) :=
  match x in lt3 a return B a x with
  | lt3_0 => V0
  | lt3_1 => V1
  | lt3_2 => V2
  end.
```

Sometimes, a type like `lt3` is used only for proofs and the values are not interesting. In this case, we prefer instead of `B` a predicate `P` that only refers to the natural number value not the `lt3` argument.

```
Definition lt3_gen_ind (P : nat -> Prop)
  (V0 : P 0) (V1 : P 1) (V2 : P 2) (x : nat) : lt3 x -> P x :=
  lt3_gen (fun (x : nat) (t : lt3 x) => P x) V0 V1 V2 x.
```

This is the most general way to prove that some predicate `P` is a consequence of `lt3`. According to specialists in proof theory this reasoning principle expresses that `lt3` is the least predicate (in the order of containment) that is satisfied by `0`, `1`, and `3`.

We can illustrate how to use this induction principle to show that `lt3 x` does not hold for numbers larger than 3. Here, we use addition with the number 3 to denote numbers larger than 3. In this definition we use crucially the fact `3 + x` is convertible with `S (S (S x))` and the latter can be recognized by pattern matching.

```
Definition lt3_3_any : forall (x : nat) (P : Prop), lt3 (3 + x) -> P :=
fun (x : nat) (P : Prop) (t : lt3 (3 + x)) =>
lt3_gen_ind
    (fun x =>
      match x with S (S (S k)) => P | _ => forall A : Prop, A -> A end)
     basic_truth
     basic_truth
     basic_truth
     (3 + x) t.
```

Basically, the theorem `lt3_3_any` expresses that if `lt3 3` was provable, then anything would be.

5.2 A singleton inductive type

A simpler example of finite set is the finite set with a single element, which we can describe as follow:

```
Inductive is3 : nat -> Prop :=
  is3_refl : is3 3.
```

The single constructor `is3_refl` simply says that the only number that satisfies `is3` is the number `3`.

The associated inductive principle has the following type and value:

```
Definition is3_gen_ind :
   forall P : nat -> Prop, P 3 -> forall x, is3 x -> P x :=
fun P V x t =>
match t in is3 x return P x with
  is3_refl => V
end.

Check is3_gen_ind.
forall P : nat -> Prop, P 3 -> forall x : nat, is3 x -> P x
```

In goal directed proof, this means that if we have a statement to prove of the form `P x` and we know `is3 x`, then we can transform the problem into having to prove `P 3`, like a rewrite operation.

5.3 A recursive inductive predicate

The next example is an inductive predicate where one constructor

```
Inductive ge3 : nat -> Type :=
  ge3n : ge3 3
| ge3S : forall m, ge3 m -> ge3 (S m).
```

Obviously, the type `ge3 3` contains an element, `ge3n`. If we want to construct an element of `ge3 4`, we can do so by combining `ge3n` with `ge3S`.

```
Check ge3S 3 ge3n.
ge3S 3 ge3n : ge3 4
```

We can construct an element of `ge3 10` by repeating this process again and again. A nice trick is to avoid having to fill up the numeric values, the Coq system can do that for us if we leave blank placeholders.

```
Check ge3S _ (ge3S _ (ge3S _ (ge3S _ (ge3S _ (ge3S _ (ge3S _ (ge3n)))))))).
ge3S 9 (ge3S 8 (ge3S 7 (ge3S 6 (ge3S 5 (ge3S 4 (ge3S 3 ge3n)))))) : ge3 10
```

We can also use the `gen_rec` we defined for natural numbers to repeat the use of constructors.

```
Check (gen_rec (fun x=> ge3 (x + 3)) ge3n (fun m => ge3S (m + 3)) 7) : ge3 10.
gen_rec (fun x : nat => ge3 (x + 3)) ge3n (fun m : nat => ge3S (m + 3)) 7
:
ge3 10
     : ge3 10
```

We can also construct a generic recursive function for the type `ge3`, this construction follows the same pattern as for `gen_rec`, `lt3_gen`, or `is3_gen_ind`.

We want to construct a function that takes all elements of all the types in the inductive family `ge3`, so the statement we want to obtain has the form:

```
forall (n : nat) (t : ge3 n), C n t
```

We use `C` for `ge3` as a parallel to the variable `B` that we used on `nat` in `gen_rec`.

For this statement to be well formed, we need `C` to have type

```
C : forall n : nat, ge3 n -> Type
```

In the definition of `gen_rec`, we used a value `V` that had type `B 0`. This time we need, a value that fits the first constructor `ge3n` of the type `ge3`. This value will need to have type `C 3 ge3`. This time we choose to name this value `W`.

For the second constructor, it has two arguments, the first one is a natural number `n` and the second argument `t` is an element of the type `ge3 n`, which is in the same inductive family. For this second argument, a recursive call is also possible, and it return a value in the type `C n t`. Using these three values, the whole branch must construct a value of type `C (S n) (ge3S n t)`. This time, we choose to name this value `G`.

Putting all together, we get the following definition that really has the same shape as the definition for `gen_rec`.

```
Fixpoint ge3_gen_rec (C : forall n : nat, ge3 n -> Type)
  (W : C 3 ge3n)
  (G : forall (n : nat) (t : ge3 n), C n t -> C (S n) (ge3S n t))
  (m : nat) (g : ge3 m) :=
match g in ge3 k return C k g with
| ge3n => W
| ge3S p t => G p t (ge3_gen_rec C W G p t)
end.
```

The type of `ge3_gen_rec` can be displayed with the help of the Coq system.

```
Check ge3_gen_rec.
ge3_gen_rec
     : forall C : forall n : nat, ge3 n -> Type,
       C 3 ge3n ->
       (forall (n : nat) (t : ge3 n), C n t -> C (S n) (ge3S n t)) ->
       forall (m : nat) (g : ge3 m), C m g
```

If we want to use such an inductive type only for proofs, the value of each element of `ge3` n is irrelevant, only their existence matters. In this case, it is worth instantiating this general induction principle with a proposition that does look at the values. It makes it possible to obtain a simpler induction principle.

```
Definition ge3_gen_ind :
  forall (P : nat -> Prop), P 3 -> (forall k, ge3 k -> P k -> P (S k)) ->
    forall (n : nat), ge3 n -> P n :=
  fun P (V : P 3) (H : forall k, ge3 k -> P k -> P (S k)) =>
  ge3_gen_rec (fun (x : nat) (t : ge3 x) => P x) V H.

Check ge3_gen_ind.
ge3_gen_ind
     : forall P : nat -> Prop,
       P 3 ->
       (forall k : nat, ge3 k -> P k -> P (S k)) ->
       forall n : nat, ge3 n -> P n
```

5.4 Inductive predicates in Coq

In Coq many predefined logical connective are actually given as inductive types. The most characteristic is equality, which looks like `is3`, except that the type is not `nat`, it is given as a parameter (named `A`, and the value is not 3, but an arbitrary value in `A`, here named `x` and also given as parameter.

```
Inductive eq (A : Type) (x : A) : A -> Prop :=
  eq_refl : eq A x x.
```

The inductive principle, `eq_ind` associated to `eq` has the same shape as the inductive principle `is3_gen_ind` from the previous section. It expresses that when `x = y` holds, if we want to prove that `y` satisfies some property `P`, we only have to show that `x` does satisfy `P`.

```
Check eq_ind.
eq_ind
     : forall (A : Type) (x : A) (P : A -> Prop),
       P x -> forall y : A, x = y -> P y
```

In practice, this statement expresses that when one need to prove `P y` and we know that `x = y`, then the problem can be changed into having to prove `P x`. This is actually a rewrite operation, and in fact the theorem `eq_ind` is the basis for the `rewrite` tactic in Coq.

In Coq logical connectives like `/\` and `\/` are actually given as inductive definitions. Existential quantification, too.

6 Dependent types in programming

In datatypes, we can mix data and proofs. This is useful to keep information about data. For instance, we can make bounded integers, by combining a natural number and a proof that it is smaller than the bound.

```
Inductive bounded (n : nat) : Type :=
  bounded_value : forall x : nat, x < n -> bounded n.
```

It would require another lesson to see all the benefits that can be obtained from using this kind of type.

7 References

The book Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions, by Bertot and Castéran gives more explanations about the variety of inductive propositions (in chapter 6) and about the construction of induction principles (in chapter 14). The French version is available online `https://www.labri.fr/perso/casteran/CoqArt/index.html`.