INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

An introduction to small scale reflection in Coq

Georges Gonthier — Assia Mahboubi

N° 7392

September 2010

Domaine 2

ISSN 0249-6399 ISRN INRIA/RR--7392--FR+ENG

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

INRIA

centre de recherche
SACLAY – ÎLE-DE-FRANCE

An introduction to small scale reflection in Coq

Georges Gonthier , Assia Mahboubi*

Domaine : Algorithmique, programmation, logiciels et architectures
Équipes-Projets TypiCal, Inria Microsoft Joint Centre

Rapport de recherche n° 7392 — September 2010 — 60 pages

Abstract: This tutorial presents the Ssreflect extension to the Coq system. This extension consists of an extension to the Coq language of script, and of a set of libraries, originating from the formal proof of the Four Color theorem. This tutorial proposes a guided tour in some of the basic libraries distributed in the Ssreflect package. It focuses on the application of the small scale reflection methodology to the formalization of finite objects in intuitionistic type theory.

Key-words: Coq, SSReflect, reflection, type theory

* This work has been partially funded by the FORMATH project, nr. 243847, of the FET program within the 7th Framework program of the European Commission

Centre de recherche INRIA Saclay – Île-de-France
Parc Orsay Université
4, rue Jacques Monod, 91893 ORSAY Cedex
Téléphone : +33 1 72 92 59 00

An introduction to small scale reflection in Coq

Résumé : Ce tutoriel présente l'extention Ssreflect pour le system Coq. Cette extention comprend un language de script et unensemble ed bibliothèques, originellement dvéloppées pour la preuve formelle du théorème des Quatre Couleurs. Ce tutoriel propose une introduction aux bibliothèques de base de la distribution de l'extention Ssreflect. Il se concentre sur l'application de la méthodologie de réflexion à petite échelle à la formalisation d'objets finis en théorie des types intuitioniste.

Mots-clés : Coq, SSReflect, réflexion, théorie des types

Contents

1 Introduction **3**

2 Instructions for the exercises **5**

3 A script language for structured proofs **7**
3.1 Sequents as stacks . . . 7
3.2 Control flow . . . 10
3.2.1 Indentation and bullets . . . 10
3.2.2 Terminators . . . 11
3.3 Goal selectors . . . 14
3.4 Forward chaining, backward chaining . . . 15
3.5 More SSReflect features, on an example . . . 18

4 Small scale reflection, first examples **21**
4.1 The two sides of deduction . . . 21
4.1.1 Interpreting assumptions . . . 22
4.1.2 Specializing assumptions . . . 23
4.1.3 Interpreting goals . . . 24
4.1.4 The `reflect` predicate . . . 24
4.1.5 Interpreting equivalences . . . 25
4.1.6 Proving `reflect` equivalences . . . 26
4.2 Exercises: sequences . . . 27
4.3 Exercises: Boolean equalities . . . 29

5 Type inference using canonical structures **31**
5.1 Canonical Structures . . . 31
5.2 Canonical constructions . . . 35
5.3 Predtypes: canonical structures for notations . . . 37

6 Finite objects in SSReflect **39**
6.1 Finite types . . . 39
6.1.1 Finite types constructions . . . 39
6.1.2 Cardinality, set operations . . . 40
6.1.3 boolean quantifiers . . . 42
6.1.4 Example: a depth first search algorithm . . . 43
6.2 Sigma types with decidable specifications . . . 48
6.3 Finite functions, finite sets . . . 51

7 Appendix: Checking, searching, displaying information **55**
7.0.1 Check . . . 55
7.0.2 Display . . . 55
7.0.3 Search . . . 55

1 Introduction

Small-scale reflection is a formal proof methodology based on the pervasive use of computation with symbolic representations. Symbolic representations are usually hidden in traditional computational reflection (e.g., as used in the

Coq[The10] `ring`, or `romega`): they are generated on-the-fly by some heuristic algorithm and directly fed to some decision or simplification procedure whose output is translated back to "logical" form before being displayed to the user. By contrast, in small-scale reflection symbolic representations are ubiquitous; the statements of many top-level lemmas, and of most proof subgoals, explicitly contain symbolic representations; translation between logical and symbolic representations is performed under the explicit, fine-grained control of the proof script.

The efficiency of small-scale reflection hinges on the fact that fixing a particular symbolic representation strongly directs the behavior of a theorem-prover:

- Logical case analysis is done by enumerating the symbols according to their inductive type: the representation describes which cases should be considered.

- Many logical functions and predicates are represented by concrete functions on the symbolic representation, which can be computed once (part of) the symbolic representation of objects is known: the representation describes what should be done in each case.

Thus by controlling the representation we also control the automated behavior of the theorem prover, which can be quite complex, for example if a predicate is represented by a sophisticated decision procedure. The real strength of small-scale reflection, however, is that even very simple representations provide useful procedures. For example, the truth-table representation of connectives, evaluated left-to-right on the Boolean representation of propositions, provides sufficient automation for most propositional reasoning.

Small-scale reflection defines a basis for dividing the proof workload between the user and the prover: the prover engine provides computation and database functions (via partial evaluation, and definition and type lookup, respectively), and the user script guides the execution of these functions, step by step. User scripts comprise three kinds of steps:

- Deduction steps directly specify part of the construction of the proof, either top down (so-called forward steps), or bottom-up (backward steps). A reflection step that switches between logical and symbolic representation is just a special kind of deductive step.

- Bookkeeping steps manage the proof context, introducing, renaming, discharging, or splitting constants and assumptions. Case-splitting on symbolic representations is an efficient way to drive the prover engine, because most of the data required for the splitting can be retrieved from the representation type, and because specializing a single representation often triggers the evaluation of several representation functions.

- Rewriting steps use equations to locally change parts of the goal or assumptions. Rewriting is often used to complement partial evaluation, bypassing unknown parameters (e.g., simplifying `(b && false)` to `false`). Obviously, it's also used to implement equational reasoning at the logical level, for instance, switching to a different representation.

It is a characteristic of the small-scale reflection style that the three kinds of steps are roughly equinumerous, and interleaved; there are strong reasons for this, chief among them the fact that goals and contexts tend to grow rapidly through the partial evaluation of representations. This makes it impractical to embed most intermediate goals in the proof script - the so-called declarative style of proof, which hinges on the exclusive use of forward steps. This also means that subterm selection, especially in rewriting, is often an issue.

The basic Coq tactic language is not well adapted as such to small-scale reflection proofs. It is heavily biased towards backward steps, with little support for forward steps, or even script layout. Many of the basic tactics implement fragile context manipulation heuristics which hinder precise bookkeeping; on the other hand the under-utilized "intro patterns" provide excellent support for case splitting.

In the present document, we briefly summarize in the two first sections some salient aspects of the SSReflect extension to the Coq language of script which originates from the proof of the Four Color theorem [Gon08]. The SSReflect language in itself can be considered as an alternative idiom to the one proposed by the standard distribution of the Coq system, introducing some improvements (term selection, enhanced rewriting, robustness of scripts,...). But the main contribution of the research line drawn by the successful formal proof of the Four Color theorem lies in the small scale reflection methodology. The two last sections of this tutorial propose a guided tour in the basic libraries distributed with the SSReflect extension. The aim of this presentation is to set out the design patterns which govern the definition of objects, and the structure of the theories developed on these objects, including the crucial use of type classes and canonical structures. Due to time and space constraints, we only present combinatoric data structures, and do not address the higher-level libraries like the ones on finite groups or on matrices. This would deserve another tutorial. We mainly present here how the formalization of *finite* objects benefits from small scale reflection. Yet we hope that this document will help the reader to get started with the library, and to start building further their own formalization on top of it.

2 Instructions for the exercises

This tutorial is intended for an audience already experienced with the notion of formal proof and with a very basic knowledge of the Coq system. For instance we do not recall the basic syntax and principles of the system or its elementary tactics. We still hope that the tutorial can be followed by a novice.

The online reference manual of the Coq system can be found on the Coq website [The10], as well as the html documentation of Coq standard libraries. The reader might also benefit from further reading on the Coq system, like [BC04].

The latest version of the SSReflect language and libraries can be downloaded here:

`http://www.msr-inria.inria.fr/Projects/math-components/`

The distribution contains sources files for the Coq[The10] language extension, the SSReflect libraries, and detailed instructions for the installation of the system. The first part of this tutorial is devoted to a quick guided tour

of the SSReflect language. We do not however include the full documentation of the language, but we still try to remain as self-contained as possible. It is nevertheless good practice to keep the SSReflect manual[GM] at hand while reading the present tutorial, as we will sometimes refer to it for further explanation.

Exercises should be done in a file starting with the following incantation[1]:

```
Require Import ssreflect ssrfun ssrbool eqtype ssrnat div seq.
Require Import path choice fintype tuple finfun finset.

Set Implicit Arguments.
Unset Strict Implicit.
Import Prenex Implicits.
```

which loads the required libraries and sets the implicit arguments options used throughout the libraries and this tutorial. Most of the exercises consist in proving results that are already present in the libraries distributed with SSReflect. When an exercise consists in defining a constant which is already present in the context (in the loaded libraries), the user is asked to re-define it using the same name, prefixed by `tuto_` to avoid name clashes. Specifications to be formally proved by the reader however usually feature the original constants available in the SSReflect libraries. When an exercise consists in proving specifications that are already present in the context, the user is asked to re-prove the specification using the same name, prefixed by `tuto_`. These redundant lemmas, whose proofs are left as exercises, specify the actual SSReflect constants (and not the `tuto_`-prefixed ones), so that the user can benefit from all the additional results already present in SSReflect libraries.

Some exercises are not necessary intended to be *easily* doable by a beginner. They most often comprise several similar questions, and the reader should be able to understand how to solve the last questions after having read the solution of the first one.

Solutions and comments can be found at the following address:

`http://www.msr-inria.inria.fr/Projects/math-components/`

Solutions to the exercises sometimes give useful information for the rest of the section. The reader is advised to read the solution of an exercise before trying the next one.

For advanced users, further documentation is available in each SSReflect library `.v` file header. Each header summarizes the main concepts and notations defined in the library and gives some comments on the use of the objects defined. See for instance the header of `fintype.v`.

We encourage every reader of the present tutorial to subscribe to the SSReflect user mailing list. To subscribe, send an email entitled 'subscribe' to:

`ssreflect@msr-inria.inria.fr`

This mailing list should be used for any further question or comment on the exercises of this tutorial or on any development using the SSReflect extension.

[1] In versions $\leq$ 1.2 of SSReflect libraries, the library `path` was named `paths`. Hence replace the second line with: `Require Import paths choice ...`

3 A script language for structured proofs

A sizable fraction of proof scripts consists of steps that do not "prove" anything new, but instead perform menial bookkeeping tasks such as selecting the names of constants and assumptions or splitting conjuncts. Indeed, SSReflect scripts appear to divide evenly between bookkeeping, formal algebra (rewriting), and actual deduction. Although they are logically trivial, bookkeeping steps are extremely important because they define the structure of the dataflow of a proof script. This is especially true for reflection-based proofs, which often involve large numbers of constants and assumptions. Good bookkeeping consists in always explicitly declaring (i.e., naming) all new constants and assumptions in the script, and systematically pruning irrelevant constants and assumptions in the context. This is essential in the context of an interactive development environment (IDE), because it facilitates navigating the proof, allowing to instantly "jump back" to the point at which a questionable assumption was added, and to find relevant assumptions by browsing the pruned context. While novice or casual Coq users may find the automatic name selection feature of Coq convenient, this feature severely undermines the readability and maintainability of proof scripts, much like automatic variable declaration in programming languages. The SSReflect tactics are therefore designed to support precise bookkeeping and to eliminate name generation heuristics. The bookkeeping features of SSReflect are implemented as tacticals (or pseudo-tacticals), shared across most SSReflect tactics, and thus form the foundation of the SSReflect proof language.

3.1 Sequents as stacks

During the course of a proof Coq always presents the user with a *sequent* whose general form is

```
c_i : T_i
...
d_j := e_j : T_j
...
F_k : P_k
...
==================
forall (x_ℓ : T_ℓ) ...,
let y_m := b_m in ... in
P_n -> ... -> C
```

The *goal* to be proved appears below the double line; above the line is the *context* of the sequent, a set of declarations of *constants* c_i, *defined constants* d_i, and *facts* F_k that can be used to prove the goal (usually, T_i, T_j : `Type` and P_k : `Prop`). The various kinds of declarations can come in any order. The top part of the context consists of declarations produced by the `Section` commands `Variable`, `Let`, and `Hypothesis`. This *section context* is never affected by the SSReflect tactics: they only operate on the upper part — the *proof context*. As in the figure above, the goal often decomposes into a series of (universally) quantified *variables* (x_ℓ : T_ℓ), local *definitions* `let` y_m `:=` b_m `in`, and *assumptions* P_n `->`, and a *conclusion* C (as in the context, variables, definitions, and assumptions can appear in any order). The conclusion is what actually needs

to be proved — the rest of the goal can be seen as a part of the proof context that happens to be "below the line".

However, although they are logically equivalent, there are fundamental differences between constants and facts on the one hand, and variables and assumptions on the others. Constants and facts are *unordered*, but *named* explicitly in the proof text; variables and assumptions are *ordered*, but *unnamed*: the display names of variables may change at any time because of α-conversion.

Similarly, basic deductive steps such as `apply` can only operate on the goal because the Gallina terms that control their action (e.g., the type of the lemma used by `apply`) only provide unnamed bound variables.[2] Since the proof script can only refer directly to the context, it must constantly shift declarations from the goal to the context and conversely in between deductive steps.

In SSReflect these moves are performed by two *tacticals* '`=>`' and '`:`', so that the bookkeeping required by a deductive step can be directly associated to that step, and that tactics in an SSReflect script correspond to actual logical steps in the proof rather than merely shuffle facts. The '`=>`' tactical moves facts and constants from "below the line" to variables and hypotheses "above the line": it performs *introduction*. The '`:`' tactical moves things the other way around: it performs *discharge*.

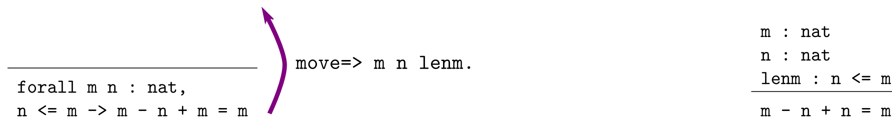

Figure 1: A pure introduction step

Figure 2: A pure discharge step

Still, some isolated bookkeeping is unavoidable, such as naming variables and assumptions at the beginning of a proof. SSReflect provides a specific `move` tactic for this purpose; about one out of every six tactics is a `move`.

Now `move` does essentially nothing: it is mostly a placeholder for '`=>`' and '`:`'. The '`=>`' tactical moves variables, local definitions, and assumptions to the context, while the '`:`' tactical moves facts and constants to the goal. For example, the proof of[3]

```
Lemma subnK : forall m n, n <= m -> m - n + n = m.
```

might start with

```
move=> m n lenm.
```

[2] Thus scripts that depend on bound variable names, e.g., via `intros` or `with`, are inherently fragile.

[3] The name `subnK` means "right cancellation rule for `nat` subtraction", see section 7.

where `move` does nothing, but `=> m n lenm` changes the variables and assumption of the goal in the constants `m n : nat` and the fact `lenm : n <= m`, thus exposing the conclusion `m - n + n = m`, as displayed on figure 1. This is exactly what the specialized Coq tactic `intros m n lenm` would do, but '`=>`' is much more general (see [GM]).

The '`:`' tactical is the converse of '`=>`': it removes facts and constants from the context by turning them into variables and assumptions. Thus as displayed on figure 2, the tactic:

```
move: m lenm.
```

turns back `m` and `lenm` into a variable and an assumption, removing them from the proof context, and changing the goal to

```
forall m, n <= m -> m - n + n = m.
```

which can be proved by induction on `n` using `elim n`.
Because they are tacticals, '`:`' and '`=>`' can be combined, as in

```
move: m lenm => p lenp.
```

simultaneously renames `m` and `lenm` into `p` and `lenp`, respectively, by first turning them into unnamed variables, then turning these variables back into constants and facts.

Furthermore, SSReflect redefines the basic Coq tactics `case`, `elim`, and `apply` so that they can take better advantage of '`:`' and '`=>`'. These Coq tactics require an argument from the context but operate on the goal. Their SSReflect counterparts use the first variable or constant of the goal instead, so they are "purely deductive": they do not use or change the proof context. There is no loss since '`:`' can readily be used to supply the required variable; for instance the proof of the subnK lemma could continue with an induction on `n`:

```
elim: n.
```

This tactic *removes* `n` from the context, following the semantic of the '`:`' discharge tactical. Experience shows that this feature helps controlling the otherwise ever growing size of the context. This default behavior can nonetheless be turned off by the variant:

```
elim: (n).
```

where the generalized object is not a pure identifier any longer. Better yet, this can be combined with previous `move` and with the branching version of the `=>` tactical (described in [GM]), to encapsulate the inductive step in a single command:

```
elim: n m lenm => [m |n IHn m lt_n_m].
```

which breaks down the proof into two subgoals,

```
m - 0 + 0 = m
```

given `m : nat`, and

```
m - S n + S n = m
```

given `m n : nat`, `lt_n_m : S n <= m`, and

```
IHn : forall m, n <= m -> m - n + n = m.
```

An example of this strong recursion principle being generated on the fly is proposed in section 3.5.

The ':' and '=>' tacticals can be explained very simply if one views the goal as a stack of variables and assumptions piled on a conclusion:

- *tactic* : a b c pushes the context constants a, b, c as goal variables *before* performing *tactic*.

- *tactic* => a b c pops the top three goal variables as context constants a, b, c, *after tactic* has been performed.

These pushes and pops do not need to balance out as in the examples above, so

```
move: m lenm => p.
```

would rename `m` into `p`, but leave an extra assumption `n <= p` in the goal.

Basic tactics like `apply` and `elim` can also be used without the ':' tactical: for example we can directly start a proof of subnK by induction on the top variable `m` with

```
elim=> [|m IHm] n le_n_m.
```

3.2 Control flow

3.2.1 Indentation and bullets

The linear development of Coq scripts gives little information on the structure of the proof other than the one provided by the programmer (tabulations, comments...). In addition, replaying a proof after some changes in the statement to be proved will usually not display enough information to distinguish between the various branches of case analysis for instance. To help the user with this organization of the proof script at development time, SSReflect provides some bullets to highlight the structure of branching proofs. The available bullets are `-`, `+` and `*`. Combined with tabulation, this highlights four nested levels of branching (the deepest case we have ever encountered is three). Indeed, the use of "simplifying and closing" switches (see section 3.5), of terminators (see section 3.2.2) and selectors (see section 3.3) is powerful enough to avoid needing more than two levels of indentation most of the time. Note that these indentation levels and bullets have no formal meaning: the fact that an indented script is actually well indented is only guaranteed if each paragraph ends with a closing tactic like the terminators of section 3.2.2.

Here is what a fragment of such a structured script can look like:

```
case E1: (abezoutn _ _) => [[| k1] [| k2]].
- rewrite !muln0 !gexpn0 mulg1 => H1.
  move/eqP: (sym_equal F0); rewrite -H1 orderg1 eqn_mul1.
  by case/andP; move/eqP.
- rewrite muln0 gexpn0 mulg1 => H1.
  have F1: t %| t * S k2.+1 - 1.
    apply: (@dvdn_trans (orderg x)); first by rewrite F0; exact:
        dvdn_mull.
    rewrite orderg_dvd; apply/eqP; apply: (mulgI x).
    rewrite -{1}(gexpn1 x) mulg1 gexpn_add leq_add_sub //.
```

```
    by move: P1; case t.
  rewrite dvdn_subr in F1; last by exact: dvdn_mulr.
  + rewrite H1 F0 -{2}(muln1 (p ^ l)); congr (_ * _).
    by apply/eqP; rewrite -dvdn1.
  + by move: P1; case: (t) => [| [| s1]].
- rewrite muln0 gexpn0 mul1g => H1.
...
```

The reader is not expected to understand what this code proves or does. But this gives a flavor of what a correctly indented script should look like and illustrates a good use of the `by` prefix terminator (see section 3.2.2).

3.2.2 Terminators

Consider the following dummy example, where `.+1` is a notation for the successor operation on natural numbers (i.e. the `S` constructor of the `nat` inductive type):

```
1   Fixpoint f n := if n is n'.+1 then (f n').+2 else 0.
2
3   Lemma foo forall n, f (2 * n) = f n + f n.
4   Proof.
5   elim => [|n ihn].
6     rewrite muln0 //.
7   rewrite !addnS !addSn -/f.
8   rewrite mulnS.
9   rewrite -ihn //.
10  Qed.
```

The proof of `foo` goes by induction, thanks to line 5. Line 6 solves the first goal generated, and the rest of the script solves the second and last goal. Now if we replay the proof after changing value `0` into `1` in the definition of `f`, line 6 does not solve the first case any more. But by accident, line 7 , which is meant to apply only to the second goal, is now a legal operation on the first goal. Hence the script breaks on line 8, which is not relevant to the goal at hand. This phenomenon can be observed on much larger scales, and turns the maintenance of proof scripts into an extremely tedious task.

To further structure scripts, SSReflect supplies *terminating* tacticals to explicitly close off tactics. When replaying scripts, we then have the nice property that an error immediately occurs when a closed tactic fails to prove its subgoal.

It is hence recommended practice that the proof of any subgoal should end with a tactic which *fails if it does not solve the current goal*. Standard Coq already provides some tactics of this kind, like `discriminate`, `contradiction` or `assumption`.

SSReflect provides a generic tactical which turns any tactic into a closing one. Its general syntax is:

`by` ⟨*tactic*⟩.

The expression:

`by` [⟨*tactic*⟩$_1$ | [⟨*tactic*⟩$_2$ | ...].

is equivalent to:

```
[by ⟨tactic⟩1 | by ⟨tactic⟩2 | ...].
```

The latter form makes debugging easier.

In the previous proof of the `foo` lemma, we should replace line 6 by:

```
by rewrite muln0.
```

Hence the script cannot be executed further if what follows the `by` tactical does not close this sub-case.

The `by` tactical is implemented using the user-defined, and extensible `done` tactic. This `done` tactic tries to solve the current goal by some trivial means and fails if it doesn't succeed. Indeed, the tactic expression:

```
by ⟨tactic⟩.
```

is equivalent to:

```
⟨tactic⟩; done.
```

Conversely, the tactic

```
by [ ].
```

is equivalent to:

```
done.
```

The default implementation of the `done` tactic, as an `Ltac` tactic, is to be found in the `ssreflect.v` file. It looks like[4]:

```
Ltac done :=
  trivial; hnf; intros; solve
   [ do ![solve [trivial | apply: sym_equal; trivial]
         | discriminate | contradiction | split]
   | case not_locked_false_eq_true; assumption
   | match goal with H : ~ _ |- _ => solve [case H; trivial] end
     ].
```

Since it is defined using Coq's toplevel tactic language `Ltac`, the `done` tactic can possibly be customized by the user, for instance to include an `auto` tactic. To ensure compatibility with other users' development it is however a better practice to redefine an enriched `my_done` tactic if needed.

Another natural and common way of closing a goal is to apply a lemma which is the `exact` one needed for the goal to be solved. For instance:

```
exact: MyLemma.
```

is equivalent to:

```
by apply: MyLemma.
```

Note that the list of tactics, possibly chained by semi-columns, that follows a `by` keyword is considered as a parenthesized block applied to the current goal. Hence for example if the tactic:

```
 by rewrite my_lemma1.
```

[4] The lemma `not_locked_false_eq_true` is needed to discriminate *locked* Boolean predicates. The `do` tactical is an iterator. See [GM] for more details.

succeeds, then the tactic:

```
by rewrite my_lemma1; apply: my_lemma2.
```

usually fails since it is equivalent to:

```
by (rewrite my_lemma1; apply: my_lemma2).
```

Exercise 3.2.1 Prove the following propositional tautologies:

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

Lemma tauto1 : A -> A.
Proof.
...
Qed.

Lemma tauto2 : (A -> B) -> (B -> C) -> A -> C.
Proof.
...
Qed.

Lemma tauto3 : A /\ B <-> B /\ A.
Proof.
...
Qed.

End Tauto.
```

Your proof script should come in place of the dots, between `Proof.` and `Qed`. Your proof is finished when the system raises a message saying so. Then the `Qed` command rechecks the proof term constructed by your script. In the following, we only give the statements of the lemmas to be proved and do not repeat `Proof` and `Qed` any longer.

The standard Coq section mechanism allows to factorize abstractions globally. Here in the section parameters `A B C` are fixed, and they are discharged after the section `Tauto` is closed by the command `End Tauto`.

Exercise 3.2.2 Prove the following statements:

```
Section MoreBasics.
Variables A B C : Prop.
Variable P : nat -> Prop.

Lemma foo1 : ~(exists x, P x) -> forall x, ~P x.


Lemma foo2 : (exists x, A -> P x) -> (forall x, ~P x) -> ~A.

End MoreBasics.
```

Hint: Remember that the intuitionistic negation `~A` is a notation for `A -> False`. Also remember that the proof of an existential statement is a pair of the witness and its proof, so you can destruct this pair by the `case` command.

Exercise 3.2.3 The SSReflect `ssrnat` library crucially redefines the comparison predicates and operations on natural numbers. In particular, comparisons are Boolean predicates, instead of the inductive versions provided by Coq standard library. Use the `Search` and `Check` commands (see section 7 or [GM]). For instance the command

```
Search _ (_ < _).
```

lists all the available results on the comparison `<` on natural numbers. What is the definition of the SSReflect `leq` predicate, denoted `<=` ? What is the definition of `<` ? Prove the following statements:

```
Lemma tuto_subnn : forall n : nat, n - n = 0.

Lemma tuto_subn_gt0 : forall m n, (0 < n - m) = (m < n).

Lemma tuto_subnKC : forall m n : nat,
  m <= n -> m + (n - m) = n.

Lemma tuto_subn_subA : forall m n p,
  p <= n -> m - (n - p) = m + p - n.
```

3.3 Goal selectors

When composing tactics, the two tacticals `first` and `last` let the user restrict the application of a tactic to only one of the subgoals generated by the previous tactic. This covers the frequent cases where a tactic generates two or more subgoals, one of which can be easily disposed of.

This is an other powerful way to linearize scripts, since it very often happens that a trivial subgoal can be solved by a shorter tactic. For instance, the tactic:

$\langle tactic\rangle_1$ `;` `last by` $\langle tactic\rangle_2$ `.`

tries to solve the last subgoal generated by $\langle tactic\rangle_1$ using the $\langle tactic\rangle_2$, and fails if it does not succeeds. Its analogous

⟨*tactic*⟩$_1$; first by ⟨*tactic*⟩$_2$.

tries to solve the first subgoal generated by ⟨*tactic*⟩$_1$ using the tactic ⟨*tactic*⟩$_2$, and fails if it does not succeeds.

SSReflect also offers an extension of this facility, by supplying tactics to *permute* the subgoals generated by a tactic. The tactic:

⟨*tactic*⟩; last first.

inverts the order of the subgoals generated by ⟨*tactic*⟩. It is equivalent to:

⟨*tactic*⟩; first last.

More generally, the tactic:

⟨*tactic*⟩; last ⟨*strict num*⟩ first.

where ⟨*strict num*⟩ is a natural number argument having value k, rotates the n subgoals $G_1, \ldots, G_n$ generated by ⟨*tactic*⟩ by k positions. The first subgoal becomes G_{n+1-k} and the circular order of subgoals remains unchanged.

Conversely, the tactic:

⟨*tactic*⟩; first ⟨*strict num*⟩ last.

rotates the n subgoals $G_1, \ldots, G_n$ generated by `tactic` in order that the first subgoal becomes G_k.

Finally, the tactics `last` and `first` combine with the branching syntax of `Ltac`: if tactic ⟨*tactic*⟩$_0$ generates n subgoals on a given goal, then tactic

tactic$_0$; last k [*tactic*$_1$|...|*tactic*$_m$] || *tactic*$_{m+1}$.

where k is a natural number, applies *tactic*$_1$ to the $n-k+1$-th goal, ..., *tactic*$_m$ to the $n-k+2-m$-th goal and *tactic*$_{m+1}$ to the others.

For instance, the script:

```
Inductive test : nat -> Prop :=
  C1 : forall n, test n | C2 : forall n, test n |
  C3 : forall n, test n | C4 : forall n, test n.

Goal forall n, test n -> True.
move=> n; case; last 2 [move=> k| move=> l]; idtac.
```

creates a goal with four subgoals, the first and last being `nat -> True`, the second and third being `True` with respectively `k : nat` and `l : nat` in their context.

3.4 Forward chaining, backward chaining

Forward reasoning structures the script by explicitly specifying some assumptions to be added to the proof context. It is closely associated with the declarative style of proof, since an extensive use of these highlighted statements make the script closer to a (very detailed) text book proof.

Forward chaining tactics allow to state an intermediate lemma and start a piece of script dedicated to the proof of this statement. The use of closing tactics (see section 3.2.2) and of indentation makes the portion of the script building the proof of the intermediate statement syntactically explicit.

The `have` tactic.

The main SSReflect forward reasoning tactic is the `have` tactic. It can be used in two modes: one starts a new (sub)proof for an intermediate result in the main proof, and the other provides a proof term for this intermediate step explicitly.

In the first mode, the syntax of `have` in its defective form is:

```
have: ⟨term⟩.
```

This tactic supports open syntax for ⟨*term*⟩: no surrounding parenthesis are needed.

Applied to a goal `G`, it generates a first subgoal requiring a proof of ⟨*term*⟩ is the context of `G`. The difference with the standard Coq tactic is that the second subgoal generated is of the form ⟨*term*⟩ `->` `G`, where ⟨*term*⟩ becomes the new top assumption, instead of being introduced with a fresh name. For instance, consider the following goal:

```
Lemma find_ex_minn : forall P : nat -> bool,
  exists n, P n -> {m | P m & forall n, P n -> n >= m}.
```

The command:

```
have: forall n, P n -> n >= 0.
```

leads to two subgoals, the first remaining to be proved being:

```
forall n, P n -> n >= 0
```

and the second:

```
(forall n, P n -> n >= 0) ->
  exists n, Pn -> {m | P m & forall n, P n -> n >= m}.
```

In this example however, since the lemma is trivial for Coq, the preferred command would be:

```
have: forall n, P n -> n >= 0 by done.
```

where the statement of an easy lemma can be followed by the short proof closing it, like in:

```
have : forall x y, x + y = y + x by move=> x y; rewrite addnC.
```

The `have` tactic can be combined with SSReflect's wildcard mechanism: a placeholder materialized by a '`_`' represents a term whose type is abstracted. For instance, the tactic:

```
have: _ * 0 = 0.
```

is equivalent to:

```
have: forall n : nat, n * 0 = 0.
```

In the same spirit, non-inferred implicit arguments are abstracted. For instance, the tactic:

```
have: forall x y, (x, y) = (x, y + 0).
```

opens a new subgoal to prove that:
`forall (T : Type)(x : T)(y : nat), (x, y)= (x, y + 0)`

An alternative use of the `have` tactic is to provide the explicit proof term for the intermediate lemma, using tactics of the form:

```
have [⟨ident⟩] := ⟨term⟩.
```

This tactic creates a new assumption of type that of ⟨*term*⟩. If the optional ⟨*ident*⟩ is present, this assumption is introduced under the name ⟨*ident*⟩. Note that the body of the constant is lost for the user.

Again, non inferred implicit arguments and explicit holes are abstracted. For instance, the tactic:

```
have H := forall x, (x, x) = (x, x).
```

adds to the context `H : Type -> Prop`. This is a schematic example but the feature is specially useful when for instance the proof term involves a lemma with some hidden implicit arguments.

Variants: the `suff` and `wlog` tactics.

As is often the case in mathematical textbooks, forward reasoning may be used in slightly different variants.

One of these variants is to show that the intermediate step L easily implies the initial goal G. By easily we mean here that the proof of $L \Rightarrow G$ is shorter than the one of L itself. This kind of reasoning step usually starts with: "It suffices to show that ...".

This is such a frequent way of reasoning that SSReflect has a variant of the `have` tactic called `suffices` (whose abridged name is `suff`). The `have` and `suff` tactics are equivalent and have the same syntax but the order of the generated subgoals is swapped.

Another useful construct is reduction, showing that a particular case is in fact general enough to entail a general property. This kind of reasoning step usually starts with: "Without loss of generality, we can assume that ...". Formally, this corresponds to the proof of a goal `G` by introducing a cut *wlog_statement*`-> G`. Hence the user shall provide a proof for both `(`*wlog_statement*`-> G)-> G` and *wlog_statement*`-> G`. This proof pattern is specially useful when a symmetry argument simplifies a proof.

SSReflect implements this kind of reasoning step through the `without loss` tactic, whose short name is `wlog`.

In its defective form:

```
wlog: / ⟨term⟩.
```

on a goal `G`, it creates two subgoals, respectively ⟨*term*⟩`-> G` and `(`⟨*term*⟩`-> G)-> G`. But the `wlog` tactic also offers the possibility to generalize a list of constants on top of the first ⟨*term*⟩`-> G` subgoal. Here is an example showing the beginning of the proof that quotient and reminder of natural number Euclidean division are unique.

```
Lemma quo_rem_unicity: forall d q1 q2 r1 r2 : nat,
  q1 * d + r1 = q2 * d + r2 -> r1 < d -> r2 < d -> (q1, r1) = (
     q2, r2).
```

```
move=> d q1 q2 r1 r2.
wlog: q1 q2 r1 r2 / q1 <= q2.
  by case (le_gt_dec q1 q2)=> H; last symmetry; eauto with arith
      .
```

Here we suppose *without loss of generality* that `q1 <= q2`, and generalize the constants (and possibly facts) `q1 q2 r1 r2`. The first goal generated after the `wlog` tactic is hence:

```
(forall q3 q4 r3 r4 : nat,
   q3 <= q4 ->
   q3 * d + r3 = q4 * d + r4 -> r3 < d ->
   r4 < d -> (q3, r3) = (q4, r4)) ->

  q1 * d + r1 = q2 * d + r2 -> r1 < d -> r2 < d ->
  (q1, r1) = (q2, r2)
```

the second one being:

```
q1 <= q2 ->
q1 * d + r1 = q2 * d + r2 -> r1 < d -> r2 < d ->
(q1, r1) = (q2, r2)
```

3.5 More SSReflect features, on an example

Some important features of the SSReflect language are not documented further in the present tutorial but a detailed description can be found in [GM]. We have mainly omitted:

- the subterm selection mechanism through occurrences and patterns
- the enhanced `rewrite` tactic
- the more complex introduction and discharge patterns

These features will be illustrated by the solutions of the exercises of the next sections. We give here an account of the facilities they provide on a detailed example borrowed from [GS09].

The `div` library defines an Euclidean division algorithm `edivn` and a predicate `edivn_spec` defining its specification.

Exercise 3.5.1 How is `edivn` programmed? What is its specification?

This way of specifying functions and relations is systematically used in SSReflect. It offers a powerful tool for case analysis in proofs thanks to Coq second order unification ability (see exercises 3.5.4 and 4.3.1). Below, let us prove that `edivn` complies with `edivn_spec`.

```
1  Lemma edivnP : forall m d, edivn_spec m d (edivn m d).
2  Proof.
3  rewrite /edivn => m [|d] //=; rewrite -{1}[m]/(0 * d.+1 + m).
4  elim: m {-2}m 0 (leqnn m) => [|n IHn] [|m] q //=; rewrite ltnS
       => le_mn.
5  rewrite subn_if_gt; case: (ltnP m d) => [// | le_dm].
```

```
6   rewrite -{1}(subnK le_dm) -addSn addnA -mulSnr; apply: IHn.
7   apply: leq_trans le_mn; exact: leq_subr.
8   Qed.
```

The proof starts with an *unfolding* of the constant `edivn`, which is performed by the `rewrite /edivn` tactic. This tactic is followed by an introduction step: we introduce `m` and then

```
[|d]
```

performs a case-split on `d`. This case splitting introduction is followed by a so-called simpl-and-closing *switch*:

```
//=
```

This switch, which applies to both branches, can be placed arbitrarily among introduction items, after the introduction arrow `=>`. Its role here is to close the first subgoal generated by the case-split. This `//=` switch is actually a combination of `//`, which closes all the subgoals that can be trivially closed and

```
/=
```

which simplifies all subgoals (generated by `[|d]` case split in this specific example), like the `simpl` tactic 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)
```

This tactic should be read as "replace the first (`{1}`) occurrence of the pattern `m` (`[m]`) by the term `(0 * d.+1 + m)`". The pattern is here given as a full term `m` but it could also be a term with holes like `[_ * (x + _)]`. This combination of occurrence and pattern selection is also available for the bare `rewrite` tactic. In this case, the tactic succeeds because `0 * d.+1 + m` is convertible 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 now 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 would like to invoke a property resembling the generic strong induction principle:

```
forall P : nat -> Prop, P 0 ->
(forall n, (forall m, m <= n -> P m) -> 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). In SSReflect this issue is addressed by the ease to define on the fly non structural ad-hoc induction schemes. This is

performed by combining the `elim` tactic with the generalization patterns and the occurrence selection.

In our case, in the second line of the proof,

```
(leqnn m)
```

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

```
0
```

generalizes `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
```

generalizes every occurrence of `m` in the goal except for 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 induction step generates two subgoals. Thanks to

```
[|n IHn]
```

we introduce an upper bound and the corresponding induction hypothesis in the second subgoal. Note that these brackets have a different effect from the ones used in the first line of the proof. There is no ambiguity: brackets after a case split or an induction are used for parallel introduction while brackets after other tactics which, like `move` or `rewrite`, do not generate new subgoals are casing brackets.

Exercise 3.5.2 What is the effect of each element in `[|m] q //=` ?

Finally,

```
rewrite ltnS
```

rewrites a strict inequality `<` into a non-strict inequality `<=` thanks to the lemma `ltnS`. At the end of 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, see exercise 3.2.3) we rewrite the `match -with` syntax into the Coq `if-then-else` syntax. The next proof step is

```
case: (ltnP m d)
```

which again performs a case split between `(m < d)` and `(d <= m)`.

Exercise 3.5.3 What is the effect of `[// | le_dm]`? Can you find an alternative, flatter introduction pattern having the same effect?

Exercise 3.5.4 In the previous script, replace `case: (ltnP m d)` by `case: ltnP`. What happens? What is the statement of `ltnP`? What is the definition of `ltn_xor_geq`? On the model of `ltn_xor_geq`, define an inductive specification `tuto_compare_nat` which performs a three-case split according to the order of two natural numbers. Prove that:

```
Lemma tuto_ltngtP : forall m n,
  compare_nat m n (m < n) (n < m) (m == n).
```

At the end of 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)
```

The tactic

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

rewrites, from right to left, only the first occurrence (because of the `{1}` specification) of the first pattern that matches the equality of lemma `subnK le_dm`.

Then

```
rewrite -addSn
```

finds 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).

Exercise 3.5.5 What does the rest of the script do?

4 Small scale reflection, first examples

4.1 The two sides of deduction

In the Calculus of Inductive Constructions [CH88, PM93], there is an obvious distinction between logical propositions and Boolean values. On the one hand, logical propositions are objects of *sort* `Prop` which is the carrier of intuitionistic reasoning. Logical connectives in `Prop` are *types*, which give precise information on the structure of their proofs; this information is automatically exploited by Coq tactics. For example, Coq knows that a proof of `A \/ B` is either a proof of `A` or a proof of `B`. The tactics `left` and `right` change the goal `A \/ B` to `A` and `B`, respectively; dually, the tactic `case` reduces the goal `A \/ B => G` to two subgoals `A => G` and `B => G`.

On the other hand, `bool` is an inductive *datatype* with two constructors `true` and `false`. Logical connectives on `bool` are *computable functions*, defined by their truth tables, using case analysis:

```
Definition (b1 || b2) := if b1 then true else b2.
```

Properties of such Boolean connectives are established using case analysis: the tactic `by case: b` solves the goal

```
b || ~~ b = true
```

where `~~` denotes the boolean negation, by replacing `b` first by `true` and then by `false`; in either case, the resulting subgoal reduces by computation to the trivial `true = true`.

Moreover, Booleans can be injected into propositions using the coercion mechanism:

```
Coercion is_true (b : bool) := b = true.
```

This allows any boolean formula `b` to be used in a context where Coq would expect a proposition. It is then interpreted as `(is_true b)`, i.e., the proposition `b = true`. Coercions are elided by the pretty-printer, so they are essentially transparent to the user. These coercions were in fact already present in the statement of the last two lemmas of exercise 3.2.3.

Thus, `Prop` and `bool` are truly complementary: the former supports robust natural deduction, the latter allows brute-force evaluation. SSReflect supplies a generic mechanism to have the best of the two worlds and move freely from a propositional version of a decidable predicate to its boolean version. As a result one of the motos of the small scale reflection methodology is: "if a predicate is decidable, it should be defined through a boolean predicate, possibly accompanied with logical specifications".

The bookkeeping facilities presented in section 3 are crafted to ease simultaneous introduction/generalization of facts and casing, naming ... operations. The SSReflect extension also provides a way to ease the combination of a stack operation immediately followed by an *interpretation* of the fact being pushed, that is to say to apply a lemma to this fact before passing it to a tactic for decomposition, application and so on. This proves specially useful when interpreting boolean predicates into logical ones.

4.1.1 Interpreting assumptions

Interpreting an assumption in the context of a proof consists in applying a correspondence lemma to this assumption before generalizing, and/or decomposing it. Such a correspondence lemma is called a *view lemma*. For instance, with the extensive use of boolean reflection, it is quite frequent to need to decompose the logical interpretation of (the boolean expression of) a fact, rather than the fact itself. This can be achieved by a combination of `move : _ => _` switches, as in the following script, where `||` is a standard Coq notation for the boolean disjunction:

```
Variables P Q : bool -> Prop.
Hypothesis P2Q : forall a b, P (a || b) -> Q a.

Goal forall a, P (a || a) -> True.
move=> a HPa; move: {HPa}(P2Q _ _ HPa) => HQa.
```

which transforms the hypothesis `HPa : P a` which has been introduced from the initial statement into `HQa : Q a`. In this example, the view lemma is `P2Q`. This operation is so common that the tactic shell has specific syntax for it. The following scripts:

```
Goal forall a, P (a || a) -> True.
move=> a HPa; move/P2Q: HPa => HQa.
```

or more directly:

```
Goal forall a, P (a || a) -> True.
move=> a; move/P2Q=> HQa.
```

are equivalent to the former one. The former script shows how to interpret a fact (already in the context), thanks to the discharge tactical ':' and the latter, how to interpret the top assumption of a goal. Note that the number of wildcards to be inserted in order to find the correct application of the view lemma to the hypothesis has this time been automatically inferred.

The view mechanism is compatible with the `case` tactic:

```
Variables P Q: bool -> Prop.
Hypothesis Q2P : forall a b, Q (a || b) -> P a \/ P b.

Goal forall a b, Q (a || b) -> True.
move=> a b; case/Q2P=> [HPa | HPb].
```

creates two new subgoals whose contexts do not contain `HQ : Q (a || b)` any more, but respectively `HPa : P a` and `HPb : P b`. This view tactic performs:

```
move=> a b HQ; case: {HQ}(Q2P _ _ HQ) => [HPa | HPb].
```

The term on the right of the / view switch is the *view lemma*. Any term coercing to a product type can be used as a view lemma.

The examples we have seen so far explicitly provided the direction of the translation to be performed. In fact, view lemmas need not be oriented. The view mechanism is able to detect which application is relevant for the current goal. For instance, the script:

```
Variables P Q: bool -> Prop.
Hypothesis PQequiv : forall a b, P (a || b) <-> Q a.

Goal forall a b, P (a || b) -> True.
move=> a b; move/PQequiv=> HQab.
```

has the same behavior as the first example above.

The view mechanism can automatically insert a *view hint* to transform the double implication into the expected simple implication. The last script is in fact equivalent to:

```
Goal forall a b, P (a || b) -> True.
move=> a b; move/(iffLR (PQequiv _ _)).
```

where:

```
Lemma iffLR : forall P Q, (P <-> Q) -> P -> Q.
```

4.1.2 Specializing assumptions

The special case when the *head symbol* of the view lemma is a wildcard is used to interpret an assumption by *specializing* it. The view mechanism hence offers the possibility to apply a higher-order assumption to some given arguments.

For example, the script:

```
Goal forall z, (forall x y, x + y = z -> z = x) -> z = 0.
move=> z; move/(_ 0 z).
```

changes the goal into:

```
(0 + z = z -> z = 0) -> z = 0
```

4.1.3 Interpreting goals

In a similar way, it is often convenient to interpret a goal by changing it into an equivalent proposition. The view mechanism of SSReflect has a special syntax `apply/` for combining simultaneous goal interpretation operations and bookkeeping steps in a single tactic.

With the hypotheses of section 4.1.1, the following script, where `~~` denotes the boolean negation:

```
Goal forall a, P ((~~ a) || a).
move=> a; apply/PQequiv.
```

transforms the goal into `Q (~~ a)`, and is equivalent to:

```
Goal forall a, P ((~~ a) || a).
move=> a; apply: (iffRL (PQequiv _ _)).
```

where `iffLR` is the analogous of `iffRL` for the converse implication.

Any SSReflect term whose type coerces to a double implication can be used as a view for goal interpretation.

Note that the goal interpretation view mechanism supports both `apply` and `exact` tactics. As expected, a goal interpretation view command `exact`/*term* should solve the current goal or it fails.

4.1.4 The `reflect` predicate

In practice, double implication is not the most efficient way to relate booleans and logical interpretations. The following inductive predicate `reflect` indeed proves far more powerful:

```
Inductive reflect (P: Prop): bool -> Type :=
  | Reflect_true: P => reflect P true
  | Reflect_false: ~P => reflect P false.
```

The statement `(reflect P b)` asserts that `(is_true b)` and `P` are logically equivalent propositions.

For instance, the following lemma:

```
Lemma andP: forall b1 b2 : bool, reflect (b1 /\ b2) (b1 && b2).
```

relates the boolean conjunction `&&` to the logical one `/\`. Note that in `andP`, `b1` and `b2` are two boolean variables and the proposition `b1 /\ b2` hides two coercions. The conjunction of `b1` and `b2` can then be viewed as `b1 /\ b2` or as `b1 && b2`.

Expressing logical equivalences through this family of inductive types makes possible to take advantage from *rewritable equations* associated to case analysis of Coq's inductive types.

Since the standard equivalence predicate is defined in Coq as:

```
Definition iff (A B : Prop) := (A -> B) /\ (B -> A).
```

where `/\` is the standard notation for logical `and`:

```
Inductive and (A B : Prop) : Prop :=
  conj : A -> B -> and A B
```

This makes case analysis very different according to the way an equivalence property has been defined.

For instance, if we have proved the lemma:

```
Lemma andE: forall b1 b2, (b1 /\ b2) <-> (b1 && b2).
```

let us compare the respective behaviors of `andE` and `andP` on a goal:

```
Goal forall b1 b2, if (b1 && b2) then b1 else ~~(b1||b2).
```

Expressing a reflection relation through the `reflect` predicate is hence a very convenient way to deal with classical reasoning, by case analysis. Using the `reflect` predicate moreover allows programming rich specifications inside its two constructors, which will be automatically taken into account during destruction (see for instance exercises of section 4.3) . This formalization style gives far more efficient specifications than quantified (double) implications.

A naming convention in SSReflect is to postfix the name of view lemmas with `P`: `orP` relates `||` and `\/`, `negP` relates `~~` and `~`, etc.

Exercise 4.1.1 State the lemma tuto_orP. Prove lemmas tuto_andP and tuto_orP.

The view mechanism is compatible with `reflect` predicates. For example, the script

```
Goal forall a b : bool, a -> b -> a /\ b.
move=> a b Ha Hb; apply/andP.
```

changes the goal `a /\ b` to `a && b` (see section 4.1.3).

Conversely, the script

```
Goal forall a b : bool, a /\ b -> a.
move=> a b; move/andP.
```

changes the goal `a /\ b -> a` into `a && b -> a` (see section 4.1.1).

The same tactics can also be used to perform the converse operation, changing a boolean conjunction into a logical one. The view mechanism guesses the direction of the transformation to be used i.e., the constructor of the `reflect` predicate which should be chosen.

4.1.5 Interpreting equivalences

Equivalent boolean propositions are simply *equal* boolean terms. A special construction helps the user to prove boolean equalities by considering them as logical double implications (between their coerced versions), while performing at the same time logical operations on both sides.

The syntax of double views is:

$$\texttt{apply/}\langle term\rangle_l\texttt{/}\langle term\rangle_r\,.$$

The term $\langle term \rangle_l$ is the view lemma applied to the left hand side of the equality, $\langle term \rangle_r$ is the one applied to the right hand side.

In this context, the identity view:

```
Lemma idP : reflect b1 b1.
```

is useful, for example the tactic:

```
apply/idP/idP.
```

transforms the goal `~~ (b1 || b2)= b3` into two subgoals, respectively `~~ (b1 || b2)-> b3` and `b3 -> ~~ (b1 || b2).`

The same goal can be decomposed in several ways, and the user may choose the most convenient interpretation. For instance, the tactic:

```
apply/norP/idP.
```

applied on the same goal `~~ (b1 || b2)= b3` generates the subgoals `~~ b1 /\ ~~ b2 -> b3` and `b3 -> ~~ b1 /\ ~~ b2`.

4.1.6 Proving `reflect` equivalences

Section 4.1.4 advocates the use of the reflect predicate to express logical equivalence between boolean predicates and their logical interpretation. To prove such `reflect` equivalences, we had so far no other choice than a naive case analysis on the value of the boolean. This is the proof method used for proving *elementary* `reflect` statements like andP. However, when it comes to proving more complex, *composed* statements, this remains a valid strategy, but a very inefficient one. In fact, it is never used in practice for higher level equivalences. The user of course expects to be able to prove such an equivalence by the usual double implication. The swiss army knife to establish `reflect` equivalences is in fact the following transitivity result:

```
Lemma iffP : forall (P Q : Prop) (b : bool),
      reflect P b -> (P -> Q) -> (Q -> P) -> reflect Q b.
```

Exercise 4.1.2 Prove the lemma tuto_iffP by case analysis on the boolean value `b`. Retry the proof, this time by case analysis on the hypothesis `(reflect P b)`.

This lemma changes a `reflect` equivalence goal `(reflect Q b)` for a new one `(reflect P b)` provided that the `Prop` statements `P` and `Q` are equivalent. Note that the converse is trivial: changing an equivalence goal `(reflect Q b)` into `(reflect Q b')` with `b` equivalent to `b'` is simply *rewriting* `b` into `b'`. Now recall the trivial idP lemma proved in section 4.1.5. Forall `(P Q : Prop)` and `(b : bool)`, the term `(iffP P Q b (idP b))` has type `((b -> Q)-> (Q -> b)-> reflect Q b)`. Hence on a goal of the form:

```
Goal reflect P b.
```

the tactic:

```
apply: (iffP idP).
```

generates the two subgoals `(b -> P)` and `(P -> b)`, realizing the expected double implication case split.

The `iffP` lemma of course accepts any `reflect` statement as an argument. On a goal of the form:

```
Goal reflect (P1 /\ P2) (b1 && b2).
```

the tactic:

```
apply: (iffP andP).
```

generates the two subgoals `(b1 /\ b2 -> P1 /\ P2)` and `(P1 /\ P2 -> b1 && b2)`.

4.2 Exercises: sequences

For technical reasons, the SSReflect library defines a clone of the standard Coq list type:

```
Inductive seq (T : Type) : Type := Nil | Cons of T & seq T.
```

Note that in this definition, we use the anonymous argument feature of the SSReflect language (see [GM]). The program computing the size of such a sequence can be written as:

```
Variable (T : Type).
Fixpoint size (s : seq T) :=
  if s is _ :: s' then (size s').+1 else 0.
```

taking benefit of the conditional pattern feature of the SSReflect language (see [GM]).

Exercise 4.2.1 Program the function tuto_cat catenating two sequences.

Here we take advantage of the Coq system `Implicit Types` feature (see [The10]). This allows to bind variable names to a given type. Hence the code of this exercise starts with the declarations:

```
Section Exo_4_2_1.w

Variable A : Type.
Implicit Types s : seq A.
Implicit Types x : A.
```

which opens a section for the code of this exercise, declares a local parameter `A` and sets the type of bound variables *starting with* `s` (resp. `x`) to be of type `( seq A)` (resp. `A`). Unless the bound variable is already declared with an explicit type in which case, this latter type is considered.

The actual cat function of the `seq` library is equipped with the `++` infix notation. Prove the lemma:

```
Lemma tuto_size_cat : forall s1 s2,
    size (s1 ++ s2) = size s1 + size s2.
```

Note that variable `s1` and `s2` are automatically declared with type `(seq A)`, without any explicit cast, thanks to the previous `Implicit Types` declaration.

Program the function tuto_last, such that
`(tuto_last x s)` returns the last element of the sequence `s` if `s` is not empty and otherwise returns `x`. Prove the lemma:

```
Lemma tuto_last_cat : forall x s1 s2,
  last x (s1 ++ s2) = last (last x s1) s2.
```

Program the functions tuto_take (resp. tuto_drop), of type:

```
nat -> seq A -> seq A
```

such that `(tuto_take n s)` (resp. `(tuto_drop n s)`) computes the prefix of s of size `n` (resp. the postfix of s skipping the `n` first elements), with default value `s` (resp. the empty sequence `[::]`). Prove:

```
Lemma tuto_cat_take_drop : forall (n0 : nat)(s : seq A),
  take n0 s ++ drop n0 s = s.
```

Program the tuto_rot function such that `(tuto_rot n s)` is the circular permutation of `s` of order `n`. Prove that:

```
Lemma tuto_rot_addn : forall m n (s : seq A),
m + n <= size s -> rot (m + n) s = rot m (rot n s).

End Exo_4_2_1.
```

For this last proof, you will need more lemmas about the function programmed in this exercise. Use the SSReflect `Search` command to find the statements you need. You can also try to guess their name according to the SSReflect naming conventions, and `Check` your guesses (see section 7).

Combining sequences with boolean predicates makes possible to start proving some combinatoric results.

Exercise 4.2.2 After declaring:

```
Section Exo_4_3_1.

Variable T : eqType.
Implicit Types x y : T.
Implicit Type b : bool.
```

program a function tuto_count which computes the number of elements of a sequence satisfying a boolean predicate.

Prove that:

```
Lemma tuto_count_predUI : forall a1 a2 s,
  count (predU a1 a2) s + count (predI a1 a2) s
 = count a1 s + count a2 s.
```

where `predU` is the boolean predicate union of its two arguments, and `predI` is the boolean predicate intersection of its two arguments.

Hint: try to use the `nat_congr` tactic, an Ltac tactic defined in the `ssrnat` library, to normalize arithmetic expressions and perform congruence.

Look for the definition of the `filter` function. Prove that:

```
Lemma count_filter : forall a s, count a s = size (filter a s).
```

and close the section with:

```
End Exo_4_3_1.
```

Combining sequences with boolean relations makes it possible to formalize decidable paths:

```
Fixpoint path (T : Type)(e : rel T) x (p : seq T) {struct p} :=
  if p is y :: p' then e x y && path e y p' else true.
```

where `rel T` is a binary boolean relation on `T`. Now let us state our first non trivial reflection lemma:

```
Lemma pathP : forall (T : Type)(e : rel T)(x : T)(p : seq T) x0,
  reflect
    (forall i, i < size p -> e (nth x0 (x :: p) i) (nth x0 p i))
    (path e x p).
```

Exercise 4.2.3 Prove the lemma tuto_pathP by induction on the path.

4.3 Exercises: Boolean equalities

The structures of types with boolean equality is the core of the hierarchy of structures defined by the SSReflect libraries. In the standard Coq `DecidableType` library, a type whose Leibniz equality is decidable is hence specified by:

```
Parameter eq_dec (T : Type) : forall x y : T,
    {x = y} + {~ (x = y)}.
```

A proof of `(eq_dec x y)` is either a proof of `(x = y)`, hence belong to the left hand side of the sum, or a proof of `~(x = y)`, in the right hand side. A sum

type being an inductive type with two constructor (one for each side), one can perform case analysis on a proof of `(eq_dec x y)`, hence a case analysis on the equality or dis-equality of the two elements `x` and `y`. In each branch, a proof of the assertion valid in this branch is available as it should be an argument of the corresponding constructor.

Small scale reflection favors the use of *boolean* predicates instead of such `sum_bool` types. Indeed, unlike sum types, boolean predicates have the computational behavior expected to let reduction handle deductive steps that only rely on truth table values. The SSReflect account of `DecidableType` is named `eqType`, and its theory is developed in the `eqtype` library. The `eqType` structure can be thought of[5] as:

```
Module Equality.

Definition axiom T e := forall x y : T, reflect (x = y) (e x y).

Record mixin_of (T : Type) := Mixin {
  op : rel T;
  _ : axiom op
}.

Record type := Pack {
  sort :> Type;
  _ : mixin_of sort
End Equality.
```

This can be thought of as a kind of sigma type packing a type with a signature and specifications. The signature + specification part is called a *mixin*. The actual boolean comparison of an `eqType` structure can be accessed through the defined `eq_op` operator:

```
  eq_op : forall T : eqType, rel T
```

which enjoys some infix notations: `(eq_op x y)` is denoted by `(x == y)`, and `(~~ (x == y))` by `(x != y)`. This operator is defined by destructing `T`, and its mixin. Moreover, we also define the `eqP` constant:

```
  eqP : forall T : eqType, Equality.axiom eq_op
```

which accesses the reflection lemma associated to the boolean comparison.

[5]For technical reasons, the structure is slightly different. More insight about the actual formalizations is given in [GGMR09]

Exercise 4.3.1 Prove the following lemmas:

```
Lemma tuto_eqxx : forall (T : eqType) (x : T), x == x.

Lemma tuto_predU1l : forall (T : eqType) (x y : T) (b : bool),
  x = y -> (x == y) || b.

Lemma tuto_predD1P : forall (T : eqType) (x y : T) (b : bool),
  reflect (x <> y /\ b) ((x != y) && b).

Lemma tuto_eqVneq : forall (T : eqType) (x y : T), {x = y} + {x
   != y}.
```

Hint: Consider using view mechanisms for equivalence, goal and assumption interpretation.
Remark : try starting the proof of eqVneq by the tactic:

```
move=> T x y; case: eqP.
```

What happens then?

Remark: An alternative to the case analysis on `(eqVneq x y)` is simply a simple case analysis on the boolean value of `(x == y)`. But one can also perform case analysis on `(eqP x y)`: one branch features `Reflect_true eq_xy` where `(eq_xy : x = y)` and the other branch `Reflect_false neq_xy` where `(neq_xy : ~(x = y))`.

Besides giving a computational content to Leibniz equality, the boolean relation embedded in an `eqType` structure also gives its decidability of course. An important consequence of this decidability is the uniqueness of their equality proofs:

```
Theorem eq_irrelevance : forall (T : eqType) (x y : T),
 forall (e1 e2 : x = y), e1 = e2.
```

The uniqueness of equality proofs for the `nat` and `bool` types is a consequence of this theorem. boolean proof irrelevance is of particular interest for the definition of sigma types with boolean specifications (see section 6.2).

5 Type inference using canonical structures

5.1 Canonical Structures

The type-theoretic formalization of an algebraic or combinatorial structure comprises representation types (usually only one), constants and operations on the type(s), and axioms satisfied by the operations. Within the propositions-as-types framework of Coq, the interface for all of these components can be uniformly described by a collection of dependent types: the type of operations depends on the representation type, and the statement (also a "type") of axioms depends on both the representation type and the actual operations. In the examples and exercises we have encountered so far, types, operations, and axioms have been represented by collections of unbundled parameters, using the `Variables`, `Parameters`, and `Hypothesis` commands.

While this unbundling allows for maximal flexibility, it also induces a proliferation of arguments that is rapidly overwhelming. A typical algebraic structure, such as a ring, involves half a dozen constants and even more axioms. Moreover such structures are often nested, e.g., for the Cayley-Hamilton theorem (see the `charpoly` library, out of the scope of this tutorial) one needs to consider the ring of polynomials over the ring of matrices over a general commutative ring. The size of the terms involved grows as C^n, where C is the average number of separate components of a structure, and n is the structure nesting depth. For Cayley-Hamilton we would have $C = 15$ and $n = 3$, and thus terms large enough to make theorem proving impractical, given that algorithms in user-level tactics are more often than not nonlinear.

Thus, at the very least, related operations and axioms should be packed using Coq's dependent records (Σ-types). Here is a toy example for a commutative group:

```
Record zmodule_mixin_of (T : Type) : Type := ZmoduleMixin {
  zero : T;
  opp : T -> T;
  add : T -> T -> T;
  addA : associative add;
  addC: commutative add;
  addm0 : left_id zero add;
  add0m : left_inverse zero opp add
}.

Record zmodule : Type := Zmodule {
  carrier :> Type;
  spec : zmodule_mixin_of carrier
}.
```

Again, the `zmodule` structure can be thought of as a kind of sigma type packing carrier type with a signature `zmodule_mixin`. For instance, Booleans can be equipped with such a structure (with a xor as addition and identity as opposite):

```
Definition bool_zmoduleMixin := ZmoduleMixin addbA addbC addFb
    addbb.
Definition bool_zmodule := Zmodule bool_zmoduleMixin.
```

Note that the four first arguments of `bool_zmodule_mixin` should be respectively `bool`, `false`, `(@id bool)` and `addb` (see the role of the `@` flag in annex 7). In fact, they have been automatically inferred from the type of the other arguments. The `:>` symbol after the `carrier` field indicates that the `carrier : zmodule -> Type` projection is in fact declared on the fly as a *coercion*. Remember coercions provide an explicit subtyping mechanism to the Coq system. They are silently inserted during type inference. For instance the following declaration is valid:

```
Variable b : bool_zmodule.
```

is accepted by the system, even if `bool_zmodule` is not a type. The command

```
Check b.
```

answers `b : bool_zmodule`, as expected. Yet if the global option of coercion display is set by the vernacular command:

```
Set Printing Coercions.
```

the answer of `Check` becomes `b : carrier bool_zmodule`. Once this structure defined, it is possible to define handy notations and develop a theory for instances of the structure. For instance, let us define a notation for the addition operation of a `zmodule`. We first need a definition to access the operation through the nested records.

```
Definition zmadd (Z : zmodule) := add (spec Z).
```

Then we define an infix notation:

```
Notation "x \+ y" := (@zmadd _ x y)(at level 50,left
    associativity).
```

where _ is a placeholder for an inhabitant of `zmodule` to be inserted by the type inference mechanism. Now we can conveniently state and prove the following result:

```
Lemma zmaddAC : forall (m : zmodule)(x y z : m),
x \+ y \+ z = x \+ z \+ y.
```

Exercise 5.1.1 Prove that zmadd is associative and commutative. Prove lemma zmaddAC. Refer to [GM] for the documentation of the SSReflect `rewrite` tactic (in particular pattern selection).

Abstract algebraic structures are motivated by the factorization of notations and theorems, which are supposed to be *shared* by every instance of a given structure. In our toy example, we hence expect addition over Booleans to *inherit* from the infix \+ notation, and from the lemma `zmaddAC`. But the following command:

```
Check false \+ true.
```

fails with the following error message:

```
Error: The term "false" has type "bool" while it is expected to
    have
  type "carrier ?15".
```

Indeed, the expression `false + true` is a notation for `@zmadd _ true false`, where _ is a placeholder for an argument of type `zmodule`. Type inference should hence unify the type `bool` of arguments `true` and `false` with `(carrier ?)` where `?` has type zmodule. There is no way for unification to guess now the `?` hole can be filled with `bool_zmodule`.

However Coq supports a way to provide hints to the unification algorithm called `Canonical Structures` [Sai97]. It can be viewed as an instance of unification hints as presented much more recently in [ARCT09]. This mechanism equips the system with a type class mechanism [WB89]. In the context of a proof assistant this feature notably enables proof inference by type inference. In the Coq system, an other type classes mechanism à la [WB89] has been implemented [SO08], independently from the canonical structures mechanism, but on top of a framework for dependent type programming. It is this latter mechanism which is actually usually referred to as 'Coq type classes'. In their current state, the SSReflect libraries only make use of the canonical structures mechanism.

Going back to our previous example, we can provide the unification algorithm with a hint to guess that if a `zmodule` structure is required on `bool`, then our intention is that it *has to be* `bool_zmodule`:

```
Canonical Structure bool_zmodule.
```

After this declaration, the `Check (false + true)` command does not raise an error message any more but answers: `false + true : bool_zmodule`. The canonical structure declaration indeed stores some equations in a database known to the unification algorithm. These equations will guide the algorithm in case some holes remain in a unification problem. One equation is stored per named field in the structure. In our example, declaring `bool_zmodule` as a canonical instance stores the hint:

$$[\ \texttt{carrier}\ ?\ \backsimeq \texttt{bool}\ ]\ \Rightarrow\ ?=\texttt{bool_zmodule}$$

plus an additional hint for the `spec` projection, which will reveal useless [6]. The error message raised at our first attempt to type `false + true` complained that `false` has type `bool` and was expected to have type `(carrier ?)`. Now after the canonical structure declaration, the first hint in the list gives a solution to this problem.

Canonical structures not only enable the sharing of notations, but also that of *proofs*: on the goal

```
Goal forall x y z : bool, x (+) y (+) z = x (+) z (+) y.
```

where `(+)` is a notation for the concrete xor operation `addb`, the command:

```
apply: zmaddAC.
```

solves the goal[7]. To avoid spurious folding and unfolding of definitions, it is a recommended practice to use generic notations as often as possible on concrete instance. Hence the previous goal would best be expressed as:

```
Goal forall x y z : bool, x \+ y \+ z = x \+ z \+ y.
```

This simple example reflects the structure of more intricate modular switches:

- Definition of generic abstract structures like `zmodule`. This can involve more subtle curryfication and dependent types to achieve full modularity, inheritance and sharing (see for instance [GGMR09]).
- Development of a generic theory for each structure, consisting of lemmas like `zmaddAC`.
- Population of the generic structures. This consists in the definition of instances of the structures, like `bool_zmodule`. These instances are most often declared canonical.
- Development of specific theories of the instances. These libraries benefit from the generic results (and notations) established at the abstract level thanks to the canonical structure hints.

[6] This is why throughout the SSReflect library projections corresponding to specifications are usually not named to avoid polluting the database with hints which will never be used.

[7] For technical reasons, when working with Coq $< v8.3$, make sure to use the SSReflect `rewrite` and `apply:` tactics to trigger canonical structure inference.

5.2 Canonical constructions

An important feature of canonical structures is that unification can chain several steps of type inference, triggering a Prolog-like engine for proof inference. Let us start with an elementary example, based on types with Boolean equality. The record presented in section 4.3 is in fact the *mixin* of the `eqType` structure.

To declare an elementary (canonical) structure of `eqType` on a type `T`, one must follow the following scheme:

1. Define a Boolean comparison on the elements of type `T`;
2. Prove that this equality is a Boolean reflection of Leibniz equality;
3. Build the mixin packing `T` with the latter proof;
4. Build the `eqType` structure on `T`.

Let us define a canonical structure of `eqType` on the type `unit`. The Boolean comparison in that case is the function `(fun _ _ : unit => true)`. The three next steps respectively consist in:

```
Lemma unit_eqP : Equality.axiom (fun _ _ : unit => true).
Proof. by do 2!case; left. Qed.

Definition unit_eqMixin := EqMixin unit_eqP.

Canonical Structure unit_eqType := EqType unit unit_eqMixin.
```

Exercise 5.2.1 How would you define a canonical structure of `eqType` on type `bool`? on type `nat`?

What is in each case the equation given as a hint to the unification algorithm?

Now dependent types may inherit some structure from their parameters, when they are themselves equipped with some structure. For instance, there is a canonical way of building a Boolean comparison of pairs of elements themselves comparable by Boolean predicates. We say that the product of two `eqType`s has a canonical structure of `eqType`. Indeed the Boolean test:

```
Definition pair_eq (T1 T2 : eqType) :=
  [rel u v : T1 * T2 | (u.1 == v.1) && (u.2 == v.2)].
```

is the expected Boolean comparison. It is defined using the notation for casted boolean relations: the notation `[rel x y : T | t]` denotes the term `(fun x y : T => t)`, of type `T -> T -> bool`[8].

Exercise 5.2.2 Prove the lemma:

```
Lemma tuto_pair_eqP : forall T1 T2, Equality.axiom (pair_eq T1
  T2).
```

Now we pose the following definitions:

[8] Many variants of this notation are defined in the `ssrbool` library: with or without cast on the arguments, ...

```
Definition prod_eqMixin (T1 T2 : eqType) :=
  EqMixin (@pair_eqP T1 T2).
Canonical Structure prod_eqType (T1 T2 : eqType) :=
  EqType (T1 * T2) prod_eqMixin.
```

This canonical structure definition stores the following equation in the database:

$$[\,\texttt{Equality.sort}\ ?_1 \backsimeq \texttt{prod}\ (\texttt{Equality.sort}\ ?_2)\ (\texttt{Equality.sort}\ ?_3)] \\ \Rightarrow \\ ?_1 = \texttt{prod_eqType}\ ?_2\ ?_3$$

where `prod` is the constant hidden by the infix `*` notation of type product. We give here an example where this hint is used, and triggers further canonical structure inference. Remember from section 4.3 that the Boolean comparison operation of an `eqType` is named `eq_op`, and supports the infix notation `==`. After having solved exercise 5.2.1, try the following command:

```
Check (true, 3) == (true && true, 1 + 2).
```

If the canonical structures of `eqType` have been correctly defined on `bool` and `nat` (they are in fact introduced respectively in libraries `ssrbool` and `ssrnat`), then the system should answer `bool`.

To type this expression, the system has to unfold the notation, hence to type the term:

```
eq_op _ (true, 3) (true && true, 1 + 2)
```

where `eq_op` has type:

```
eq_op : forall T : eqType, rel (Equality.sort T)
```

as shown by the command `Check eq_op` in `Set Printing Coercions` mode. Since the two last arguments of the term to be typed are of type `(bool * nat)`. The system should hence unify:

`(Equality.sort ?)` with `(prod bool nat)`

There is no way of solving this unification problem without extra information coming from canonical structures. Canonical structure inference is triggered by head constants: in this unification problem, the respective head symbols `Equality.sort` and `prod` of both sides match the head symbols of product of `eqType` hint. This hint says that we can obtain `?` as `(prod_eqType ?2 ?3)` if we can solve the two new problems: unify

`bool` with `(Equality.sort ?2)`
and
`nat` with `(Equality.sort ?3)`

And this should very much look like the solution of exercise 5.2.1.

To craft canonical constructions, always remember that their inference is triggered by head constants and projections. Unfortunately, the vernacular support for canonical structures is rather elementary, in fact limited to the:

```
Print Canonical Projections.
```

command, which lists the hints present in the database.

5.3 Predtypes: canonical structures for notations

An important case where canonical structures implement a shared notation is the infix notation for membership. In SSREFLECT libraries, if `P` is a boolean predicate, the statement "x satisfies `P`" can be written applicatively as `(P x)` or using an explicit infix connective, as `(x \in P)`. In the latter case, `P` is called a "collective" predicate and supports the notations:

- `(x \in P)` for "x satisfies the collective predicate `P`"
- `(x \notin P)` for "x does not satisfy the collective predicate `P`"

A collective predicate is typically a membership predicate for lists, finite types or more generally, containers. A given predtype `T` is expected to support (at most) a single membership (collective) predicate, giving an unambiguous meaning to the Boolean expression `(x \in A)`, with `(A : T)`. When there is not natural way of seeing a given type as a container, it is not relevant to equip it with a predtype structure. To equip a type `T` with the two above prenex notations, the user should declare a canonical structure of `predType` on `T`. We are not describing in detail the technical aspect of the the `predType` structure definition here. It is sufficient to understand that, just like an `eqType` structure bundles a type `T` with a Boolean relation on `T`, which is required to reflect the Leibniz equality on `T`, a `(predType T)` structures equips an other type with a canonical Boolean membership predicate. For instance, the predicate:

```
mem_seq : forall T : eqType, seq T -> T -> bool
```

tests the membership of an element of type `(T : eqType)` (see section 4.3) in any sequence (see section 4.2) of elements of `T`. Any type `(seq T)`, with `(T : eqType)` is canonically equipped with a `predType` structure using this membership definition. Hence we can write:

```
Section SeqMem.

Variable T : eqType.

Implicit Type s : seq T.
Implicit Types x y : T.

Lemma tuto_in_cons : forall y s x,
  (x \in y :: s) = (x == y) || (x \in s).
Proof. by []. Qed.
```

where the `Implicit Types` declarations avoid further otherwise necessary casts.

Exercise 5.3.1 In the same section `SeqMem`, prove the following lemmas:

```
Lemma tuto_in_nil : forall x, (x \in [::]) = false.

Lemma tuto_mem_seq1 : forall x y, (x \in [:: y]) = (x == y).

Lemma tuto_mem_head : forall x s, x \in x :: s.
```

Note that while the bare definition of membership does not require the sequence to be based on a type with Boolean equality, the entire SSREFLECT sequence library is geared towards reflection, in the sense of the three lemmas proved in exercise 5.3.1.

It is sometimes desirable to change an infix notation `(x \in P)` into `(P x)`. For any collective predicate, this can be done using the generic rewrite multirule `inE`. A mulirule is a COQ constant defined as a list of rewrite lemmas. The tactic:

```
rewrite inE.
```

looks in the list `inE` for the first rewrite rule which applies to the current goal and hence changes the first occurrence of a pattern `(x \in P)` into `(P x)`[9]. For more details on mulirules, please refer to [GM].

Exercise 5.3.2 Prove the following lemmas:

```
Lemma tuto_mem_cat : forall x s1 s2,
  (x \in s1 ++ s2) = (x \in s1) || (x \in s2).

Lemma tuto_mem_behead: forall s, {subset behead s <= s}.
```

where the last statement stands for:

```
forall s x, x \in behead s -> x \in s
```

Program by induction a Boolean test `tuto_has: pred T -> seq T -> bool` which tests whether a sequence features an element satisfying a given Boolean predicate. Prove the following reflection lemma:

```
Lemma tuto_hasP : forall (a : pred T) s,
   reflect (exists2 x, x \in s & a x) (has a s).
```

where the standard COQ constructor `exists2` specifies a witness for a *conjunction* of predicates. Program by induction a Boolean test `tuto_all: pred T -> seq T -> bool` which tests whether all the elements of a sequence satisfy a given Boolean predicate. Prove the following reflection lemmas:

```
Lemma tuto_allP : forall (a : pred T) s,
  reflect (forall x, x \in s -> a x) (all a s).

Lemma tuto_allPn : forall (a : pred T) s,
  reflect (exists2 x, x \in s & ~~ a x) (~~ all a s).
End SeqMem.
```

[9] In fact, the generic `inE` multirule should usually be extended each time a new membership predicate is defined. In the sequence library `inE` is for instance redefined to include lemmas `in_cons` and `mem_seq1`.

6 Finite objects in SSReflect

6.1 Finite types

6.1.1 Finite types constructions

Sequences are used to define types with a finite number of inhabitants. A `finType` is not built out of distinct constructors but instead it consists of a sequence enumerating its elements. This proves to be more efficient for combinatoric operations. A type `(T : finType)` hence embeds a type with boolean equality and a duplicate-free sequence containing all the elements of the carrier type. Let us equip the `bool` type with a (canonical) structure of `finType`[10]:

```
Lemma bool_enumP : Finite.axiom [:: true; false].
Proof. by case. Qed.
Definition bool_finMixin := FinMixin bool_enumP.
Canonical Structure bool_finType := FinType bool bool_finMixin.
```

where the sequence `[:: true; false]` enumerates the inhabitants of the type (here all the elements of the underlying `eqType`), and `Finite.axiom` is the specification:

```
Finite.axiom (T : eqType)(e : seq e) :=
      forall x, count (@pred1 T x) e = 1.
```

which ensures that the sequence contains exactly one occurrence of each element of the underlying `eqType`. The sequence enumerating the elements of `(T : finType)` is `(enum T)`. By construction it satisfies the `Finite.axiom` specification:

```
Lemma enumP : forall T : finType, Finite.axiom (Finite.enum T).
```

Exercise 6.1.1 Declare a canonical structure of `finType` on the `unit` type.

Now, canonical constructions can transmit a structure of finite type to a dependent type whose parameters are themselves finite types. For instance, an option type on a finite type is itself a finite type. The construction of the (canonical) structure goes this way:

1. Construct the enumeration of the inhabitants of the finite type:

   ```
   Definition option_enum (T : finType) :=
     None :: map some (Finite.enum T).
   ```

2. Prove that it satisfies the finite type specification:

   ```
   Lemma option_enumP : forall T : finType,
     Finite.axiom (option_enum T).
   ```

3. Construct the finite type mixin:

   ```
   Definition option_finMixin (T : finType) :=
     FinMixin option_enumP.
   ```

[10] In versions $\leq$ 1.2 of the SSReflect libraries, the last line should be `Canonical Structure bool_finType:= FinType bool_finMixin`

4. Define the corresponding `finType` structure and declare it canonical[11]:

```
Canonical Structure option_finType :=
  FinType (option T) option_finMixin.
```

Exercise 6.1.2 Prove lemma `tuto_option_enumP`.

It might be more convenient to build a `finType` by proving separately that the enumeration is duplicate-free and that it contains all the elements of the underlying `eqType`. The `fintype` library hence provides an alternative `mixin` called `UniqFinMixin` for this purpose.

Exercise 6.1.3 Define the function:

```
Definition tuto_sum_enum (T1 T2 : finType) : seq (T1 + T2) :=
```

where `(T1 + T2)` is the sum operation on types (see section 4.3), which enumerates *all* the elements of `(T1 + T2)`. Prove that it returns a duplicate-free sequence by proving:

```
Lemma tuto_sum_enum_uniq : forall T1 T2, uniq (sum_enum T1 T2)
    .
```

Then the following definitions declare a canonical construction of `finType` on the sum of two arbitrary `finType`s.

```
Definition sum_finMixin :=
  UniqFinMixin sum_enum_uniq mem_sum_enum.
Canonical Structure sum_finType :=
  FinType (T1 + T2) sum_finMixin.
```

Exercise 6.1.4 Using `UniqFinMixin`, build a canonical construction of `finType` on the product of two arbitrary `finType`s.

6.1.2 Cardinality, set operations

The cardinal operator applies to any boolean predicate on a `finType`: if `T : finType` and `A : T -> bool`, then `#|A|` counts the number of elements of `T` which are assigned a `true` value by `A`. Moreover, `#|T|` denotes the number of elements of the whole `finType`. The `enum` operator[12] builds a duplicate-free list of all the elements of `T` satisfying `A`. Hence we have the key property:

```
Lemma cardE : forall (T : finType)(A : pred T),
    #|A| = size (enum A).
```

Moreover, two extensionally equal boolean predicates (on the same `finType`) have the same enumeration:

```
Lemma eq_enum : forall P Q, P =i Q -> enum P = enum Q.
```

[11]In versions ≤ 1.2 of the SSReflect libraries, the last line should be
`Canonical Structure option_finType:= FinType option_finMixin`

[12]This operator should not be confused with the above `Finite.enum` field, which is a projection of the `finType` structure.

Seeing the boolean predicates of `finType` as characteristic functions, we can state and prove the corresponding cardinality lemmas:

```
Section OpsTheory.

Variable T : finType.

Implicit Types A B C P Q : pred T.
Implicit Types x y : T.
Implicit Type s : seq T.

Lemma card0 : #|@pred0 T| = 0.
Proof. by rewrite cardE enum0. Qed.

Lemma cardT : #|T| = size (enum T).
Proof. by rewrite cardE. Qed.

Lemma card1 : forall x, #|pred1 x| = 1.
Proof. by move=> x; rewrite cardE enum1. Qed.
```

Exercise 6.1.5 The boolean predicate (on boolean predicates over finite types):

```
Definition pred0b (T : finType) (P : pred T) := #|P| == 0.
```

characterizes an empty characteristic function. In the `OpsTheory` section prove the lemma:

```
Lemma tuto_pred0P : forall P, reflect (P =1 pred0) (pred0b P).
```

Exercise 6.1.6 Again in section `OpsTheory`, prove that:

```
Lemma tuto_cardUI : forall A B,
#|[predU A & B]| + #|[predI A & B]| = #|A| + #|B|.

Lemma tuto_eq_card : forall A B, A =i B -> #|A| = #|B|.
```

where the bracket notations pretty-print set operations for collective predicates. Hint: use the lemmas proved in the exercises of section 4.2.

The expression `[disjoint A & B]` is a boolean which is true if and only if the collectives boolean predicates `A, B : pred T` where `T` is a `finType` are disjoint: the intersection of `A` and `B` should satisfy `pred0`.

Exercise 6.1.7 Again in section `OpsTheory`, prove the lemmas:

```
Lemma tuto_disjoint0 : forall A, [disjoint pred0 & A].
Lemma tuto_disjoint_sym : forall A B, [disjoint A & B] = [
    disjoint B & A].

Lemma tuto_disjointU : forall A B C,
[disjoint predU A B & C] = [disjoint A & C] && [disjoint B & C
    ].

End OpsTheory.
```

Hint: try to use the `congr` tactic (see [GM]).

For any two boolean predicates `pA` and `pB` ranging over the same type `T`, we can say that `pA` is a *subset* of `pB` if it selects elements of `T` that are also selected by `pB` which is not a decidable test in the general test. Yet if `A` and `B` are two boolean predicates on the same *finite* domain, this notion indeed becomes a boolean test since there is only finitely many values to inspect. The boolean predicate `A \subset B`, for arguments `A, B : pred T`, holds if and only if `(pred0b [predD A B])`, i.e. if and only if $A \backslash B$ is empty.

Exercise 6.1.8 Prove the reflection lemmas:

```
Lemma tuto_subsetP : forall A B,
reflect {subset A <= B} (A \subset B).

Lemma tuto_subsetPn : forall A B,
reflect (exists2 x, x \in A & x \notin B) (~~ (A \subset B)).
```

Hint: both these proofs should start by the tactic:

```
rewrite unlock.
```

to release the seals protecting unwanted reductions. This is imposed by the definition of `subset`. Technical details about these seals can be found in [GM], yet the reader can safely skip this point.

Prove the lemmas:

```
Lemma tuto_subset_eqP : forall A B,
reflect (A =i B) ((A \subset B) && (B \subset A)).

Lemma tuto_subset_cardP : forall A B,
#|A| = #|B| -> reflect (A =i B) (A \subset B).
```

6.1.3 boolean quantifiers

On finite types, logical quantifiers can be reflected into boolean ones: indeed, a universal statement amounts to a finite conjunction of tests, and an existential one amounts to a boolean test on a finite number of values. Then the boolean existential connective `(existsb x, A x)` is defined by stating that `A` is not empty, in the sense of the `pred0` predicate. The boolean universal connective `(forallb x, A b)` is defined by stating that the complement of A is empty. These boolean quantifiers satisfy the rules of classical logic (since the domain of

quantification is finite) and are provably equivalent to their logical constructive counterparts.

Exercise 6.1.9 State and prove the reflection lemmas tuto_existsP and tuto_forallP relating the Prop quantifiers with their boolean versions.

Prove the lemmas:

```
Lemma tuto_negb_forall : forall (T : finType)(P : pred T),
  ~~ (forallb x, P x) = (existsb x, ~~ P x).

Lemma tuto_negb_exists : forall (T : finType)(P : pred T),
  ~~ (existsb x, P x) = (forallb x, ~~ P x).
```

Exercise 6.1.10 Prove that on any non-empty subset described by a predicate `(P : pred T)`, where `(T : finType)`, a function `F : T -> nat` has a maximum and a minimum.

Hint: the `fintype` library defines a `pick` choice operator, which is legal on a type with a finite number of inhabitants. Hence `[pick x | P]` is `Some x`, for an `x` such that `P` holds, or `None` if there is no such `x`. This operator is specified by a `pickP` specification lemma. Also, the `ssrnat` library defines the minimum `ex_minn` (resp. maximum `ex_maxn`) of the values satisfying a non-empty (reps. bounded non empty) predicate `p : pred nat`. Use the Search command to investigate the theory developed on these operations.

6.1.4 Example: a depth first search algorithm

In this section, we illustrate the formalization of an algorithm, its specification and the formal proof of its correctness on the case of a depth first search algorithm in a graph[13]. This commented proof also illustrates the feature of the SSReflect language, and in particular of the view mechanism, on an example of formalization by boolean reflection.

We consider a graph given by the `finType` of its vertices, and its neighbor function:

```
Variables (T : finType) (e : T -> seq T).
```

In this graph, there is an edge between two vertices `x` and `y` if and only if `y` is in the image of `x` by the neighbor function `e`. Hence the adjacency relation is defined by:

```
Definition grel := [rel x y | y \in e x].
```

using the bracket notation for boolean relations. The *depth first search algorithm* computes all the vertices of a graph which are reachable from a given initial vertex by a path in the graph. It proceeds by visiting recursively all the neighbors of the initial vertex, then the neighbors of these neighbors, etc. Since the underlying graph may feature cycles, it is necessary to mark the vertices visited by the algorithm to avoid extraneous recursive calls and infinite loops. Consider the function:

[13]This formalization can be found in the SSReflect connect library.

```
Fixpoint dfs (n : nat) (a : seq T) (x : T) {struct n} :=
  if n is n'.+1 then
    if x \in a then a else foldl (dfs n') (x :: a) (e x)
  else a.
```

This function performs `n` steps of the depth first search, starting from the vertex `x` with some already visited vertices stored in the sequence `a`. Our goal is to prove the following specification:

```
Lemma dfs_pathP : forall x y,
  reflect
  (exists2 p, path grel x p & y = last x p)
  (y \in dfs #|T| [::] x).
```

This specification ensures that a vertex `y` is reachable form the vertex `x` by a path in the graph if and only if it is found in `#|T|` (the number of vertices in the graph) steps by the function `dfs`, starting from vertex `x` with an empty set of marked vertices.

The core of this proof is the invariant of the `dfs` function:

```
Lemma dfsP : forall n x y (a : seq T),
#|T| <= #|a| + n ->
y \notin a -> reflect (dfs_path x y a) (y \in dfs n a x).
```

where the `dfs_path` predicate is defined as:

```
Inductive dfs_path x y (a : seq T) : Prop :=
  DfsPath p of path grel x p & y = last x p & [disjoint x :: p &
    a].
```

meaning that the predicate `(dfs_path x y a)` holds if and only if there exists `(p : seq T)` such that:

- two successive elements of the sequence `x :: p` are related by the `grel` relation (they are adjacent in the graph),
- `y` is the last element of `p`,
- the sequence `x :: p` does not contain any element of the sequence `a`.

In other words `(dfs_path x y a)` holds if there is a path from `x` to `y` in the graph which avoids the marked vertices of `a`. Now let us start the proof of theorem `dfsP`, by induction on the natural number `n`:

```
Proof.
elim=> [|n IHn] x y a Hn Hy /=.
```

We are now ready to prove the base case of the induction.

Exercise 6.1.11 Why is the context of this subgoal inconsistent?

Indeed, the theorem:

```
Lemma max_card : forall (T : finType)(A : pred T),
    #|A| <= #|T|.
```

is violated by this context. We would hence like to replace the current goal by the absurd boolean statement that could be derived under such assumption. A common and convenient way of performing this step of boolean contradiction is the following: Suppose that you know that a boolean statement `B` is provable, to replace the current goal by `~~B`, just use the tactic:

```
case/idPn: PB
```

where PB is a proof of `(is_true B)`. In our proof, this tactic is:

```
case/idPn: (max_card (predU1 y (mem a))).
```

Note that `(mem a)` is the standard way to transform a sequence into a predicate, which is the characteristic function of the set of elements in the sequence.

Exercise 6.1.12 Observe the effect of the previous tactic on the goal. Use the vernacular command:

```
Show Proof.
```

to display the current state of the proof term. Look for the occurrence of `idPn`. Which function is this occurrence an argument of? How has this function been inserted (see [GM], section 8.2)? Try to decompose the last tactic into more elementary steps, without using the automatic insertion of view hints.

The subgoal is then closed by the following tactic:

```
by rewrite -ltnNge cardU1 (negPf Hy) addSn addnC.
```

Note the `by` closing tactic which ensures this subgoal is killed by this script.

Exercise 6.1.13 Use the command:

```
Check cardU1.
```

What is the type of each subterm? Why is this statement well-typed? Hint: Look at the result of the command:

```
Set Printing Coercions.
Check cardU1.
Unset Printing Coercions.
```

Exercise 6.1.14 What is the type of `negPf`? What is the type of `(negPf Hy)`? Why is this last statement well typed (same hint as exercise 6.1.13)?

For the inductive case, the proof goes by case analysis on `x` being an element of `a`:

```
case Hx: (x \in a).
```

Exercise 6.1.15 Prove the first case where `Hx : (x \in a)= true`. Hint: Don't forget to use the `Search` vernacular tactic (see [GM] for the syntax). For instance:

```
Search _ [disjoint _ & _].
```

lists all the available results on `disjoint`.

We now start a step of forward reasoning, proving an auxiliary result which will be used several times in the rest of the proof:

```
have subset_dfs : forall m (u v : seq T),
   u \subset foldl (dfs m) u v.
```

This command starts a new subgoal, with the same context as the main proof we just left, but requiring a proof of the lemma.

Exercise 6.1.16 Prove this lemma (by double induction on `n` and `b`, generalizing with respect to `a`). Again, use `Search` to find the lemmas needed. For instance:

```
Search (_ \in _ :: _).
```

shows all the theorems concluding that an element is in a non empty list.

Back to the main proof, the lemma has now been added to the context, under the name `subset_dfs`. Let us give a name to the sequence `x :: a`. To introduce an *abbreviation*, we can use the tactic:

```
pose a' := x :: a.
```

In the present case, it will be more convenient to introduce this new name under the form of a new constant and an *equality*:

```
move Da': (x :: a) => a'.
```

Now we reason by case analysis on the fact the `(y \in a')`:

```
case Hya': (y \in a').
```

This tactic introduces in each subgoal a hypothesis `Hya'`, giving the respective values `true` and `false` to the boolean `y \in a'`. In the first subgoal, we know that:

```
(y \in foldl (dfs n) a' (e x)) = true
```

because of hypothesis `Hya'` and of the lemma `subset_dfs`. The boolean `y \in foldl (dfs n)a' (e x)` can be *rewritten* to `true` using the tactic:

```
rewrite (subsetP (subset_dfs n _ _) _ Hya').
```

Note that the same coercion is applied as in exercise 6.1.14.

Exercise 6.1.17 Finish the proof of this first case.

Now remains the case of `Hya': y \in a' = false`. Hypothesis `Hn` says that `#|T| <= #|a| + n.+1` but here we know more:

```
have Hna': #|T| <= #|a'| + n by rewrite -Da' /= cardU1 Hx /=
    add1n addSnnS.
```

Since this proof is so short that both the statement of the lemma and its proof use less than 80 characters, we can use this open syntax, without separating the statement and the proof script by a point.

Exercise 6.1.18 Introduce a new object `b` and an equality hypothesis `Db : e x = b`, like we did above to introduce the sequence `a'`.

Now, we will again reason forward using the command:

```
suffices IHb: reflect (exists2 x', x' \in b & dfs_path x' y a')
                      (y \in foldl (dfs n) a' b).
```

This time, the proof of the intermediate result is postponed to the second subgoal, and in the first subgoal, the context has been augmented with hypothesis `IHb`.

Exercise 6.1.19 Prove this subgoal. Hint: first transform it into two implications using `IHb` (see section 4.1.6).

Here is a script which finishes the proof:

```
elim: b a' Hya' Hna' {a x Da' Db Hy Hn Hx} => [|x b IHb] a Hy Hn
    /=.
  by rewrite Hy; right; case.
have Ha := subset_dfs n a [ :: x ]; simpl in Ha.
case Hdfs_y: (y \in dfs n a x).
  rewrite (subsetP (subset_dfs n _ b) _ Hdfs_y); left.
  exists x; [ exact: mem_head | apply: (IHn _); auto; exact (
      negbT Hy) ].
have Hca := subset_leq_card Ha; rewrite -(leq_add2r n) in Hca.
apply: {IHb Hca}(iffP (IHb _ Hdfs_y (leq_trans Hn Hca))).
  move=> [x' Hx' [p Hp Ep Hpa]]; rewrite disjoint_sym in Hpa.
  exists x'; [ exact: predU1r | exists p => // ].
  rewrite disjoint_sym; exact (disjoint_trans Ha Hpa).
move=> [x' Hx' [p Hp Ep Hpa]].
case Hpa': [disjoint x' :: p & dfs n a x].
  case/orP: Hx' => [Dx'|Hx']; last by exists x'; auto; exists p.
  move: (pred0P Hpa x'); rewrite /= mem_head /= => Hax'.
  case/idP: (pred0P Hpa' x'); rewrite /= mem_head //=.
  apply/(IHn _ _ _ Hn (negbT Hax')).
  exists (Nil T)=> //; first by move/eqP: Dx'.
  by rewrite disjoint_has /= -(eqP Dx') Hax'.
case/(IHn _ _ _ Hn (negbT Hy)): Hdfs_y.
case/pred0Pn: Hpa' => [x'' H]; case/andP: H => [ /= Hpx'' Hdfs_x
    '')].
have Hax'' := pred0P Hpa x''; rewrite /= Hpx'' in Hax''.
case/(IHn _ _ _ Hn (negbT Hax'')): Hdfs_x'' => [q Hq Eq Hqa].
case/splitPl: {p}Hpx'' Hp Ep Hpa => [p1 p2 Ep1].
rewrite path_cat -cat_cons disjoint_cat last_cat Ep1.
move/andP=> [Hp1 Hp2] Ep2; case/andP=> [Hp1a Hp2a]; exists (cat q
     p2).
- by rewrite path_cat Hq -Eq.
- by rewrite last_cat -Eq.
by rewrite -cat_cons disjoint_cat Hqa.
Qed.
```

Exercise 6.1.20 List the places where a reflection lemma is used. Where is it used as a function? Where has a `Hint View` been inserted? Can you comment the script with the steps of the informal proof?

Exercise 6.1.21 Using `dfsP`, prove the specification:

```
Lemma dfs_pathP : forall x y,
  reflect
  (exists2 p, path grel x p & y = last x p)
  (y \in dfs #|T| [::] x).
```

6.2 Sigma types with decidable specifications

Sigma types as defined in the standard Coq prelude `Init.Specif` (automatically loaded by Coq), are a convenient way to define new types in comprehension style. They support a built-in curly bracket notation, so that:

```
Definition evens := {x : nat | exists k, 2 * k = x}.
```

is the type whose inhabitants are even natural numbers. They are implemented as a pair whose first projection is usually a datatype, called the `value`, and the second one a proof that the first element satisfy the definitional predicate. Sigma types do not behave as conveniently as desired: let us prove that 2 can be seen as an element of `evens` by two different ways. The first one is straightforward and the second one make an unnecessary detour:

```
Definition two_even1 : evens.
Proof. by exists 2; exists 1. Defined.

Definition two_even2 : evens.
Proof. by exists 2; rewrite -(addn0 2) addn0; exists 1. Defined.

Goal two_even1 = two_even2.
reflexivity.
Abort.
```

The error message is due to the fact that not all the proofs of a given theorem are equal. Try `Print two_even1` and `Print two_even2` to compare these two terms. Since we are comparing pairs of elements whose second components are (non convertible) proofs, there is no way these two elements are convertible. In general, there is not even any reason why they should be provably equal. However the situation is much different when the sigma type is defined by means of a boolean predicate. Consider the definition:

```
Definition odds := {x : nat | odd x}.
```

where `odd : nat -> bool` is defined in the `ssrnat` library. Now we prove that 1 can be seen as an element of `odds` by two different ways:

```
Definition one_odd1 : odds.
Proof. by exists1. Defined.

Definition one_odd2 : odds.
Proof. by exists 1; rewrite -(addn0 1) addn0. Defined.

Goal one_odd1 = one_odd2.
try reflexivity. (* still not convertible *)
by congr exist; apply: bool_irrelevance.
```

```
Qed.
```

A sigma type with boolean specifications still does not allow to *convert* elements sharing the same first projection. Hence the `reflexivity` tactic, which checks the convertibility of the two sides, fails. Yet proof irrelevance holds for type `bool`: for any `(b : bool)`, all the proofs of `(b = b)` are the same. Since the second projection of an element of type `odds` should a proof of (something convertible to) `(true = true)`, two elements of type `odd` sharing the same value are provably equal. Hence the `bool_irrelevance` lemma applies, reducing the goal to proving the trivial equality `(2 = 2)`.

To take advantage of this notable property, the SSReflect `eqType` library provides an special interface for `subType`s. A `subType` is defined on top of a sigma type by typically:

- Defining a sigma like type, under the form of an ad hoc `Record` type of the form:

  ```
  Record myType (T : Type) := {myval :> T ; _ : P (myval)
      }
  ```

  where`(P : T -> bool)` is a concrete predicate. This definition generates an elimination scheme `myTypeT`.

- Use this scheme to declare a canonical structure of `subType` of `T` for `mysubType`:

  ```
  Canonical Structure myTypeSubType :=
  [subType for myval by myType_rect].
  ```

  The above notation hides the generic construction patterns which automates the definition of a `subType` instance.

Exercise 6.2.1 Define the `Record` type `tuto_tuple_of: nat -> Type -> Type` such that `(tuto_tuple_of n T)` if the type of sequences on type `T` of fixed length `n`. Remember that we want a boolean specification, and that the type `nat` has a canonical structure of `eqType`. Craft this definition so that it also declares a coercion from `tuto_tuple_of` to `seq` (see example in section 5.1).

Now define a canonical structure of `subType` on the type `tuple_of`.

Prove that `[::]` can be equipped with a canonical structure of `tuple`. Prove that `cons` is an operation which builds `tuple` from a `tuple` (and a head element).

Prove that:

```
Lemma tuto_cat_tupleP : forall T n1 n2 (t1 : n1.-tuple T) (t2
    : n2.-tuple T),
size (t1 ++ t2) == n1 + n2.
```

where `(n.-tuple T)` is a notation for tuples of elements in `t` of length `n`.

Define a canonical construction of `tuple` for the catenation of two `tuple`s.

Prove that similarly the sequence operations, `drop`, `take`, `rot` (see exercise 4.2.1) canonically preserve the `tuple` structure of of their arguments.

Canonical instances of the `subType` structure benefit from generic operators and lemmas such as the injection of an element of the sigma type `myType` into the bigger type `T`:

```
val : forall (T : Type) (P : pred T) (s : subType P), s -> T
```

and the proof `val_inj` that `val` is *injective*.We also know that the value of a sigma type satisfies its specification:

```
valP : forall (T : Type) (P : pred T) (sT : subType P) (u : sT)
    ,
     P (val u)
```

Exercise 6.2.2 Prove the lemmas:

```
Section TuplesExercises.
Variables (T : finType)(n : nat).

Lemma tuto_size_tuple : forall (t : n.-tuple T), size t = n.

Lemma leq_card_tuple : forall (t : n.-tuple T), #|t| <= n.

Lemma uniq_card_tuple : forall (t : n.-tuple T),
   uniq t -> #|t| = n.
```

where again `(n.-tuple T)` is a notation for `(tuple_of n T)` defined in the SSReflect `tuple` library.

Here is an excerpt of the `tuple` library:

```
Lemma tnth_default : forall (t : n.-tuple T)(i : 'I_n), T.
Proof. by case=> [[|//]]; move/eqP <-; case. Qed.

Definition tnth t i := nth (tnth_default t i) t i.
```

What type does the `'I_n` notation stands for? Hint: use the `Search` command. How could you define this type as a `subType`? What does the `tnth` function computes? Prove that:

```
Lemma tuto_tnth_nth : forall (x : T)(t : n.-tuple T) i, tnth t
    i = nth x t i.
```

(which answers the previous question...)

Moreover the operator `Sub` is a generic constructor for elements of a subtype: `(Sub x Px)` where `(x : T)` and `Px` is a proof of `(P x)` constructs the corresponding elements of the subtype of the elements of `T` satisfying property `P`. Note that the predicate `P` is guessed automatically by the construction if the return type is known.

Exercise 6.2.3 Define the element 2 of type `(ordinal 3)`.

Define the type `odds` of odd integers, define the corresponding `subType` `odds_subType` of `nat`. Define the element of `3 : odds_subType`.

If a type `T` is equipped with a boolean equality, this equality (or more precisely its restriction) is also a valid one for any sigma type defined on `T`. An important role of the `subType` structure is to convey in a systematic way a structure of `eqType`[14] present on the larger type `T` to any sigma type defined

[14] and of `choiceType`

on `T`. There is in fact a *systematic construction* of `eqType` for subtypes defined on top of an `eqType`. Following the above notations, if `myType` has a canonical structure of subtype on a type `T` *equipped with a canonical structure of* `eqType`, then a (canonical) structure of `eqType` can be declared for `myType` by:

- Defining an `eqMixin` by the generic construction triggered by the following notation:

  ```
  Definition myType_eqMixin := [eqMixin of myType by <:].
  ```

  This notation requires a previous canonical structure of `subType` for `myType`.

- Defining the (canonical) `eqType` structure[15]:

  ```
  Canonical Structure mySubType_eqType := EqType myType
      myType_eqMixin.
  ```

Exercise 6.2.4 Define a canonical structure of `eqType` on `odds` (cf. exercise 6.2.3). Define a canonical structure of `eqType` on `tuple`.

Prove the lemma:

```
Lemma tuto_map_tnth_enum : forall (t : n.-tuple T),
map (tnth t) (enum 'I_n) = t.
```

and its extensionality corollary:

```
Lemma tuto_eq_from_tnth : forall (t1 t2 : n.-tuple T),
tnth t1 =1 tnth t2 -> t1 = t2.

End TuplesExercises.
```

6.3 Finite functions, finite sets

An important application of the `tuple` construction, studied in the exercises of the previous section, is the formalization of functions on finite domains. A function `(f : aT -> rT)`, where `aT` is a type with a finite number of inhabitants and `rT` an arbitrary type, is completely determined by a finite object: the list of values respectively assigned to the finite sequence of elements of the domain. Yet defining such a function as a bare inhabitant of the arrow type `(aT -> rT)` is not enough to benefit from this finiteness. For instance, the Calculus of Inductive Constructions being essentially non extensional, we cannot use the fact that:

```
forall f1 f2 : aT -> rT, (forall x : aT, f1 x = f2 x) -> f1 =
    f2
```

even since in the present case, testing that `(forall x : aT, f1 x = f2 x)` only requires a finite number of tests. This can prove specially uncomfortable for instance in combinatoric proofs or in quotient constructions. The SSReflect `finfun` library implements a definition of functions with a `fintype` domain and an arbitrary codomain, as tuples of values: `(f : finfun aT rT)` where

[15] In versions ≤ 1.2 of the SSReflect libraries, the last line should be `Canonical Structure mySubType_eqType := EqType myType_eqMixin.`

(aT : finType) is a #|aT|.-tuple of values in rT. The elements of aT being given as an (ordered) enumerating sequence, the finfun lists their respective values in the same order. Now these functions, coerced to their functional types, enjoy the equivalence between intensional and extensional equality:

```
Lemma ffunP : forall (aT : finType) rT (f1 f2 : {ffun aT rT},
  f1 =1 f2 <-> f1 = f2.
```

An important special case of finite functions is the boolean one: they are characteristic functions of subsets of the base finType. Otherwise said, they define a mask on the finType domain. This case is important enough to deserve the definition of a special subtype:

```
Inductive set_type (T : finType) := FinSet of {ffun pred T}.
```

denoted by {set T} where T is *required* to hold a canonical structure of finType. In particular, as for finite functions, (Leibniz) intensional and extensional equalities coincide for such sets:

```
Lemma setP : forall (T : fintype)(A B : {set T}), A =i B <-> A =
    B.
```

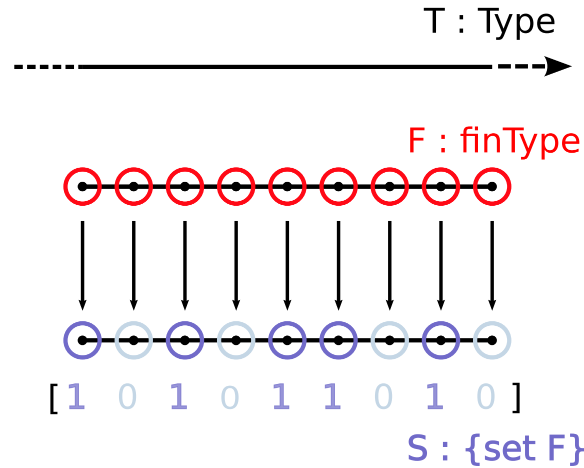

Figure 3: Finite sets as masks on finite types

Most usual set constructions are defined on these sets: if A, B : {set T} and P : {set {set T}}:

- x \in A denotes that x belongs to A
- A \subset B denotes that A is a subset of B
- A \proper B denotes that A is a proper subset of B
- mem A is the boolean predicate corresponding to A
- finset p is the set corresponding to a boolean predicate p
- [set x | C] is the set containing the x such that C holds (x is bound in C)

- `[set x \in D]` is the set containing the `x` in the collective predicate `D`
- `[set x \in D | C]` is the set containing the `x` in `D` such that `C` holds
- `set0` denotes the empty set
- `[set: T]` or `setT` denotes the full set, containing all the elements of the `finType T`
- `A :|: B` is the union of `A` and `B`
- `x |: A` is union of the singleton `x` with the set `A`
- `A :&: B` is the intersection of sets `A` and `B`
- `~: A` is the complement of `A` (in the `finType T`)
- `A :: B` is the difference `A` minus `B`
- `A : x` is the set obtained by removing the element `x` from `A`

Finite sets also inherit from the cardinality theory developed on predicates on `fintypes`, and the library specializes all the results proved on these predicates to the set case.

Exercise 6.3.1 Prove the following lemmas:

```
Section setOpsExos.

Variable T : finType.
Implicit Types a x : T.
Implicit Types A B C D : {set T}.

Lemma tuto_eqEsubset : forall A B,
   (A == B) = (A \subset B) && (B \subset A).

Lemma tuto_set1P : forall x a, reflect (x = a) (x \in [set a]).

Lemma tuto_setD1P : forall x A b,
  reflect (x != b /\ x \in A) (x \in A :\ b).

Lemma tuto_setIA : forall A B C, A :&: (B :&: C) = A :&: B :&: C.

Lemma tuto_setUIl : forall A B C,
   (A :&: B) :|: C = (A :|: C) :&: (B :|: C).

Lemma tuto_setCU : forall A B, ~: (A :|: B) = ~: A :&: ~: B.

End setOpsExos.
```

Hint: use the results proved in exercise 6.1.8, and the `inE` rewrite rule.

Exercise 6.3.2 In this exercise, we prove the existence of a minimal subset satisfying a given property.

```
Section MinSet.

Variable T : finType.
Notation sT := {set T}.
Implicit Types A B C : sT.
Implicit Type P : pred sT.

Definition tuto_minset P A := forallb B : sT, ...
```

Complete the definition tuto_minset to give a boolean characterization of the minimal subset satisfying the predicate P. Remember that boolean quantifiers have already been studied in exercise 6.1.9.

Prove the following lemmas:

```
Lemma tuto_minset_eq : forall P1 P2 A,
P1 =1 P2 -> minset P1 A = minset P2 A.

Lemma tuto_minsetP : forall P A,
reflect ((P A) /\ (forall B, P B -> B \subset A -> B = A))
        (minset P A).

Lemma tuto_minsetp : forall P A, minset P A -> P A.

Lemma tuto_minsetinf : forall P A B,
  minset P A -> P B -> B \subset A -> B = A.
```

Complete the following proof:

```
Lemma tuto_ex_minset : forall P, (exists A, P A) -> {A | minset
     P A}.
Proof.
move=> P exP; pose pS n := [pred B | P B && (#|B| == n)].
pose p n := ~~ pred0b (pS n); have{exP}: exists n, p n.
  by case: exP => A PA; exists #|A|; apply/existsP; exists A;
     rewrite PA /=.
case/ex_minnP=> n; move/pred0P; case: (pickP (pS n)) => // A.
...
Qed.
```

And finally prove that:

```
Lemma tuto_minset_exists : forall P C,
   P C -> {A | minset P A & A \subset C}.
```

7 Appendix: Checking, searching, displaying information

Using large and numerous libraries developed by others is never an easy task. This section aims at giving hints to the user facing problems like: "Why does this lemma not apply?" or "Is there a lemma doing what I want here?"

7.0.1 Check

The command:

```
Check term.
```

displays the type of term. When called in proof mode, the term is checked in the local context of the current subgoal. When `term` has been defined with implicit arguments (like all the constants in the SSReflect libraries), you might encounter an error message. In that case, try again with the command:

```
Check @term.
```

where the standard Coq `@` flag disables the implicit argument mechanism.

7.0.2 Display

A more robust, but more verbose alternative to the `Check` command is:

```
Print term.
```

This command should always succeed if `term` is an object available in the context. It displays information on the declared or defined `term` object, including its body, type, and implicit arguments.

When Coq displays the current state of a proof, a lot of information can be hidden to the user such as implicit arguments or inserted coercions (this is Coq's explicit subtyping mechanism). Such hidden information is also invisible in the results of the `Check` and `Print` commands.

This default mode can be disabled by the global vernacular command:

```
Set Printing All.
```

7.0.3 Search

The vernacular command `Search` is used to browse the corpus of lemmas available in the loaded libraries. The SSReflect version of the command can be used to inspect this body selectively, using names, patterns, module names in notation-compliant way. We recall here the documentation of this command. The syntax is:

`Search` [⟨*pattern*⟩] [[-][⟨*string*⟩[`%`⟨*key*⟩] | ⟨*pattern*⟩]]* [`in` [[-]⟨*name*⟩]$^+$]`.`

where ⟨*name*⟩ is the name of an open module. This command returns the list of lemmas:

- whose *conclusion* contains a subterm matching the optional first ⟨*pattern*⟩. A - reverses the test, producing the list of lemmas whose conclusion does not contain any subterm matching the pattern;
- whose name contains the given strings. A - prefix reverses the test, producing the list of lemmas whose name does not contain the string. A string that contains symbols or is followed by a scope ⟨*key*⟩, is interpreted as the constant whose notation involves that string (e.g., + for addn), if this is unambiguous; otherwise the diagnostic includes the output of the Locate standard vernacular command.
- whose statement, including assumptions and types contains a subterm matching the next patterns. If a pattern is prefixed by -, the test is reversed;
- contained in the given list of modules, except the ones in the given modules prefixed by a -.

Note that:

- Patterns with holes should be surrounded by parentheses.
- Search always volunteers the expansion of the notation, avoiding the need to execute Locate independently. Moreover, a string fragment looks for any notation that contains fragment as a substring. If the ssrbool library is imported, the command:

  ```
  Search "~~".
  ```

  answers :

  ```
  "~~" is part of notation (~~ _)
  In bool_scope, (~~ b) denotes negb b
  negbT forall b : bool, b = false -> ~~ b
  contra forall c b : bool, (c -> b) -> ~~ b -> ~~ c
  introN forall (P : Prop) (b : bool), reflect P b -> ~ P ->
      ~~ b
  ```

- A diagnostic is issued if there are different matching notations; it is an error if all matches are partial.
- Similarly, a diagnostic warns about multiple interpretations, and signals an error if there is no default one.
- The command `Search in M.` is a way of obtaining the complete signature of the module M.
- Strings and pattern indications can be interleaved, but the first indication has a special status if it is a pattern, and only filters the conclusion of lemmas:
  - The command :

    ```
    Search (_ =1 _) "bij".
    ```

lists all the lemmas whose conclusion features a '=1' and whose name contains the string `bij`.

– The command :

```
Search "bij" (_ =1 _).
```

lists all the lemmas whose statement, including hypotheses, features a '=1' and whose name contains the string `bij`.

Exercise 7.0.3 Use the `Search` command to know the name of the constant hidden behind the `*` notation. Use the `Print` command to see how this operation is defined.

What is the constant denoted by `==>`? How is it defined? What are the lemmas concluding with something of the form `_ ==> true`?

What is the name of the lemma stating the commutativity of the `&&` operator?

An other way of guessing the name of a lemma is to infer it. Patterns might indeed reveal useless with the properties of operators are stated under a normalized form such as `(commutative andb)`. The list of operator properties used throughout SSReflect libraries can be found in the header of the `ssrfun` library source file. Lemmas in the distributed libraries respect the following name policy:

- **Generalities**
  - Most of the time the name of a lemma can be read off its statement: a lemma named `fee_fie_foe` will say something about `(fee .. ( fie ..(foe ..)..)..)`, e.g. lemma `size_cat` in `seq.v`.
  - We often use a one-letter suffix to resolve overloaded notation, e.g., `addn`, `addb`, `addr` denote `nat`, Boolean, ring addition, respectively. This policy does not necessarily apply to constants that should always be hidden behind a generic notation, and handled by a more generic theory.
  - Finally, a handful of theorems have a historical name, e.g., `Cayley_Hamilton` or `factor_theorem`.
- **Structures and Records**
  - Each structure type starts with a lower case letter, and its constructor has the same name but with a capital first letter.
  - Each instance of a structure type has a name formed with the name of the carrier type, followed by an underscore and the one of the structure type like in `seq_sub_subType`, the structure of `subType` defined on `seq_sub` (see `fintype.v`). Notable exceptions to this rule are canonical constructions taking benefits of modular name spaces, like in `ssralg.v`.
- **Suffixes**
  - Lemma whose conclusion is a predicate, or an equality for a predicate: that predicate is a suffix of the lemma name, like in `addn_eq0` or `rev_uniq`.

- Lemmas whose conclusion is a standard property such as `\char`, `<|`, etc.: the property should be indicated by a suffix (like `_char`, `_normal`, etc), so the lemma name should start by a description of the argument of the property, such as its key property, or its head constant. Thus we have `quotient_normal`, not `normal_quotient`, etc. This convention does not apply to monotony rules, for which we either use the name of the property with the suffix for the operator (e.g., `groupM`), or the name of the operator with the S suffix for subset monotony (e.g., `mulgS`).
- We try to use and maintain the following set of lemma suffixes:
  * `0` : zero, or the empty set
  * `1` : unit, or the singleton set (use `_set1` for the latter to disambiguate)
  * `2` : two, doubling, doubletons
  * `3` etc, similarly
  * `A` : associativity
  * `C` : commutativity, or set complement (use `Cr` for trailing complement)
  * `D` : set difference
  * `E` : definition elimination (often conversion lemmas)
  * `F` : Boolean false, or finite type variant (as in `canF_eq`)
  * `G` : group argument
  * `I` : set intersection
  * `J` : group conjugation
  * `K` : cancellation lemmas
  * `L` : left hand side (in `canLR`)
  * `M` : group multiplication
  * `N` : Boolean negation
  * `P` : characteristic properties (often reflection lemmas)
  * `R` : group commutator, or right hand side (in `canLR`)
  * `S` : subset argument, or integer successor (no ambiguity)
  * `T` : Boolean truth and Type-wide sets
  * `U` : set union
  * `V` : group and multiply inverse
  * `W` : weakening
  * `X` : group exponentiation, and set cartesian product

Exercise 7.0.4 What is the name of the lemma stating the commutativity of the `*` operation? What is the name of the lemma whose statement is:

```
forall b1 b2 b3, b1 || b2 || b3 = b2 || b1 || b3
```

Guess what could be the statement of the lemma `setUIl`? Verify your guess using `Check`.

Acknowledgments

The second author wishes to thank Benjamin Werner for suggesting some of the exercises proposed in this tutorial. We also thank Laurence Rideau, Laurent Théry and Enrico Tassi for proof reading and comments. We are specially grateful to François Garillot, Bas Spitters and the anonymous referees for numerous comments which have significantly improved the original version of this document.

References

[ARCT09] Andrea Asperti, Wilmer Ricciotti, Claudio Sacerdoti Coen, and Enrico Tassi. Hints in unification. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher Order Logics, TPHOLs 2009 proceedings*, volume 5674 of *Lecture Notes in Computer Science*, pages 84–98. Springer, 2009.

[BC04] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development, Coq'Art:the Calculus of Inductive Constructions*. Springer-Verlag, 2004.

[CH88] Th. Coquand and G. Huet. The Calculus of Constructions. *Information and Computation*, 76(2/3), 1988.

[GGMR09] François Garillot, Georges Gonthier, Assia Mahboubi, and Laurence Rideau. Packaging mathematical structures. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher Order Logics, TPHOLs 2009 proceedings*, volume 5674 of *Lecture Notes in Computer Science*, pages 327–342. Springer, 2009.

[GM] Georges Gonthier and Assia Mahboubi. *A small scale reflection extension for the Coq system*. INRIA Technical report, http://hal.inria.fr/inria-00258384.

[Gon08] Georges Gonthier. Formal proof - the Four Color Theorem. *Notices of the American Mathematical Society*, 55(11):1382–1393, 2008.

[GS09] Georges Gonthier and Le Roux Stéphane. An Ssreflect Tutorial. Technical Report RT-0367, INRIA, 2009. http://hal.inria.fr/inria-00407778/en/.

[PM93] C. Paulin-Mohring. Inductive Definitions in the System Coq - Rules and Properties. In M. Bezem and J.-F. Groote, editors, *Proceedings of the conference Typed Lambda Calculi and Applications*, number 664 in LNCS. Springer Verlag, 1993. Also LIP research report 92-49, ENS Lyon.

[Sai97] Amokrane Saibi. Typing algorithm in type theory with inheritance. In *Principles of Programming Languages, POPL 1997 proceedings*, pages 292–301, 1997.

[SO08] Matthieu Sozeau and Nicolas Oury. First-class type classes. In Otmane Aït Mohamed, César Muñoz, and Sofiène Tahar, editors, *Theorem Proving in Higher Order Logics, TPHOLs 2008 proceedings*, volume 5170 of *Lecture Notes in Computer Science*, pages 278–293. Springer, 2008.

[The10] The Coq Development Team. *The Coq System*. http://coq.inria.fr, 2010.

[WB89] Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad hoc. In *16'th Symposium on Principles of Programming Languages*. ACM Press, 1989.

Centre de recherche INRIA Saclay – Île-de-France
Parc Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 Orsay Cedex (France)

Centre de recherche INRIA Bordeaux – Sud Ouest : Domaine Universitaire - 351, cours de la Libération - 33405 Talence Cedex
Centre de recherche INRIA Grenoble – Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier
Centre de recherche INRIA Lille – Nord Europe : Parc Scientifique de la Haute Borne - 40, avenue Halley - 59650 Villeneuve d'Ascq
Centre de recherche INRIA Nancy – Grand Est : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex
Centre de recherche INRIA Paris – Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex
Centre de recherche INRIA Rennes – Bretagne Atlantique : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex
Centre de recherche INRIA Sophia Antipolis – Méditerranée : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399