An Ssreflect Tutorial

Georges Gonthier — Stéphane Le Roux

N° 367

July 2009

Thème SYM

Rapport technique

ISSN 0249-0803 ISRN INRIA/RT--367--FR+ENG

An Ssreflect Tutorial

Georges Gonthier[*] , Stéphane Le Roux[†]

Thème SYM — Systèmes symboliques
Équipes-Projets Composants Mathématiques, Centre Commun INRIA Microsoft Research

Rapport technique n° 367 — July 2009 — 30 pages

Abstract: This document is a tutorial for ssreflect which is a proof language based on Coq. This tutorial is mostly dedicated to people who already know the *basics* of logic.

Key-words: proof assistants, formal proofs, Coq, small scale reflection, tactics.

[*] Microsoft Research, Cambridge, R-U, Centre commun INRIA Microsoft Research
[†] Centre commun INRIA Microsoft Research

Un tutoriel Ssreflect

Résumé : Ce document est un tutoriel pour ssreflect qui est un langage de preuve dérivé de Coq. Ce tutoriel est principalement écrit pour qui connaît déjà les *bases* de la logique.

Mots-clés : assistants à la preuve, preuve formelle, Coq, réflexion à petite échelle, tactiques.

Contents

1 Introduction

1.1 Ssreflect is based on Coq

Ssreflect is a formal proof language that is based on Coq. The main design feature of ssreflect is on the one hand that it uses the Coq definition language, *i.e.* the language that is dedicated to the definition of objects in Coq, and on the other hand that it provides additional tactics, *i.e.* reasoning steps, suitable for the long mathematical proofs that ssreflect is intended to help encode. This extension of Coq has already proved very efficient (See the 4-colour theorem proved using ssreflect).

The ssreflect additional tactics are few, but they can be combined with additional tacticals, *i.e.* tactic modifiers, such that one same tactic may cope with a wide range of similar situations. Also the tactics combine nicely with each other, so that proof scripts may be sometimes as short as the pen-and-paper proofs that they formally encode.

1.2 Interactive proof building

In Coq and ssreflect, proofs are usually built through the interactive mode: the user writes definitions, statements and proofs in a window, and asks the software to verify them step by step. Along these verifications the software outputs agreeing or complaining messages in another window. When in the middle of a proof verification, the software displays additional information: which objects are at the user's disposal and which intermediate statements (subgoals) are still to be proved in order to complete the proof of the initial statement (goal). These are graphically organised as follows: (part of) the objects that are at the user's disposal, thus constituting the *context*, are displayed above an horizontal line, while the subgoals are displayed below the line. When several subgoals are still to be proved, they are all displayed below the line but only the context of the first subgoal, which is the current/working subgoal, is displayed above the line. In the example below, two hypotheses are available in the context: `Hyp1` which states that `A` holds and `Hyp2` which states that `B` holds. The first subgoal to be proved is that if `C` holds then `D` holds, where the implication is represented by an arrow. Moreover, there is a second subgoal `E -> F`.

```
Hyp1 : A
Hyp2 : B
===============
C -> D

subgoal 2 is:
 E -> F
```

In order to prove the (sub)goal(s), the user performs reasoning steps, which are called *tactics*. Some tactics can put objects in or remove them from the context, but the other tactics can only interact with the subgoals. Contexts are shelves storing tools and material, while subgoals are workbenches where we actually operate the tools and build parts of the intended piece of furniture. Furthermore, subgoals are like stacks: when moving objects around, one can only push them to or pop them from the left-hand side of the subgoal. Therefore subgoals may be called *proof stacks*. In the first subgoal of the example above, `C` is at the top of the proof stack.

1.3 Structure of the tutorial

Section 2 translates part of the Coq tutorial (v8.1) into ssreflect. Section 3 gives slightly more advanced arithmetic examples related to Euclidean division. It is strongly advised that the reader actually run the proof script (either from the file tutorial.v or by copy and paste from this tutorial) while reading the tutorial. Also, the explanations given here are mostly intuitive. For more formal explanations see the reference manual (http://hal.inria.fr/inria-00258384).

2 The Coq tutorial briefly translated to ssreflect

The following files are imported from the ssreflect standard library.

```
Require Import ssreflect ssrbool eqtype ssrnat.
```

2.1 Hilbert's axiom S

The following script proves Hilbert's axiom S step by step for pedagogical purpose.

```
Section HilbertSaxiom.

Variables A B C : Prop.

Lemma HilbertS : (A -> B -> C) -> (A -> B) -> A -> C.
Proof.
move=> hAiBiC hAiB hA.
move: hAiBiC.
apply.
  by [].
by apply: hAiB.
Qed.
```

The command

```
Section HilbertSaxiom.
```

starts a Coq section that will be closed later.

The command

```
Variables A B C : Prop.
```

assumes the propositional variables A, B and C (whose type is therefore Prop). These variables will be available throughout the current section HilbertSaxiom.

The lemma above is called HilbertS. It states

```
(A -> B -> C) -> (A -> B) -> A -> C
```

where

```
A -> B -> C
```

stands for

```
A -> (B -> C)
```

which means "if A holds then if B holds then C holds".

Technically, it is different from

```
(A /\ B) -> C
```

which means "if A and B hold then C holds", although these two statements are equivalent.

In the proof above, the command

```
move=> hAiBiC hAiB hA.
```

first introduces in the context the assumption that lies on the top of the proof stack (goal), *i.e.* A -> B -> C, and names it hAiBiC; then it proceeds in the same way for the two remaining assumptions of the goal. This actually performs the same job as the Coq command

```
intros hAiBiC hAiB hA.
```

The command

```
move=> hAiBiC hAiB hA TooMany.
```

would fail and yield an error message because it asks ssreflect to pop four assumptions from the proof stack while there are only three.

The tactic move=> can be decomposed as follows: move is also a tactic and => is called a *tactical*. In ssreflect as in Coq, a tactical builds new tactics from existing tactics. Here, the tactic move has actually no effect and all the semantic of the Coq tactic intros is carried by the tactical => only. Note that the ssreflect writing style prescribes that assumptions have rather informative names, which intends to make larger proof scripts more readable. (Here hAiB stands for "hypothesis: A implies B".)

The colon : is also a tactical. It is the inverse of =>, so

```
move: hAiBiC.
```

takes the hypothesis hAiBiC from the context and pushes it on the top (left-hand side) of the goal. If the hypothesis hAiBiC did not exist in the context, the tactic would fail. This corresponds to the Coq command

```
revert hAiBiC.
```

In the same way,

```
move: (hAiBiC).
```

corresponds to the Coq command

```
generalize hAiBiC.
```

which does not clear the hypothesis from the context. (Try it.) Note that the colons in move: and A : Prop are unrelated, but actual confusion is unlikely.

The tactic

```
apply
```

tries to see the right-hand side of the goal (*i.e.* the proof stack without its top) as an instance/-consequence of the left-hand side (*i.e.* the top of the proof stack), and it asks the user to provide missing arguments; this generates two subgoals in the present case.

Note the indentation before the next tactic

```
by []
```

which suggests that two subgoals are to be proved at that stage.

The success, *i.e.* approval by the proof system, of the tactic `by []` means that the current subgoal can be and actually is solved trivially.

Note that `by []` is a specific instance of the more general syntax

```
by [tactic1 | tactic2 | ...]
```

which tries to apply `tactic1` to the goal; if it fails, it tries `tactic2`, *etc.*; if and when it succeeds, it then tries to solve the current subgoal trivially; otherwise it fails and returns an error message. Such tactics that fail when not solving the current subgoal are called *terminators*. After `by []` only one subgoal remains, so the indentation becomes normal again. Indentation and terminators intend to make larger proof scripts more readable by showing the structure of the proof.

The command

```
apply: hAiB.
```

is a shortcut for

```
move: hAiB.
apply.
```

which could also be written as one single compound tactic instead of two thanks to the semi-colon Coq tactical `;`.

```
move: hAiB; apply.
```

As in Coq,

```
tactic1; tactic2.
```

performs `tactic2` in all the subgoals generated by `tactic1`, or fails.

The last word of the proof is `Qed`. This triggers the verification of the proof that has just been build. This second checking is more reliable than the step-by-step validation because it relies only on the correctness of the small Coq kernel instead of the correctness of the whole set of tactics.

Below, a shorter proof of Hilbert's axiom S. Before actually stating the lemma the hypotheses are assumed for good (*i.e.* throughout the current section) in the context. Note that from the system's viewpoint, `Variable`, `Variables`, `Hypothesis`, and `Hypotheses` perform the same operation.

```
Hypotheses (hAiBiC : A -> B -> C) (hAiB : A -> B) (hA : A).

Lemma HilbertS2 : C.
Proof.
apply: hAiBiC; first by apply: hA.
exact: hAiB.
Qed.
```

In the proof above, the command

```
apply: hAiBiC; first by apply: hA.
```

performs first

```
apply: hAiBiC.
```

and then

```
by apply: hA.
```

only in the *first* subgoal generated by `apply: hAiBiC`. If the tactical `first`, which is called a *selector*, were not written, the system would attempt to perform `apply: hA` in both subgoals, and would fail on the second subgoal.

If the terminator `by` were not written, the proof would still succeed. However, the ssreflect writing style prescribes that every tactic that finishes the proof of a subgoal be a terminator, *i.e.* a "sub-`Qed`". Such information on the proof structure makes not only the proof scripts more readable but also more robust. Indeed, consider one given theorem relying on a prior development (*i.e.* definitions and lemmas). When the software or the prior development changes (new version or new formalism) the previous proof of the theorem may not be valid any longer. In order to patch the proof, one may want to start modifying the old proof where the interpreter first rejects the old proof. However, the first rejected step may not be the first step where the proof script deviates from the intended proof. Deviation may have occurred long before the place where an error is detected, and irrelevant steps may still be approved by the interpreter by chance, which interferes with debugging. On the contrary, terminators fail if they don't finish the current subgoal, thus avoiding approval of irrelevant proof steps. This facilitates debugging.

The command

```
exact: hAiB.
```

is a shortcut for

```
move: hAiB; exact.
```

where the terminator `exact` is a sort of combination between `apply` and `by []`.

Below, yet a shorter proof of Hilbert's axiom S.

```
Lemma HilbertS3 : C.
Proof. by apply: hAiBiC; last exact: hAiB. Qed.
```

The proof first performs

```
apply: hAiBiC.
```

then it performs

```
exact: hAiB.
```

only in the *last* subgoal generated by `apply: hAiBiC`, and terminates this subgoal. Then `by` terminates the remaining subgoal.

Before going any further, we need to explain how to build bigger objects/terms from smaller objects/terms in Coq, and how these are typed. The variables below were already assumed above.

```
Hypotheses (hAiBiC : A -> B -> C) (hAiB : A -> B) (hA : A).
```

So the term

```
(hAiB hA)
```

has type `B` (In traditional mathematics one would write `hAiB(hA)` instead, like `f(x)`). To verify the type of the term above, try the following command.

```
Check (hAiB hA).
```

Also, the term `(hAiBiC hA)` has type `B -> C`, and the term

```
((hAiBiC hA) (hAiB hA))
```

has type `C`, and is written

```
hAiBiC hA (hAiB hA)
```

since there is no possible confusion.

Below, a shorter proof of Hilbert's axiom S.

```
Lemma HilbertS4 : C.
Proof. exact: (hAiBiC _ (hAiB _)). Qed.
```

The term `(hAiBiC _ (hAiB _))` above has two holes represented by the placeholder `_`. Yet it still provides enough information about the proof term we mean so that ssreflect (Coq) guesses how to fill the holes. Note that the last proof above is closely related to the proof of `HilbertS3`, in terms of what is explicitly given to the software, and what is guessed by the software.

Below, an even shorter proof.

```
Lemma HilbertS5 : C.
Proof. exact: hAiBiC (hAiB _). Qed.
```

In the proof above, `(hAiB _)` pushes `B` on the top of the goal `C` by saying that the required argument `A` will be somehow provided later. Then it pushes the hypothesis `hAiBiC` on the goal and applies it, so we need to prove `A` and we are done. The `exact` terminator solves it trivially.

Lemmas that have been proved can be invoked, just as in the proof below.

```
Lemma HilbertS6 : C.
Proof. exact: HilbertS5. Qed.
```

Indeed `HilbertS5` belongs to the context although it is not displayed above the graphical line.

Below, we compare the different proofs of Hilbert's axiom S.

```
Print HilbertS5.
Print HilbertS2.
Print HilbertS.
Check HilbertS.

End HilbertSaxiom.

Print HilbertS5.
```

The command

```
Print HilbertS5.
```

prints out the proof term (witnessing lemma) `HilbertS5`, *i.e.* the object that is verified at `Qed` time. This object is as follows:

```
hAiBiC hA (hAiB hA)
```

It is of type `C`. Said otherwise, `hAiBiC hA (hAiB hA)` is a proof of proposition `C`. Printing `HilbertS2` shows that both objects are identical: the proof terms that are built and ultimately verified by the Coq kernel at `Qed`'s request are the same in both cases, although the ways to built it are different. The proof term of `HilbertS` is bigger than the others because the assumptions are all in the initial statement instead of being section/context variables.

The command

```
Check HilbertS.
```

prints out the type of `HilbertS`, that is, only the proposition that it witnesses. The command `Check` may help recall the actual statement of a given lemma.

After the Coq section `HilbertS` has been closed by the command

```
End HilbertSaxiom.
```

the section variables are *discharged*, *i.e.* lemmas proved within the section now explicitly account for the variables they use. That is why the command

```
Print HilbertS5.
```

does not produce the same output as before. Note that lemmas `HilbertS` and `HilbertS5` have the same proof term now.

2.2 Logical connectives

This new subsection proves a few basic lemmas about booleans and propositions.

The command

```
Print bool.
```

Displays the following:

```
Inductive bool : Set := true : bool | false : bool
```

This is how Coq defines/constructs the booleans (`bool`), *i.e.* by saying that `true` and `false` are of type `bool`, which implicitly means that a boolean is either `true` or `false`. So `true` and `false` are called *constructors*.

As we shall see in the remainder of the tutorial, the word `Inductive` may be involved in many more *inductive* (in a broad sense) definitions.

```
Section Symmetric_Conjunction_Disjunction.

Lemma andb_sym : forall A B : bool, A && B -> B && A.
Proof.
case.
  by case.
by [].
Qed.
```

The statement is actually

```
forall A B : bool, (A && B = true) -> (B && A = true)
```

where `&&` is (a notation for) a function expecting two arguments and computing the boolean conjunction of these. However the statement is represented by

```
forall A B : bool, A && B -> B && A
```

which is more readable. This ssreflect feature uses the Coq coercion mechanism and is defined in the file ssrbool.v of the ssreflect standard library.

Roughly speaking, the tactic `case` performs a case-splitting along the constructors of (the rightmost part of) the top of the proof stack whose type must be defined inductively. So the first use of `case` performs a case-splitting on `A` being `true` or `false`.

The tactic `by case` solves the first subgoal when `A` is `true` as follows: it case-splits on `B` being `true` or `false`, both subgoals being trivially solved by the tactical `by`.

The tactic `by []` solves trivially the second subgoal generated by the first use of `case`.

Below, the same lemma is proved using brute force.

```
Lemma andb_sym2 : forall A B : bool, A && B -> B && A.
Proof. by case; case. Qed.
```

In the proof above, the command

```
by case; case.
```

case-splits on `A` and then cases split on `B` (in each of the two subgoals), and the four subgoals are solved trivially. Note that this proof is one-line long as the proof of andb_sym is three-line long. It is because the ssreflect writing style prescribes that each line end with one full-stop, and that full-stop be only written there. This way, the interpreter validates one line and only one line at each interpretation step.

Below, the same lemma is proved using brute force again.

```
Lemma andb_sym3 : forall A B : bool, A && B -> B && A.
Proof. by do 2! case. Qed.
```

The command

```
do 2! case.
```

is equivalent to

```
case; case.
```

If there were ten boolean variables instead of two, one could write `do 10! case`. In this specific setting, this (roughly) simulates the `tauto` tactic without introducing any new tactic.

Now let there be two propositional variables and assume that they hold, as below.

```
Variables (C D : Prop) (hC : C) (hD : D).
```

The command

```
Check (and C D)
```

displays

```
C /\ D : Prop
```

which shows that `C /\ D` is the Coq notation for propositional conjunction `and C D`.

The command

```
Print and.
```

displays

```
Inductive and (A : Prop) (B : Prop) : Prop := conj : A -> B -> A /\ B
For conj: Arguments A, B are implicit
```

The first line above (the second line is explained in the next paragraph) shows that conjunction in Coq is defined inductively through one single constructor `conj`.

Checking the type of `conj` displays

```
conj : forall A B : Prop, A -> B -> A /\ B
```

which shows that `conj` expects two propositions `A` and `B` (hence `forall A B : Prop`), then proofs that the propositions hold, *i.e.* objects of type `A` and `B` (hence `A -> B -> ...`), and constructs/returns a proof that the conjunction of the two propositions also holds.

According to the explanation above, one would expect the term `conj C D hC hD` to be well-typed, but this is not the case. Indeed, it is possible to infer `C` from `hC` because `C` is the type of `hC`. (The same holds for `D` and `hD`.) In the definition of propositional conjunction in Coq, it is specified that the inferable arguments of a function should remain implicit (hence the line `For conj: Arguments A, B are implicit` in the paragraph above). Therefore, Coq considers the term `conj hC hD` well-typed instead of `conj C D hC hD`. When we want to explicitly give all the arguments of a function (even the ones defined as implicit), we use the `@` symbol: `@conj C D hC hD` is well-typed.

The command

```
Check (conj hC hD).
```

displays

```
conj hC hD : C /\ D
```

which shows that `conj hC hD` is a proof that `C /\ D` holds.

Below, we prove the propositional counterpart of the symmetry of conjunction that was proved in a boolean setting above.

```
Lemma and_sym : forall A B : Prop, A /\ B -> B /\ A.
Proof. by move=> A1 B []. Qed.
```

In the proof above, the command

```
move=> A1 B [].
```

is a shortcut for

```
move=> A1 B; case.
```

Note that when naming the introduced variables, one can either change the names of the universally quantified variables, *i.e.* use `A1` instead of `A`, or keep the same names, *i.e.* use `B`.

The `case` performs a "curryfication" in the goal

```
A1 /\ B -> B /\ A1
```

which becomes

```
A1 -> B -> B /\ A1
```

and is solved trivially. What actually happens is the following: `case` performs a case-splitting on (the rightmost part of) the top of the proof stack, *i.e.* `A1 /\ B`, which has only one constructor `conj`. So only one subgoal is generated. The goal assumption `A1 /\ B` can only be built through `conj` and the two arguments are of type `A1` and `B`, so `A1` and `B` are provided as assumptions on the proof stack so that "no information may be lost" in the case-splitting.

Let us now turn our attention to disjunction. Printing the object `or` displays the definition of the Coq disjunction, as follows.

```
Inductive or (A : Prop) (B : Prop) : Prop :=
    or_introl : A -> A \/ B | or_intror : B -> A \/ B
```

It involves two constructors because `A \/ B` may either derive from `A` or derive from `B`.

Checking the type of either constructor of `or` (by the command `Check or_introl.`) suggests that a constructor is the same kind of object as a lemma (just as `conj`). See below.

```
or_introl : forall A B : Prop, A -> A \/ B
```

Below, we prove the symmetry of the disjunction.

```
Lemma or_sym : forall A B : Prop, A \/ B -> B \/ A.
Proof. by move=> A B [hA | hB]; [apply: or_intror | apply: or_introl]. Qed.
```

In the proof above, the term

```
move=> A B [hA | hB]
```

is a shortcut for

```
move=> A B; case=> [hA | hB]
```

where `case=> [hA | hB]` case-splits on `A \/ B` (more specifically on the two constructors of `\/`) and `=> [hA | hB]` introduces and names the assumptions (more specifically the arguments of the constructors) in both subgoals `A -> B \/ A` and `B -> B \/ A` .

Then the syntax

```
[apply: or_intror | apply: or_introl]
```

applies the relevant constructor/lemma in both subgoals. More generally, the syntax

```
tactic0; [tactic1 | tactic2 | ...]
```

gives instructions for all the subgoals generated by `tactic0`. Note that the semi-colon is required between `tactic0` and `[tactic1 | tactic2 | ...]`.

Below, we prove the symmetry of the disjunction by translating the propositional statement into its boolean counterpart, which is easily proved using brute force. This proof technique is called *reflection*. Ssreflect's design allows for and ssreflect's spirit recommends wide use of such a technique. Since it is usually used locally to handle efficiently small parts of the proofs (instead of being used in the overall proof structure), this is called *small scale reflection*, hence the name ssreflect.

```
Lemma or_sym2 : forall A B : bool, A \/ B -> B \/ A.
Proof. by move=> [] [] AorB; apply/orP; move/orP : AorB. Qed.

End Symmetric_Conjunction_Disjunction.
```

In the proof above, the syntax

```
move=> [] [] AorB
```

is a shortcut for

```
case; case; move=> AorB.
```

Indeed the first `[]` yields a case-splitting on `A`, and the second `[]` yields a second case-splitting, this time on `B`, in *both* subgoals generated by the first `[]`. Then `AorB` introduces `A \/ B` in the context of the *four* subgoals corresponding to `A` (resp. `B`) being `true` or `false`.

The command

```
apply/orP.
```

rephrases the goal using the reflection lemma orP, which is defined in the file ssrbool.v. (Check it!) A reflection lemma is an equivalence property between the boolean world and the `Prop` world, and it is called a view when used in this way.

The command

```
move/orP : AorB.
```

is a shortcut for

```
move: AorB; move/orP.
```

where `move/orP` rephrases the top of the proof stack, *i.e.* `A \/ B`, using the reflection lemma orP. If the proof stack is empty, *i.e.* there is no universal quantifier or implication, the tactic fails.

Note that this kind of proof works with `A B : bool`, but not with `A B : Prop`.

Also, notice that since `move` is usually used with `=>` or `:` in order to pop objects from or push objects onto the top of the proof stack, `move/lemmaP` was designed to modify the top of the proof stack; since `apply` is usually used with `:` and considers the whole goal, `apply/lemmaP` was designed to modify the whole goal.

2.3 Two so-called paradoxes

This section uses the existential quantifier `ex` (print it!), which is defined inductively with one constructor and referred to as `exists`, its more convenient stage name.

The following shows that a symmetric and transitive relation is not irreflexive provided that it has at least one pair.

```
Section R_sym_trans.

Variables (D : Type) (R : D -> D -> Prop).

Hypothesis R_sym : forall x y, R x y -> R y x.

Hypothesis R_trans : forall x y z, R x y -> R y z -> R x z.

Lemma refl_if : forall x : D, (exists y, R x y) -> R x x.
Proof.
move=> x [y Rxy].
by apply: R_trans _ (R_sym _ y _).
Qed.

End R_sym_trans.
```

Above, `D` is a type (because it has type `Type`). This means that we may write `x : D`, which says that `x` has type `D`. (Set-theoretically, it reads "`x` belongs to `D`".) The object `R` is an arbitrary binary relation built on `D`. More specifically, `R` is a term expecting two inhabitants (*i.e.* elements) of `D` and returning a proposition (saying that the two inhabitants are related).

In the proof above, the command

```
move=> x [y Rxy].
```

is a shortcut for

```
move=> x; case=> y Rxy.
```

One could have written

```
move=> x [y] Rxy.
```

instead, but `[y Rxy]` is more informative than `[y] Rxy` since `Rxy` names a goal assumption generated by the case-splitting.

The command

```
by apply: R_trans (R_sym _ y _).
```

first builds a goal with holes, and then guesses the missing subterms so that the intermediate goal (after pushing `(R_sym _ y _)`) is indeed an instance of `R_trans`. Finally, the `by` finishes the job.

Below, we prove that in any bar with at least one person (who is called `d` below), there is a person such that if she is drinking then everybody is drinking. (Below `P x` means that `x` is drinking.)

```
Section Smullyan_drinker.

Variables (D : Type)(P : D -> Prop).

Hypothesis (d : D) (EM : forall A, A \/ ~A).

Lemma drinker : exists x, P x -> forall y, P y.
Proof.
case: (EM (exists y, ~P y)) => [[y notPy]| nonotPy]; first by exists y.
exists d => _ y; case: (EM (P y)) => // notPy.
by case: nonotPy; exists y.
Qed.

End Smullyan_drinker.
```

First note that the negation

```
~A
```

is a Coq notation for

```
A -> False
```

where the proposition `False` (not the boolean `false`!) is an inductive proposition with no constructor. (Print it!)

In the first line of the proof above, the command

```
case: (EM (exists y, ~P y)).
```

is a shortcut for

```
move: (EM (exists y, ~P y)); case.
```

It case-splits on

```
(exists y, ~P y) \/ ~(exists y, ~P y)
```

and generates two subgoals.

In subgoal 1, the command `[y notPy]` case-splits on the sole constructor of `exists` (also called `ex`), and introduces both the witness (called `y` for the occasion) and its non-drinking property. Then

```
first by exists y
```

says that in subgoal 1 (hence `first`), the non-drinking `y` is a witness for the existence property to be proved (hence `exists`), and that the system can trivially prove this fact (hence `by`). The proof actually uses the contradiction between `y` not drinking and the assumption on top of the goal stack.

In subgoal 2, the assumption that *nobody* is *not* drinking is introduced.

Note that `case: (EM (exists y, ~P y)) => [[y notPy]| nonotPy]` could be replaced with

```
have [[y notPy]| nonotPy] := EM (exists y, ~P y)
```

The tactic `have` will be introduced later on, but one can already see the point of using `have` here: the actual branching *i.e.* `[[y notPy]| nonotPy]`, is written in the beginning of the line whereas the justification occurs later. This is good because the branching says something about the proof structure, whereas its justification does not. This is consistent with the ssreflect writing style, which prescribes writing the key information of a line as early as possible for each line. (Ideally, a line being a meaningful compound proof step.)

In line 2, in the only remaining goal

```
exists d => _ y
```

says that `d` is a witness of the sought property (actually any other inhabitant of `D` would do) and the underscore `_` discards the assumption on the top of the proof stack. Then

```
case: (EM (P y)) => // notPy.
```

case-splits on `y` being drinking or not. Then

```
//
```

tries to solve trivial subgoals (among the two in the present situation). It solves the first one only. Then in the only remaining goal the assumption that `y` is not drinking is introduced. Note that also `//` is intended to solve trivial subgoals that were just generated, it would not fail if it did not solve any: `//` is not a terminator. Therefore it should not be used instead of a terminator when we want to underpin the proof structure.

In line 3,

```
case: nonotPy
```

case-splits on the constructors of the (rightmost part of) the top of the proof stack, which is the proposition `False` here. Indeed recall that `~A` stands for `A -> False`. Since `False` has no constructor, this generates no subgoal. However, in the current situation, one can invoke the assumption `False` only after proving the left-hand part of the implication

```
(exists y : D, ~ P y) -> False
```

This is an existential statement, which is proved using the `y` at hand.

2.4 Rewriting with (Leibniz) equalities and definitions

In this subsection, we present the ssreflect `rewrite` tactic, which is one of the main efficient features of ssreflect. In the examples we use natural numbers, *i.e.* terms typed in `nat`, but we do not yet need to explain how `nat` is defined.

```
Section Equality.

Variable f : nat -> nat.
```

```
Hypothesis f00 : f 0 = 0.

Lemma fkk : forall k, k = 0 -> f k = k.
Proof.
move=> k k0.
by rewrite k0.
Qed.
```

Like in Coq, the command

```
rewrite k0.
```

replaces `k` (the left-hand side of the equality `k0`) with `0` (the right-hand side) in the goal. The ssreflect `rewrite` is more general than the Coq `rewrite` as we shall see in the rest of the tutorial.

Below, we prove the same lemma faster.

```
Lemma fkk2 : forall k, k = 0 -> f k = k.
Proof. by move=> k ->. Qed.
```

The command

```
move=> k ->.
```

is a shortcut for

```
move=> k hyp; rewrite {} hyp.
```

where `rewrite {} hyp` rewrites with `hyp` and clears it from the context. The rewriting arrow is useful when `hyp` is only to be used once, *i.e.* most of the time.

The following exemplifies other features of `rewrite`.

```
Variable f10 : f 1 = f 0.

Lemma ff10 : f (f 1) = 0.
Proof. by rewrite f10 f00. Qed.
```

The command

```
rewrite f10 f00
```

is a shortcut for

```
rewrite f10; rewrite f00.
```

The proofs of the following two lemmas involve conversion from the Coq equality into boolean equality, and vice versa.

```
Variables (D : eqType) (x y : D).

Lemma eq_prop_bool : x = y -> x == y.
Proof. by move/eqP. Qed.

Lemma eq_bool_prop : x == y -> x = y.
Proof. by move/eqP. Qed.

End Equality.
```

An `eqType` is a type with decidable Leibniz (*i.e.* rewritable) equality. See the file `eqType.v` for details. The equality `x == y` is a boolean, whereas `x = y` is a proposition. Both proofs above invoke a reflection lemma saying that `x == y` and `x = y` are equivalent for `x` and `y` of type `D`. Note that `move/eqP` ”guesses” which implication of the equivalence is invoked.

The following shows how to rewrite definitions and also presents another use of the syntax `move/lemma`.

```
Section Using_Definition.

Variable U : Type.

Definition set := U -> Prop.

Definition subset (A B : set) := forall x, A x -> B x.

Definition transitive (T : Type) (R : T -> T -> Prop) :=
 forall x y z, R x y -> R y z -> R x z.
```

Above, the type `U` is the ”universe of the discourse” of what follows. Then the command

```
Definition set := U -> Prop.
```

defines a new type, *i.e.* `set`, using types that already exist, *i.e.* `U` and `Prop`. (Yes, `Prop` is a type, check it!) More specifically, `set` is the type of all the predicates on `U`. Set-theoretically speaking, this corresponds to the subsets of `U`.

In the definition

```
Definition subset (A B : set) := forall x, A x -> B x.
```

the object `x` need not be explicitly typed (we did not write `forall x : U`), whereas `A` and `B` must be typed: since the terms `A x` and `B x` are written and since `A` and `B` have type `U -> Prop`, the system infers that `x` must have type `U` for `A x` and `B x` to be well-typed. Note that the line above actually defines `subset A B` for two arbitrary parameters `A` and `B`, which implicitly defines `subset` itself.

```
Lemma subset_trans : transitive set subset.
Proof.
rewrite /transitive /subset => x y z subxy subyz t xt.
by apply: subyz; apply: subxy.
Qed.
```

In the proof above, the command

```
rewrite /transitive /subset.
```

replaces the objects `transitive` and `subset` with their actual definitions. This simulates the Coq command

```
unfold transitive, subset.
```

without introducing any new tactic, since it is relevant to provide `rewrite` with this new feature.

Note that

```
rewrite /transitive /subset => x y z subxy subyz t xt.
```

is a shortcut for

```
rewrite /transitive /subset; move=> x y z subxy subyz t xt.
```

Below, we prove the same lemma in a slightly different way, which gives the opportunity to present new features.

```
Lemma subset_trans2 : transitive set subset.
Proof.
move=> x y z subxy subyz t.
by move/subxy; move/subyz.
Qed.

End Using_Definition.
```

First note that we did not actually need to rewrite like in the previous proof: the introduction of variables with `move=>` forces the unfolding of the definition of `transitive`. Then the command

```
move/subxy.
```

transforms the top of the proof stack `x t` into `y t`. The transformation is valid for the following reason: since `x t -> y t` according to the term `subxy`, if we prove `y t -> z t` then we have proved `x t -> z t`. This view does not come from a reflection lemma as before, but from a half-equivalence, *i.e.* an implication. Therefore the view is one-way, whereas the views coming from reflection/equivalence lemmas are two-way.

3 Arithmetic for Euclidean division

This section shows a few assets of ssreflect for arithmetic.

Subsection 3.1 presents the natural numbers in Coq and the basic arithmetic operations (addition, multiplication) and relations (inequality). We point out the advantages of ssreflect, in terms of definitions and how these influence proof efficiency. Subsection 3.2 defines the Euclidean division. Subsection 3.3 proves that our division meets the requirements for Euclidean division and that Euclidean division is injective. Subsection 3.4 shows that the parametric inductive type families that we use for the specification of Euclidean division are not unique but are the most suitable.

3.1 Basics

In Coq the natural numbers are formally called `nat`. Checking the object `nat` displays the following line.

```
nat : Set
```

where `Set` is similar to `Type`, so that it is possible to write the following line below, which means that an arbitrary natural number is assumed and named `n`.

```
Variable n : nat.
```

The natural numbers are built in two steps, inductively: first, zero is a natural; second, the successor of a natural is also a natural. Printing the object `nat` displays the following line.

```
Inductive nat : Set := O : nat | S : nat -> nat
```

This means that `nat` has two constructors, which are named `O` and `S`, the letters of the alphabet. The constructor `O` has type `nat`, and it actually builds the first natural, *i.e.* zero, which is represented by the figure `0` for convenience. The second constructor of `nat`, *i.e.* `S`, expects a natural number, let us call it `n`, and returns a new natural number `S n` which is meant to be the successor of `n`. In ssreflect `S n` is usually written `n.+1`. Fortunately, the system understands the notation `3`, which represents `S(S(S O)))`, as suggested by the following lemma and its trivial proofs. Otherwise writing big numbers would not be practical.

```
Lemma three : S (S (S O)) = 3 /\ 2 = 0.+1.+1.
Proof. by []. Qed.
```

In Coq a primitive notion of fixpoint is defined. Thanks to fixpoints any objects defined inductively, *e.g.* `nat`, can be used to define recursive functions (whose domain is the inductively defined objects) and induction principles (used to prove predicates on the inductively defined objects).

For instance, the function `plus` (whose notation in Coq is `+`) expects two naturals and returns a natural, and it performs the usual addition over the natural. Printing it displays the following.

```
plus =
fix plus (n m : nat) {struct n} : nat :=
  match n with
  | 0 => m
  | p.+1 => (p + m).+1
  end
     : nat -> nat -> nat
```

We see from above that `plus` was defined by recursion thanks to a fixpoint, hence

```
fix
```

The function expects two arguments but

```
{struct n}
```

tells the system that the recursion is made on the first argument. So, when `plus` is given two arguments, it case-splits on the inductive definition of the first argument. The case-splitting is introduced by

```
match n with
```

and performed by

```
| 0 => m (* zero case *)
| p.+1 => (p + m).+1 (*non-zero case *)
```

If the first argument is zero, the result of the addition is the second argument, hence

```
| 0 => m
```

If the first argument is the successor of a given natural, the result is the successor of the addition of the given natural (the first argument is structurally decreasing, so the recursion is well-defined) and the same second argument, hence

```
| p.+1 => (p + m).+1
```

The function `plus` has an actual computational content, so `plus 16 64` reduces to `80` ”easily”. For example, try the proof below with and without the Coq tactic `simpl`. In this proof the point is not the lemma and its proof but only the effect of the tactic `simpl`.

```
Lemma concrete_plus : plus 16 64 = 80.
Proof. (* simpl. *) by []. Qed.
```

However, when using simplifying tactics to simplify a specific part of a goal expression, this may simplify unwanted parts too (such as the addition). Relying on a broad experience of tens of thousands of lines of proof script, ssreflect’s library was designed to avoid this side effect for specific (but not all) cases. In ssreflect, the natural addition function `addn` is equivalent to the following.

```
addn = nosimpl plus.
```

The object `nosimpl` is actually implemented in the file ssreflect.v, but we do not comment on it in this tutorial. In ssreflect, the notation `+` refers to `addn`. As one can see below, the proof of `16 + 64 = 80` is a trivial as before, but the point is that the Coq tactic `simpl` has no effect.

```
Lemma concrete_plus_bis : 16 + 64 = 80.
Proof. (* simpl. *) by []. Qed.
```

As shown below the natural multiplication, `mult` or `*` in Coq, is also defined recursively, along the two constructors of the naturals and referring to the addition.

```
mult =
fix mult (n m : nat) {struct n} : nat :=
  match n with
  | 0 => 0
  | p.+1 => (m + p * m)
  end
     : nat -> nat -> nat
```

Just as for addition, ssreflect blocks unwanted simplifications in the multiplication, `muln` or `*` in ssreflect, thanks to the `nosimpl` function, as below.

```
muln = nosimpl mult.
```

The following shows the advantage of defining `<=`, *i.e.* the less-than-or-equal-to relation, as a computable function, *i.e.* a function returning a boolean, instead of an inductive predicate in `Prop`. In Coq the natural inequality less-than-or-equal-to is also called `le`, and is defined as an inductive predicate. Printing it yields the following.

```
Inductive le (n : nat) : nat -> Prop :=
    le_n : (n <= n)
  | le_S : forall m : nat, (n <= m) -> (n <= m.+1)
```

By construction, a proof of `n <= p` may come from two different situations: either `p` equals `n` so that the first constructor allows building a proof of `n <= p`, or `p` is the successor of some `m` and we have a proof of `n <= m`, so that the second constructor allows building a proof of `n <= p`.

In order to prove that `1` is less than or equal to `3` one may use a transitivity lemma once and a lemma saying that any `n` is less than or equal to `n + 1` twice, as shown below.

```
Lemma concrete_le : le 1 3.
Proof. by apply: (Le.le_trans _ 2); apply: Le.le_n_Sn. Qed.
```

If the difference between the two naturals to be compared is big, one may proceed as follows.

```
Lemma concrete_big_le : le 16 64.
Proof. by auto 47 with arith. Qed.
```

The tactic `auto` tries to build a proof automatically (which is more ambitious than "just" verifying a proof). The number `47` is a parameter of the depth of the search and `arith` is the name of a pool of lemmas that `auto` is allowed to invoke. The resulting proof script is indeed small, at the cost of an elaborate (and potentially slow) tactic. Moreover, although the proof script is small, the lemma that it builds is big. (Print it!)

In ssreflect, `<=` is implemented as a Boolean predicate, as follows.

```
leq = fun m n : nat => m - n == 0
     : nat -> nat -> bool
```

where `-` is a notation for the natural "non-simplifying" subtraction below.

```
subn = nosimpl subn_rec.
```

where `subn_rec` is defined as follows with the `Fixpoint` syntax, which is equivalent to the `Definition-fix` syntax, and more synthetic when usable.

```
Fixpoint subn_rec (m n : nat) {struct m} :=
  match m, n with
  | m'.+1, n'.+1 => (m' - n')
  | _, _ => m
  end
```

This double `match` allows case-splitting along `m` and `n` being zero or not, which sums up to four cases altogether. The first line of the body of the `match-with`, namely

```
| m'.+1, n'.+1 => (m' - n')
```

accounts for when both naturals are non-zero. The second line, namely

```
| _, _ => m
```

accounts for all the cases that have not been accounted for before this line. In the specific specific situation at hand, it accounts for when at least one of the naturals is zero.

This way, through subtraction and `leq`, the concept of natural inequality is given a computational content, and a proof script of concrete inequality may involve a basic tactic only.

```
Lemma concrete_big_leq : 0 <= 51.
Proof. by []. Qed.
```

Moreover, the lemma itself is as small as its proof. (Print it!) It involves only a coercion of Boolean to `Prop`.

A few semi-concrete-semi-abstract lemmas involving inequality may also be proved by plain computation.

```
Lemma semi_concrete_leq : forall n m, n <= m -> 51 + n <= 51 + m.
Proof. by []. Qed.
```

The lemma above has a trivial proof because `51 + n <= 51 + m` reduces to `50 + n <= 50 + m`, and ultimately to `n <= m`. The corresponding lemma using Coq definitions and tactics would require a structured proof, whereas, in ssreflect, most of the proof burden is born by a simple computation. This feature is not limited to arithmetic inequality: it is one of the main motivations of ssreflect's design. For instance Boolean connectives are also made for computing, as shown below.

```
Lemma concrete_arith : (50 < 100) && (3 + 4 < 3 * 4 <= 17 - 2).
Proof. by []. Qed.
```

For any objects defined inductively, Coq generates an induction principle automatically. For `nat`, the generated induction principle `nat_ind` is the usual one. Checking `nat_ind` displays the following.

```
nat_ind
     : forall P : nat -> Prop,
       P 0 -> (forall n : nat, P n -> P n.+1) -> forall n : nat, P n
```

Literally, `nat_ind` states that for every predicate `P` over the naturals the following holds: if `P 0` holds, if `P n` implies `P n.+1` for every natural `n`, then the predicate holds for all naturals.

The following proves the commutativity of the addition of natural numbers.

```
Lemma plus_commute : forall n1 m1, n1 + m1 = m1 + n1.
Proof.
by elim=> [| n IHn m]; [elim | rewrite -[n.+1 + m]/(n + m).+1 IHn; elim: m].
Qed.
```

The tactic

```
elim
```

performs an induction on the top universally-quantified variable of the goal (along the induction principle generated automatically for the type of the variable), which is possible only when the type of the variable has been defined inductively. So in the proof above, the first `elim` performs an induction on the natural `n1` (along the usual induction principle for naturals). So this `elim` is an extension of

```
apply: nat_ind.
```

Put otherwise, `elim` case-splits on an object of an inductive type by considering the different constructors alternatively, just like the tactic `case`. In addition `elim` assumes the induction hypothesis, which `case` does not. So, `elim` is strictly more powerful. Still, the ssreflect writing style prescribes using `case` whenever possible to make the proof script more informative: a proof that mentions the induction hypothesis without actually using it may be confusing.

Then

```
=> [| n IHn m]
```

introduces no variable in the first subgoal, where `n1` is `0`, hence `[|`, and it introduces the natural `n`, the induction hypothesis about commutativity, and the natural `m` in the second subgoal.

Then

```
[elim |
```

performs an induction on `m1` in the first subgoal, both branches being solved trivially thanks to the `by` at the beginning of the line.

In the second subgoal,

```
rewrite -[n.+1 + m]/(n + m).+1 IHn
```

rewrites `n.+1 + m` into `(n + m).+1`. This works because the two formulas are convertible: they compute to the same value and generates no additional subgoal. More precisely, `rewrite /...` instructs to replace an expression by this definition, the `-` instructs to do the inverse, then to make the formula appear, and the fragment `[...]` indicates where this should happen. Then, the `rewrite` tactic rewrites with the equality of the induction hypothesis named `IHn`. Note that although there is a quantifier before the equality, the `rewrite` tactic uses a pattern matching algorithm to find a matching pattern (In the current goal there is only one such pattern) and rewrites all of its instances (In the current goal there is only one such instance).

Then, still in the second subgoal,

```
elim: m
```

which is a shorthand for

```
move: m; elim
```

performs an induction on `m`, both branches being solved trivially thanks to the `by` at the beginning of the line.

3.2 Definitions

This section explains how the euclidean division is implemented in the file div.v of the ssreflect standard library, and it mentions how it may be used.

First is defined the function `edivn_rec` whose type is as follows.

```
nat -> nat -> nat -> nat * nat
```

So `edivn_rec` expects three natural arguments and returns a pair of naturals. The actual definition is

```
Definition edivn_rec d := fix loop (m q : nat) {struct m} :=
  if m - d is m'.+1 then loop m' q.+1 else (q, m).
```

The function `edivn_rec` above is defined through an auxiliary function `loop` which is defined recursively. The first argument of `edivn_rec`, namely `d`, can be defined as a parameter because its value remains constant throughout the recursion of `loop`. The last two arguments of `edivn_rec` are the two arguments of `loop`, namely `m` and `q`. As specified by

```
{struct m}
```

the recursion of `loop` uses `m`, so Coq will accept the recursive definition of `loop` *iff* it complies with the Coq sufficient conditions guaranteeing that the first argument `m` is actually decreasing structurally. The body of the recursive definition involves the ssreflect syntax

```
if-is-then-else
```

which corresponds to the Coq syntax

```
match-with
```

The `if-is-then-else` syntax is designed to provide one-line, readable pattern matching.

The recursion says that if the first argument `m` minus the parameter `d` is not zero (*i.e.* `m` is greater than `d.+1`), the function `loop` is re-invoked with another first argument, *i.e.* `m - (d.+1)` (recognised by Coq as structurally smaller than `m`) and the second argument incremented by 1; if `m` minus `d` is zero, the function returns both arguments in a pair. The parameter `d` is meant to be the predecessor of the divisor of the Euclidean division defined afterwards; the argument `m` is meant to be the remainder; and the argument `q` is meant to be the quotient.

Below the auxiliary function `edivn_rec` is used in the definition of the actual Euclidean division `edivn`, which expects two arguments, a dividend and a divisor, and returns the quotient and the remainder.

```
Definition edivn m d := if d > 0 then edivn_rec d.-1 m 0 else (0, m).
```

The function `edivn` expecting the divisor as an argument invokes `edivn_rec` with the predecessor of the divisor, and `edivn_rec` actually subtracts the successor of its argument. This may sound awkward at first glance: why first subtract `1` to the divisor (`d.-1`) and then add `1` back again (`m - d is m'.+1`)? Actually, it makes much sense. Indeed, in the definition of `edivn` this does not cost anything because, in the first place, the function needs to case-split on the divisor being zero or not. Without the subtraction, the function would be as follows.

```
Definition edivn2 m d := if d > 0 then edivn_rec2 d m 0 else (0, m).
```

In `edivn_rec` the subtraction-addition allows case-splitting on the temporary remainder being less than the divisor or not, through a simple subtraction. If using `edivn2`, we would need to use an auxiliary function like below.

```
Definition edivn_rec2 d := fix loop (m q : nat) {struct m} :=
  if m - (d - 1) is m'.+1 then loop m' q.+1 else (q, m).
```

Indeed, consider the following function with the same recursive structure as `edivn_rec`.

```
Definition edivn_rec3 d := fix loop (m q : nat) {struct m} :=
  if m - d is m'.+1 then term1 else term2.
```

where `term1` and `term2` are arbitrary. `edivn_rec3` could not work with `edivn2`: when `m - d` is zero, it may be either because they are equal, in which case we need one more recursive call, or because `m` is less than `d`, in which case we can return the values now.

In order to prove that `edivn_rec` actually performs the Euclidean division, we need to specify what we mean by Euclidean division first. This is done with the definition below.

```
CoInductive edivn_spec (m d : nat) : nat * nat -> Type :=
  EdivnSpec q r of m = q * d + r & (d > 0) ==> (r < d) :
   edivn_spec m d (q, r).
```

The difference between `CoInductive` and `Inductive` is that `CoInductive` does not try to generate a recursion or induction principle. Since the definition above is not intended to be used with such principles, using `CoInductive` ensures that it will not be so. The definition above defines parametric families of types. The parameters are `m` and `d`. For given `m` and `d`, the type family expects a pair of naturals and returns a `Type`. It builds (and returns) a `Type` as follows: to the constructor `EdivnSpec`, it gives two naturals `q` and `r`, a proof of `m = q * d + r`, and a proof of `(d > 0) ==> (r < d)`. Then the resulting object has type `edivn_spec m d (q, r)`. The `of-&`

syntax is from ssreflect. To define the same type in a Coq style with exactly the same inhabitants (*i.e.*, "elements"), we could write the following.

```
CoInductive edivn_spec (m d : nat) : nat * nat -> Type :=
  EdivnSpec :
   forall q r, m = q * d + r -> (d > 0 -> r < d) -> edivn_spec m d (q, r).
```

3.3 Results

In the remainder of this section we are going to see two results. The first one says that the function `edivn` meets the specification `edivn_spec` and thus performs *one* Euclidean division. The second result says that if some naturals meet the specifications then they must be the ones returned by `edivn`. This shows uniqueness of the Euclidean division.

Below, let us prove that `edivn` complies with `edivn_spec`.

```
Lemma edivnP : forall m d, edivn_spec m d (edivn m d).
Proof.
rewrite /edivn => m [|d] //=; rewrite -{1}[m]/(0 * d.+1 + m).
elim: m {-2}m 0 (leqnn m) => [|n IHn] [|m] q //=; rewrite ltnS => le_mn.
rewrite subn_if_gt; case: ltnP => [// | le_dm].
rewrite -{1}(subnK le_dm) -addSn addnA -mulSnr; apply: IHn.
apply: leq_trans le_mn; exact: leq_subr.
Qed.
```

In the first line of the proof, the pattern

```
[|d]
```

case-splits on `d`. The first case is solved by the tactic

```
//=
```

which is actually a combination of `//` and

```
/=
```

We have already seen that `//` solves trivial subgoals (generated by `[|d]` in this specific example). As for `/=`, it simplifies all subgoals (generated by `[|d]` in this specific example), like `simpl` in Coq. So `//=` simplifies all subgoals and solves the trivial ones. Then in the only remaining subgoal, *i.e.* the second one, we replace the first occurrence of `m` with `0 * d.+1 + m`, using

```
rewrite -{1}[m]/(0 * d.+1 + m)
```

The tactic succeeds because `0 * d.+1 + m` is reducible to `m`. The reason why we replace `m` with the more complicated `0 * d.+1 + m` will become clear below.

Before the second line of the proof, the goal is

```
edivn_spec (0 * d.+1 + m) d.+1 (edivn_rec d m 0)
```

We need a proof by strong induction rather than simple induction: instead of using

```
nat_ind
     : forall P : nat -> Prop,
       P 0 -> (forall n : nat, P n -> P n.+1) -> forall n : nat, P n
```

we invoke a property resembling the generic strong induction principle below.

```
forall P : nat -> Prop,
  P 0 -> (forall n, (forall k, k <= n -> P k) -> P n.+1) -> forall n, P n
```

Depending on the situation, the strong induction principle that is required may vary slightly, so there is no strong induction principle that suits every situation (even in the specific case of naturals). Nonetheless this is not an issue in ssreflect because strong induction can be invoked in one line.

In this second line of the proof,

```
(leqnn m)
```

pushes `m <= m` on top of the proof stack, and just to the left of `(leqnn m)`

```
0
```

generalises `0` in an arbitrary natural `n`. So, in order to prove what we want, we first prove a more general property. This was the purpose of the mysterious `rewrite` in the first line of the proof. Then

```
{-2}m
```

generalises every occurrence of `m` in the goal (`m` corresponds to `k` in the generic strong induction principle above) except the second occurrence, which is intended to correspond to the upper bound `n` in the generic strong induction principle above. Finally

```
elim: m
```

starts an induction on the upper bound `m`, which amounts to the strong induction that we needed. This `elim: m` generates two subgoals. Thanks to

```
[|n IHn]
```

we introduce an upper bound and the corresponding induction hypothesis in the second subgoal. Then in both subgoals, the

```
[|m]
```

case-splits on the natural below the upper bound and in the four (new) subgoals

```
q
```

introduces the "generalised `0`". Three subgoals out of the four are solved by `//=`. Finally,

```
rewrite ltnS
```

rewrites a strict inequality `<` into a non-strict inequality `<=` thanks to the lemma `ltnS`.

At the third line of the proof, the goal is

```
edivn_spec (q * d.+1 + m.+1) d.+1
     match m.+1 - d with
     | 0 => (q, m.+1)
     | m'.+1 => edivn_rec d m' q.+1
     end
```

Thanks to the lemma `subn_if_gt` (and thanks to inequality being defined through subtraction in ssreflect) we rewrite the `match-with` syntax into the Coq `if-then-else` syntax. The next proof step is

```
case: ltnP
```

To understand what this does let us first check `ltnP`.

```
ltnP : forall m n : nat, ltn_xor_geq m n (n <= m) (m < n)
```

And then print `ltn_xor_geq`.

```
CoInductive ltn_xor_geq (m : nat) (n : nat) : bool -> bool -> Set :=
    LtnNotGeq : m < n -> ltn_xor_geq m n false true
  | GeqNotLtn : n <= m -> ltn_xor_geq m n true false
```

Here again, a coinductive is preferred to an inductive to build a non-recursive type family without generating an inductive principle. If the first constructor `LtnNotGeq` is given a proof of `m < n` then it returns an object of type `ltn_xor_geq m n false true`. If the second constructor `GeqNotLtn` is given a proof of `n <= m` then it returns an object of type `ltn_xor_geq m n true false`. Technically, `ltnP m d` means that the pair `(m < d, d <= m)` is `(true,false)` if `m < d`, and `(false,true)` if `d <= m`. Thus lemma `ltnP` says that for all naturals `m` and `n`, either `m < n` or `n <= m` in a mutually exclusive way, which is useful here. The command `case: ltnP` case-splits on such a disjunction. Since lemma `ltnP` is given no parameter, a pattern matching algorithm (different from the one used in the `rewrite` tactic) finds two relevant naturals for the case-splitting. Here it case-splits on `m < d` or `d <= m`, which is in the present situation more convenient than the disjunction `m < d` or `~m < d`.

Writing `case: (ltnP m d)` instead would work just as well without relying on the pattern matching algorithm.

Then, note that

```
[// | le_dm]
```

could be shortened into

```
// le_dm
```

but the slightly longer command explains that the trivial subgoal is the first one, which the shorter command does not suggest.

At the fourth line of the proof, the goal is

```
edivn_spec (q * d.+1 + m.+1) d.+1 (edivn_rec d (m - d) q.+1)
```

By writing

```
rewrite -addSn
```

we find in the goal the first pattern that matches the equality of lemma `addSn` and rewrites in the goal all occurrences of this pattern using the equality from right to left (hence the minus symbol). Instead

```
rewrite -{1}(subnK le_dm)
```

rewrites, from right to left, only the first occurrence (hence `{1}`) of the first pattern that matches the equality of lemma `subnK le_dm`. At this stage of the tutorial, the rest of the proof is understandable without further explanations.

The following lemma and its proof show that for any divisor, quotient, and remainder that is less than the divisor, `edivn` with first argument the remainder plus the quotient times the divisor and second argument the divisor returns the quotient and the remainder. This proves that the specifications for the Euclidean division are non-ambiguous.

```
Lemma edivn_eq : forall d q r, r < d -> edivn (q * d + r) d = (q, r).
Proof.
move=> d q r lt_rd; have d_pos: 0 < d by exact: leq_trans lt_rd.
case: edivnP lt_rd => q' r'; rewrite d_pos /=.
wlog: q q' r r' / q <= q' by case (ltnP q q'); last symmetry; eauto.
rewrite leq_eqVlt; case: eqP => [-> _|_] /=; first by move/addnI->.
rewrite -(leq_pmul2r d_pos); move/leq_add=> Hqr Eqr _; move/Hqr {Hqr}.
by rewrite addnS ltnNge mulSn -addnA Eqr addnCA addnA leq_addr.
Qed.
```

In the first line of the proof above, the command

```
have d_pos: 0 < d by exact: leq_trans lt_rd
```

asserts that `0 < d`, proves it by

```
by exact: leq_trans lt_rd
```

then names this proved assertion `d_pos` and puts it in the context.

Second line of the proof, the command

```
case: edivnP lt_rd
```

Pushes first hypothesis `lt_rd` stating `r < d`, then lemma `edivnP` on the top of the proof stack. Then it case-splits on lemma `edivnP`. (It may do so since the conclusion of lemma `edivnP` is the inductively defined `edivn_spec`.) It is a shortcut for

```
move: edivnP lt_rd; case.
```

and the subgoal after `move: edivnP lt_rd` is

```
 (forall m d0 : nat, edivn_spec m d0 (edivn m d0)) ->
   r < d -> edivn (q * d + r) d = (q, r)
```

Since no argument is given to lemma `edivnP` before the `case`, since lemma `edivnP` involves the term `edivn m d0`, and since the goal involves the term `edivn (q * d + r) d`, the pattern matching algorithm (specific to the `case` tactic) decides to instantiate `m` with `(q * d + r)` and `d0` with `d`. So, alternatively, we could have written

```
case: (edivnP (q * d + r) d) lt_rd.
```

or even

```
move: (edivnP (q * d + r) d) lt_rd; case.
```

to understand better what is happening. After `move: (edivnP (q * d + r) d) lt_rd`, the goal is

```
 edivn_spec (q * d + r) d (edivn (q * d + r) d) ->
   r < d -> edivn (q * d + r) d = (q, r)
```

Since `edivn_spec` has only one constructor, the `case` tactic generates no additional subgoal. What happens in the goal, roughly, is that `edivn_spec (q * d + r) d (edivn (q * d + r) d)` is replaced with the conditions of construction of `edivn_spec`. Since the last argument of `edivn_spec` is a pair of natural, all occurrences of `edivn (q * d + r) d` are rewritten into the relevant `(q0,r0)`.

Then the command

```
rewrite d_pos /=
```

is equivalent to

```
rewrite d_pos => /=
```

In the example above only one `=>` is spared, but it is especially useful when we need to rewrite again after the simplification.

Third line of the proof, the goal is

```
q * d + r = q' * d + r' -> r' < d -> r < d -> (q', r') = (q, r)
```

The command

```
wlog: q q' r r' / q <= q'
```

(where `wlog` is a shorthand for the tactic `without loss`) states that we can assume `q <= q'` without loss of generality. More specifically, the original unique goal becomes the second subgoal where the assumption `q <= q'` has been pushed on the proof stack, and a first subgoal is generated, which states that if for all `q`, `q'`, `r`, and `r'` the inequality assumption implies the property, then the property holds. The sequence

```
move: r r'; wlog: q q' / q <= q'
```

would do too, but

```
move: q q' r r'; wlog: / q <= q'
```

although valid, would not help since the assumption `q <= q'` must be inserted before the `q` and `q'` variables are generalised. Therefore the tactic `wlog` cannot be easily simulated using other Coq tactics or even ssreflect tactics that we have seen so far: it has a substantial added value.

The tactics `symmetry` (inverting equalities) and `eauto` (automatic basic proof searching) are both Coq tactics.

Fourth line of the proof:

```
move/addnI
```

simplifies the top of the proof stack using injectivity of the addition.

The rest of the proof should not require detailed explanations at this stage of the tutorial.

3.4 Parametric type families and alternative specifications

Now we explain why the arguments of `edivn_spec` whose type is

```
nat -> nat -> nat * nat -> Set
```

are not all mentioned at the same stage of the definition of `edivn_spec`. Indeed the first two arguments are parameters and the last one, the pair of naturals, makes `edivn_spec m d` an inductively defined type family. The chosen definition is the following one.

```
CoInductive edivn_spec (m d : nat) : nat * nat -> Type :=
  EdivnSpec q r of m = q * d + r & (d > 0) ==> (r < d) :
  edivn_spec m d (q, r).
```

Two alternative definitions are presented below. The first one is

```
CoInductive edivn_spec_right : nat -> nat -> nat * nat -> Type :=
  EdivnSpec_right m d q r of m = q * d + r & (d > 0) ==> (r < d) :
  edivn_spec_right m d (q, r).
```

It shifts the two parameters to the right of the colon `:`, so that `edivn_spec_right` has the same type as `edivn_spec`. There is no longer any parameter, but the alternative type constructor `EdivnSpec_right` requires two more arguments than `EdivnSpec`, namely `m` and `d`.

The second alternative definition is

```
CoInductive edivn_spec_left (m d : nat)(qr : nat * nat) : Type :=
EdivnSpec_left of m = (fst qr) * d + (snd qr) & (d > 0) ==> (snd qr < d) :
   edivn_spec_left m d qr.
```

It shifts the pair argument to the left of the colon `:`, so that every argument is seen as a parameter and `edivn_spec_left` has the same type as `edivn_spec`. So `EdivnSpec_left` requires two less arguments than `EdivnSpec`. However, the last parameter of `edivn_spec_left` being a pair, we need to project it before using it in the body of the definition.

The two lemmas

```
Lemma edivnP_right : forall m d, edivn_spec_right m d (edivn m d).
```

and

```
Lemma edivnP_left : forall m d, edivn_spec_left m d (edivn m d).
```

can be proved with the proof of lemma `edivnP`. However it gets more complicated for the lemmas corresponding to lemma `edivn_eq`, as shown by their proofs.

```
Lemma edivn_eq_right : forall d q r, r < d -> edivn (q * d + r) d = (q, r).
Proof.
move=> d q r lt_rd; have d_pos: 0 < d by exact: leq_trans lt_rd.
set m := q * d + r; have: m = q * d + r by [].
set d' := d; have: d' = d by [].
case: (edivnP_right m d') => {m d'} m d' q' r' -> lt_r'd' d'd q'd'r'.
```

```
move: q'd'r' lt_r'd' lt_rd; rewrite d'd d_pos {d'd m} /=.
wlog: q q' r r' / q <= q' by case (ltnP q q'); last symmetry; eauto.
rewrite leq_eqVlt; case: eqP => [-> _|_] /=; first by move/addnI->.
rewrite -(leq_pmul2r d_pos); move/leq_add=> Hqr Eqr _; move/Hqr {Hqr}.
by rewrite addnS ltnNge mulSn -addnA -Eqr addnCA addnA leq_addr.
Qed.
```

Beside lines 2 to 5 and `-Eq` on the last line, the proof above is similar to the proof of lemma `edivn_eq`. The second line of the proof,

```
set m := q * d + r
```

defines `m` as `q * d + r` and replaces `q * d + r` with `m` in the goal. The two `have` push the corresponding equalities on the proof stack. These four operations intend to keep track of `d` and `q * d + r` after the `case` which replaces them with fresh universally quantified variables because the first two arguments of `edivn_spec_right` are no longer (compare to `edivn_spec`) parametric. Line 4, note that `edivnP_right` needs arguments. Lines 4 and 5, `{m d'}` and `{d'd m}` clear useless context variables and hypotheses.

```
Lemma edivn_eq_left : forall d q r, r < d -> edivn (q * d + r) d = (q, r).
Proof.
move=> d q r lt_rd; have d_pos: 0 < d by exact: leq_trans lt_rd.
case: (edivnP_left (q * d + r) d) lt_rd; rewrite d_pos /=.
set q':= (edivn (q * d + r) d).1; set r':= (edivn (q * d + r) d).2.
rewrite (surjective_pairing (edivn (q * d + r) d)) -/q' -/r'.
wlog: q r q' r' / q <= q' by case (ltnP q q'); last symmetry; eauto.
rewrite leq_eqVlt; case: eqP => [-> _|_] /=; first by move/addnI->.
rewrite -(leq_pmul2r d_pos); move/leq_add=> Hqr Eqr _; move/Hqr {Hqr}.
by rewrite addnS ltnNge mulSn -addnA Eqr addnCA addnA leq_addr.
Qed.
```

The proof above is similar to the proof of lemma `edivn_eq`, beside lines 3 and 4, and `edivnP_left` needing arguments for the case-splitting, and the variables reordering in `wlog: q r q' r'`. Line 3 the command

```
set q':= (edivn (q * d + r) d).1
```

defines `q'` as the first component of the pair `edivn (q * d + r) d`.
Then

```
rewrite (surjective_pairing (edivn (q * d + r) d)) -/q' -/r'
```

decomposes the pair explicitly and folds the definitions of `q'` and `r'`, which enables the *without loss of generality* subsequent proof structure.

This example suggests that, in the definition of `edivn_spec`, the choice to define some arguments as parametric (to the left of the typing colon) and some other arguments as non-parametric (to the right of the typing colon) is made in order to minimise the proof burden. Generally speaking, such a choice should be carefully made, while bearing in mind that parametric type families that are equal in extension may not be equivalent in terms of usability. Experience will tell.

4 Conclusion

By shortening the Coq tutorial and coping with Euclidean division, ssreflect has proved to be a synthetic and efficient proof language. The reader may look at the ssreflect standard library to find more examples or even to practise by reproving the results therein. This is recommended.

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-0803