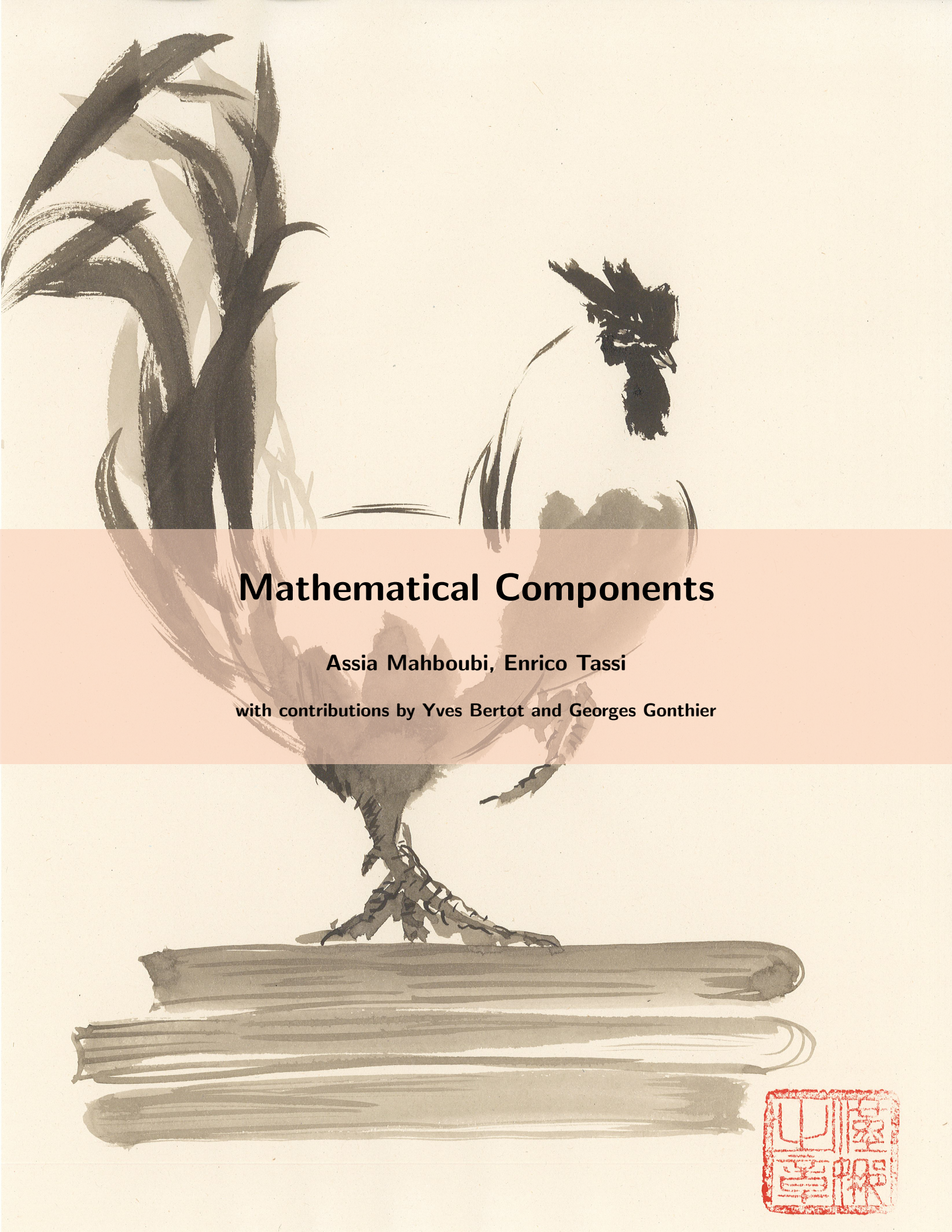


Mathematical Components

Assia Mahboubi, Enrico Tassi

with contributions by Yves Bertot and Georges Gonthier

Draft version, December 2016 v1-87-g077b493

Chapter 0

Introduction

Mathematical Components is the name of a library of formalized mathematics for the Coq system. It covers a variety of topics, from the theory of basic data structures (e.g., numbers, lists, finite sets) to advanced results in various flavors of algebra. This library constitutes the infrastructure for the machine-checked proofs of the Four Color Theorem [11] and of the Odd Order Theorem [12].

The reason of existence of this book is to break down the barriers to entry. While there are several books around covering the usage of the Coq system [3, 21, 5] and the theory it is based on [24, chapter 4][20, 25], the Mathematical Components library is built in an unconventional way. As a consequence, this book provides a non-standard presentation of Coq, putting upfront the formalization choices and the proof style that are the pillars of the library.

This books targets two classes of public. On one hand, newcomers, even the more mathematically inclined ones, find a soft introduction to the programming language of Coq, Gallina, and the Ssreflect proof language. On the other hand accustomed Coq users find a substantial account of the formalization style that made the Mathematical Components library possible.

By no means does this book pretend to be a complete description of Coq or Ssreflect: both tools already come with a comprehensive user manual [24, 13]. In the course of the book, the reader is nevertheless invited to experiment with a large library of formalized concepts and she is given as soon as possible sufficient tools to prove non-trivial mathematical results by reusing parts of the library. By the end of the first part, the reader has learned how to prove formally the infinitude of prime numbers, or the correctness of the Euclidean's division algorithm, in a few lines of proof text.

Acknowledgments. The authors wish to thank Reynald Affeldt, Guillaume Allais, Sophie Bernard, Alain Giorgetti, Laurence Rideau, Lionel Rieg, Damien Rouhling, Michael Soegtrop for their comments on earlier versions of this text. They are specially grateful to Pierre Jouvelot, Darij Grinberg, Michael Nahas and Anton Trunov for their careful proofreading and for their suggestions. Many thanks to Hanna for the illustrations.

Structure of the book

In its current state, the book has two parts which are meant to be read in order. Some more advanced sections are identified with stars, with more stars for more advanced topics.

Part 1: Definitions and proofs

This part introduces two languages and a formalization approach.

The first one is the formal language used to represent mathematics in the Coq proof assistant. It is called *Gallina* and, as expert reader may know, it is based on a variant of type theory named the Calculus of Inductive Constructions. As this part of the book explains, the very same language is used to define mathematical objects, to describe their properties and to spell out the proofs of these properties. Another distinguishing feature of this foundational framework is the status it awards to computation, and the prominent role computations shall play in proofs.

The second one is called Small Scale Reflection, *Ssreflect* for short. Most often, a user of the Coq proof assistant writes formal proofs gradually, and takes benefit from frequent interactions with the system. Ssreflect is a programming language designed to ease this activity of writing formal proofs. It has been designed to provide some support for the authors of large formal libraries, developed over a long time frame. The maintenance of large formal libraries require a solid writing discipline to be tractable, as it is the case for any large corpus of code. The *Ssreflect* language helps writing scripts that are robust to the ancillary changes triggered by improving the libraries or changing the formal definitions. Actually, small scale reflection is firstly the name of a formalization methodology. Initially conceived for the formal proof of the Four Colors Theorem, it became a pillar of the Mathematical Components library and of the formal proof of the Odd Order Theorem. The Ssreflect language was named after this methodology because of the support it provides for its implementation.

Part 2: Formalization techniques

This part provides the tools to build a large library of formalized mathematics. In particular it presents a powerful form of automation and a formalization technique that makes it possible to organize concepts in a rational way and easily define new ones by linking them to the already existing ones.

Automation is provided by *programming type inference*. The Coq system provides a user-extensible type inference algorithm. It can be extended with declarative programs giving canonical solutions to otherwise unsolvable problems. Such solutions typically involve notions and theorems that are part of the Mathematical Components library. By programming type inference one can hence teach Coq the contents of the library. The system is then able to reconstruct non-trivial missing piece of information, as an informed reader typically

does when reading a mathematical text.

Formalized knowledge is organized by means of interfaces (in the spirit of algebraic structures) and relations between them. Type inference is programmed to play the role of a librarian and recognize when an abstract theory has the right to apply to a specific example.

Finally the rich language of Coq lets one define new concepts by refining existing ones, typically by gluing an object with a proof of some extra property. Type inference is programmed to transport all the theory available on the original concept to the derived one.

How to use the book

Conventions

Sections that are labeled with one ($\star$) are of medium complexity and are of interest to the reader willing to extend the Mathematical Components library. Sections that are labeled with two ($\star\star$) are of high complexity and are of interest to the reader willing to understand the technical implementation details of the Mathematical Components library.

Tricky details typically overlooked by beginners are signaled as follows:

Warning

Mind this detail.

Advice one should keep in mind are signaled as follows:

Advice

Remember this advice.

Coq code is in `typewriter` font and `(surrounded by parentheses)` when it occurs the middle of the text. Longer snippets are in boxes with line numbers like the following one:

A sample snippet

```
1  Example Gauss n : \sum_(0 <= i < n.+1) i = (n * n.+1) %/ 2.
2  Proof.
3  elim: n =>[|n IHn]; first by apply: big_nat1.
4  rewrite big_nat_recr //= IHn addnC -divnMDl //.
5  by rewrite mulnS muln1 -addnA -mulSn -mulnS.
6  Qed.
```

Code snippets are often accompanied by the goal status printed by Coq.

Output from Coq after line 3

```
n : nat
IHn : \sum_(0 <= i < n.+1) i = (n * n.+1) %/ 2
============================
\sum_(0 <= i < n.+2) i = (n.+1 * n.+2) %/ 2
```

Names of components one can `Require` in Coq are written like ssreflect or fintype.

Running examples in the Coq system

The contents of this book is mostly about interacting with a computer program consisting of the Coq system and the Mathematical Components library. Many examples are given, and we advise readers to experiment with this program, after having installed the Coq system and the Mathematical Components library on a computer. Documentation on how to install Coq is available at http://coq.inria.fr, while documentation on how to install the Mathematical Components library is available at https://math-comp.github.io/math-comp/.

The initial revision of the book was written and tested against version 8.5 of the Coq system and version 1.6 of the Mathematical Components library.

There are a variety of ways to run the Coq system: a command line is provided under the name `coqtop`, while a windowed interface is provided under the name `coqide`. The Coq community also develops alternative approaches to integrate Coq in their preferred programming environment. For instance, there exist special extensions of Coq for the popular `Emacs` programming editor (known as `Proof General`) and for the `Eclipse` programming environment (known as `coqoon`). These extensions and similar projects can easily be found by a search on the internet.

When starting a Coq session, a few commands must be sent to the Coq system to tell it to load the Mathematical Components library and to configure its behavior so it matches the usual programming style of the Mathematical Components developers:

Default starting header

```
1 From mathcomp Require Import all_ssreflect.
2 Set Implicit Arguments.
3 Unset Strict Implicit.
4 Unset Printing Implicit Defensive.
```

The first line actually instructs the Coq system to load the first level of the Mathematical Components library. More advanced levels are available under names all_fingroup, all_algebra, all_solvable, all_field, and all_character. While the first chapters of this book rely mainly on the first level, later chapters will rely on the other levels. This will be clearly stated as the topics evolve.

Contents

Part I

Basics for Formal Mathematics

Chapter 1

Functions and Computation

In the formalism underlying the Coq system, functions play a central role akin to the one of sets in set theory. In this chapter, we give a flavor of how to define functions in Coq, how to perform computations and how to define basic mathematical objects in this formalism.

1.1 Functions

Before being more precise about the logical foundations of the Coq system in chapter 3, we review in this section some vocabulary and notations associated with functions. We will use the words *function* and *operation* interchangeably. Sometimes, we will also use the word *program* to talk about functions described in an effective way, i.e. by a code that can be executed. As a consequence we borrow from computer science some jargon, like *input* or *return*. For example we say that an operation takes as input a number and returns its double to mean that the corresponding program computes or outputs the double of its input, or that the function maps a number to its double. For this purpose, we study examples involving natural numbers. In Coq syntax, we casually use `nat` to refer to the collection of natural numbers and the infix symbol `+` (resp. `*`) to denote the addition (resp. product) operation on natural numbers, before providing their actual formal definition in section 1.2.2. We will also assume implicitly that numerals $0, 1, 2, \ldots$ denote natural numbers, represented by their analogue `0`, `1`, `2`, ... in Coq syntax.

1.1.1 Defining functions, performing computation

Mathematical formulas are expressions composed of operation symbols, of a certain arity, applied to some arguments which are either variables or themselves (sub)-expressions. In many cases these expressions are written using notations

for the operations, like the infix + the expression

$$2 + 1$$

This expression represents a natural number, obtained as the result of an operation applied to its arguments. For instance, it is the result of the binary operation of addition, applied to two natural numbers 2 and 1. The same expression can also represent the result of the unary operation of *adding one on the right to a natural number*, applied to a natural number 2.

In Coq, the operation of *adding one on the right to a natural number* is written in the following manner:

```
fun n => n + 1
```

The syntactic transformation of expression `(2 + 1)` into an explicit application of this function to one argument can be described as follows: select the subexpression that is considered as the argument of the function, here `2`, and replace it with a symbolic name, here `n`. Then encapsulate the resulting expression `(n + 1)` with the prefix "`fun n =>`". We commonly say that the prefix "`fun n =>`" *binds* the variable `n` in the expression `n + 1`. The keyword "`fun`" stands for *function*.

Now we still need to *apply* this operation to the argument `2`, and this is written as:

Adding one to two

```
(fun n => n + 1) 2
```

Note that applying a function to an argument is represented simply by writing the function on the left of the argument, with a separating space but *not necessarily with parentheses around the argument*; we will come back to this later in the present section. As a first approximation, we can see that expressions, also called *terms*, in the syntax of Coq are either variables, or functions of the form `(fun x => e)`, where `e` is itself an expression, or the application `(e1 e2)` of an expression `e1` to another expression `e2`. However, just like with pen and paper, the addition operation is denoted in the Coq syntax using an infix notation `+`. The transformation we just detailed, from expression `(2 + 1)` to expression `(fun n => n + 1) 2`, is called *abstracting `2` in `(2 + 1)`*.

While expression `(fun n => n + 1)` describes an operation without giving it a name, the usual mathematical practice would be to rely on a sentence like *consider the function f from $\mathbb{N}$ to $\mathbb{N}$ which maps n to $n + 1$*, or on a written definition like:

$$f : \begin{array}{ccc} \mathbb{N} & \to & \mathbb{N} \\ n & \mapsto & n+1 \end{array} \tag{1.1}$$

In Coq, the user can also associate a name with the operation `(fun n => n + 1)`, in the following manner:

```
Definition f := fun n => n + 1.
```

An alternative syntax for exactly the same definition is as follows and this alternative syntax is actually preferred.

```
1  Definition f n := n + 1.
```

In this syntax, the name of the argument `n` is provided *on the left of* the separating `:=`, to be then referenced *on the right of* the separating `:=`, in the actual definition of the function. The code of the actual definition is hence comprised between the `:=` symbol and the terminating dot.

The information on the domain and codomain of `f`, as in $f : \mathbb{N} \to \mathbb{N}$, is provided using the `nat` label to annotate the argument and the output in the Coq definition:

```
1  Definition f (n : nat) : nat := n + 1.
```

The label `nat` is actually called a *type*. We refer once again to chapter 3 for a more accurate description of types: in the present chapter we rely on the loose intuition that a type refers to a collection of objects that can be treated in a uniform way. A type annotation has the shape `(t : T)`, where a colon `:` surrounded by two spaces separates an expression `t` on its left from the type `T` itself on the right. For instance in `(n : nat)`, the argument `n` of the function `f` is annotated with type `nat`.

The other occurrence of `nat`, visible in `... : nat := ...`, annotates the output of the function and indicates that this output is of type `nat`. In other words, the type of any expression made of `f` applied to an argument is `nat`.

Coq provides a command to retrieve relevant information about the definitions completed so far. Here is the response to a query about `f`:

Gathering information on f

```
1  About f.
```

Response

```
f : nat -> nat
```

which confirms the type information about the domain and codomain of `f`: the arrow `->` separates the type of the input of `f` from the type of its output.

We can be more inquisitive in our requests for information about `f`, and ask also for the value behind this name `f`:

```
1  Print f.
2
```

```
f = fun n : nat => n + 1
  : nat -> nat
```

The output of this `Print` query has some similarities with the mathematical notation in (1.1) and provides both the actual definition of `f` and its type information. Actually the way the definition is printed also features a type annotation of the arguments of the function, in the fragment "`fun n : nat =>`".

Types are used in particular to avoid confusion and rule out ill-formed expressions. Coq provides a command to check whether an expression is *well typed*.

For example the function `f` we just defined can *only be applied to a natural number*, i.e., a term of type `nat`. Therefore this first query succeeds, as `3` is a natural number:

Check well-typedness

```
1 Check f 3.
```

Response

```
f 3 : nat
```

But `f` cannot be applied to something that is not a natural number, like for example a function:

Type error

```
1 Check f (fun x : nat => x + 1).
2
3
```

Response

```
Error:
The term "(fun x : nat => x + 1)"
has type "nat -> nat" while it is
expected to be "nat".
```

As expected, it makes little sense to compute the addition between a function and 1.

Expressions that are well typed can be *computed* by Coq, meaning that they are *reduced* to a "simpler" form, also called a *normal form*. For example computing `(f 3)` returns value `4`:

Evaluating a function

```
1 Eval compute in f 3.
```

Response

```
= 4 : nat
```

At the very least, we can observe that the argument `3` has been substituted for the argument variable in the definition of function `f` and that the addition has been evaluated. Although this capability of Coq plays a crucial role in the Mathematical Components library, as we will see in chapter 2, it is too early to be more precise about this normalization strategy. This would require at least describing in more details the formalism underlying the Coq system, which we do only in chapter 3. The interested reader should actually refer to [24, section 8.7.1] for the official documentation of `compute`. For now, we suggest to keep only the intuition that this normalization procedure provides the "expected value" of a computation.

1.1.2 Functions with several arguments

The syntax we used to define a function with a single argument generalizes to the case of functions with several arguments, which are then separated by a space. Here is an example of a function with two arguments:

```
1 Definition g (n : nat) (m : nat) : nat := n + m * 2.
```

In fact, for the sake of compactness, contiguous arguments of one and the same type can be grouped and share the same type annotation, so that the above definition is better written:

```
1 Definition g (n m : nat) : nat := n + m * 2.
```

which asserts firstly (by the `(n m : nat)` type annotation) that both arguments `n` and `m` have type `nat`, and secondly that the output of the function also has type `nat`, as prescribed by the `... : nat := ...` type annotation. Again, the `About` command provides information on the type of the arguments and values of `g`:

Gathering information on g	Response
`About g.`	`g : nat -> nat -> nat`

The two first occurrences of `nat` in the response `nat -> nat -> nat` assert that function `g` has two arguments, both of type `nat`; the last occurrence refers to the type of the output. This response actually reads `nat -> (nat -> nat)`, as the `->` symbol associates to the right. Otherwise said, *a multiple argument function is a single argument function that returns a function.* This idea that multiple argument functions can be represented using single argument functions (rather, for instance, than tuples of arguments) is called *currying* and will play a special role in chapter 3. For now, let us just insist on the fact that in Coq, functions with several arguments are not represented with a tuple of arguments.

Back to our example, here is an alternative definition `h` which has a single argument `n` of type `nat` and returns a function of one argument as a result:

```
Definition h (n : nat) : nat -> nat := fun m => n + m * 2.
```

Queries on the respective types of `g` and `h` provide identical answers:

Gathering information on h	Response
`About h.`	`h : nat -> nat -> nat`

If we go further in our scrutiny and ask the Coq system to print the definitions associated to names `g` and `h`, we see that these definitions are exactly the same: The Coq system does not make any difference between these two ways of describing a two-argument function.

```
Print g.

Print h.
```

```
g = fun n m : nat => n + m * 2
  : nat -> nat -> nat
h = fun n m : nat => n + m * 2
  : nat -> nat -> nat
```

Since `g` is also a one-argument function, it is sensible to apply this function to a single argument. We can call this *partial application* because `g` is also meant to be applied to two arguments. As expected, the value we obtain this way is itself a function, of a single argument, as illustrated by the following query:

Applying g partially	Response
`Check g 3.`	`g 3 : nat -> nat`

Now function `g` can be applied to the numbers `2` and `3`, as in `g 2 3`. This results in a term of type `nat`, the type of the outputs of `g`, and we can even compute this value:

```
Eval compute in g 2 3.
```

```
= 8 : nat
```

Term `(g 2 3)` actually reads `((g 2) 3)` and features three nested sub-expressions. The symbol `g` is the deepest sub-expression and it is applied to `2` in `(g 2)`. The value of this sub-expression is a function, which can be written:

```
fun m => 2 + m * 2
```

Finally, `(g 2 3)` is in turn the application of the latter function to `3`, which results in:

```
2 + 3 * 2
```

by substituting the bound variable `m` in `fun m => 2 + m * 2` by the value `3`.

1.1.3 Higher-order functions

Earlier in this section we defined functions like `f`, `g` and `h` which operate on natural numbers. Functions whose arguments are themselves functions are called *higher-order functions*. For instance, the following definition introduces a function that takes a function from `nat` to `nat` and produces a new function from `nat` to `nat`, by iterating its argument.

```
Definition repeat_twice (g : nat -> nat) : nat -> nat :=
  fun x => g (g x).
```

Once again, let us scrutinize this Coq statement. The first line asserts that the name of the function to be defined is `repeat_twice`. We also see that it has one argument, a function of type `nat -> nat`. For later reference in the definition of `repeat_twice`, the argument is given the name `g`. Finally we see that the value produced by the function `repeat_twice` is itself a function from `nat` to `nat`.

Reading the second line of this statement, we see that the value of `repeat_twice` when applied to one argument is a new function, described using the "`fun .. => ..`" construct. The argument of that function is called `x`. After the `=>` sign, we find the ultimate value of this function. This fragment of text, `g (g x)`, also deserves some explanation. It describes the application of function `g` to an expression `(g x)`. In turn, the fragment `(g x)` describes the application of function `g` to `x`. Remember that the application of a function to an argument is denoted by juxtaposing the function (on the left) and the argument (on the right), separated by a space. Moreover, application associates to the left: expressions made with several sub-expressions side by side should be read as if there were parentheses around the subgroups on the left. Parentheses are only added when they are needed to resolve ambiguities. For instance, the inner `(g x)` in `(g (g x))` needs surrounding parentheses because expression `(g g x)` reads `((g g) x)`. The latter expression would be ill-formed because it contains the sub-expression `(g g)` where `g` receives a function as argument, while it is expected to receive an argument of type `nat`.

We can play a similar game as in section 1.1.2, and scrutinize the expression obtained by applying the function `repeat_twice` to the function `f` and the number `2`. Let us compute the resulting value of this application:

```
Eval compute in repeat_twice f 2.
```

```
= 4 : nat
```

Expression `(repeat_twice f 2)` actually reads `((repeat_twice f) 2)` and features three nested sub-expressions separated by spaces. The symbol `repeat_twice`

is the deepest sub-expression and it is applied to `f` in `(repeat_twice f)`. According to the definition of the `repeat_twice` function, the value of this sub-expression is a function, which is then applied to `2`. The resulting expression is `(f (f 2))`, and given the definition of `f`, this expression can also be read as `((2 + 1) + 1)`. Thus, after computation, the result is `4`.

Function `repeat_twice` is an instance of a function with several arguments: as illustrated in section 1.1.2, its partial application to a single argument provides a well-formed function, from `nat` to `nat`:

Partial application

```
Check (repeat_twice f).
```

Response

```
repeat_twice f : nat -> nat
```

Now looking at the type of `repeat_twice` and adding redundant parentheses we obtain `((nat -> nat) -> (nat -> nat))`: once the first argument `(f : nat -> nat)` is passed, we obtain a term the type of which is the right hand side of the main arrow, that is `(nat -> nat)`. By passing another argument, like `(2 : nat)`, we obtain an expression the type of which is, again, the right hand side of the main arrow, here `nat`. Remark that each function has an arrow type and that the type of its argument matches the left hand side of this arrow (as depicted with different underline styles). When this is not the case, Coq issues a type error.

Ill-formed application

```
Check (repeat_twice f f).
```

Response

```
Error: The term "f" has type "nat
     -> nat"
while it is expected to have type
     "nat".
```

1.1.4 Local definitions

The process of abstraction described at the beginning of this section can be seen as the introduction of a name, the one used for the bound variable, in place of a sub-expression that may appear at several occurrences. For instance, in expression `(fun x => x + x + x) B`, we use the variable `x` as an abbreviation for `B`. It is specially useful when `B` happens to be a very large expression: readability is improved by avoiding the repetition of `B`, which may otherwise obfuscate the triplication pattern. In the Coq language, declaring such an abbreviation can be made even more readable by bringing closer the name of the abbreviation `x` and the expression it refers to:

```
let x := B in
   x + x + x
```

In this expression the variable `x` is *bound* and can be used in the text that comes after the `in` keyword. Variants of this syntax make it possible to state the type ascribed to the variable `x`, which may come handy when the code has to be very explicit about the nature of the values being abbreviated. Here is an example of usage of this syntax:

Evaluating local definitions

```
Eval compute in
  let n := 33 in
  let e := n + n + n in
    e + e + e.
```

Response

```
= 297 : nat
```

When it comes to comparing the values of computations, a local definition has the same result as the expression where all occurrences of the bound variable are replaced by the corresponding expression. Thus, the example expression above has exactly the same value as:

```
(33 + 33 + 33) + (33 + 33 + 33) + (33 + 33 + 33)
```

However, in practice the evaluation strategy used in a normalization command like `Eval compute in ..` takes advantage of the `let .. in ..` notation to avoid duplicating computation efforts. In our example, the value of the partial sum `(33 + 33 + 33)` is computed only once and shared at every occurrence of the bound variable. This abbreviation facility can thus also be used to organize intermediate computations.

1.2 Data types, first examples

A well-formed mathematical expression is just a juxtaposition of symbols of a given language (and variables) that respects the prescribed arities of the operations. For instance, the expression:

$$(\top \vee \bot) \wedge b$$

is a well formed expression in the language $\{\top, \bot, \vee, \wedge\}$ of boolean arithmetic, while expression:

$$0 + x \times (S\ 0)$$

is a well formed expression in the language $\{0, S, +, \times\}$ of Peano arithmetic. In these examples, however, the usual axioms that formalize the respective arithmetic confer a distinctive status to the symbols $\top$ and $\bot$ for booleans, and to the symbols 0 and S for natural numbers. More precisely, any variable-free boolean expression is equal to either $\top$ or $\bot$ modulo these axioms, and any variable-free expression in Peano arithmetic is equal either to 0 or to an iterated application of the symbol S to 0. In both cases, the other symbols in the signature represent functions, whose computational content is prescribed by the axioms.

In Coq, such expressions are represented using a *data type*, whose definition provides in a single declaration the name of the type, the symbols of (constants and) operations that *build* elements of this type, plus some rules on how to compute on these elements. These data types are introduced by the means of an *inductive type definition*. One can explicitly define more operations on the elements of the type by describing how they compute on a given argument in

the type using a case analaysis (or even a recursive definition) on the syntactic shape of this argument. This approach is used in a systematic way to define a variety of basic data types, among which boolean values, natural numbers, pairs or sequences of values are among the most prominent examples.

1.2.1 Boolean values

The collection $\mathbb{B} := \{\top, \bot\}$ of boolean values is formalized by a type called `bool`, with two inhabitants `true` and `false`. The declaration of this type happens in one of the first files to be automatically loaded when Coq starts, so booleans look like a built-in notion. Nevertheless, the type of boolean values is actually defined in the following manner:

Declaration of bool

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

It is actually one of the simplest possible inductive definitions, with only base cases, and no inductive ones. This declaration states explicitly that there are exactly two elements in type `bool`: the distinct constants `true` and `false`, called the *constructors* of type `bool`.

In practice, this means that we can *build* a well-formed expression of type `bool` by using either `true` or `false`.

Queries

```
1  Check true.
```

Response

```
true : bool
```

In order to *use* a boolean value in a computation, we need a syntax to represent the two-branch case analysis that can be performed on an expression of type `bool`. The Coq syntax for this case analysis is "`if .. then .. else ..`" as in:

```
1  if true then 3 else 2
```

More generally, we can define a function that takes a boolean value as input and returns one of two possible natural numbers in the following manner:

```
1  Definition twoVthree (b : bool) := if b then 2 else 3.
```

As one expects, when `b` is `true`, the expression `(twoVthree b)` evaluates to `2`, while it evaluates to `3` otherwise:

Evaluating twoVthree

```
1  Eval compute in twoVthree true.
2  Eval compute in twoVthree false.
```

Response

```
   = 2 : nat
   = 3 : nat
```

As illustrated on this example, the `compute` command rewrites any term of the shape `if true then t1 else t2` into `t1` and any term of the shape `if false then t1 else t2` into `t2`.

The Mathematical Components library provides a collection of boolean operations that mirror reasoning steps on truth values. The functions are called `negb`, `orb`, `andb`, and `implyb`, with notations `~~`, `||`, `&&`, and `==>`, respectively (the last three operators are infix, i.e., they appear between the arguments, as in `b1 && b2`). Note that the symbol `~~` uses two characters `~`: it should not be confused with two consecutive occurrences of the one-character symbol `~`, which would normally be written with a separating space. The latter has a meaning in Coq, but is almost never used in the Mathematical Components library.

For instance, the functions `andb` and `orb` are defined as follows.

```
1  Definition andb (b1 b2 : bool) := if b1 then b2   else false.
2  Definition orb  (b1 b2 : bool) := if b1 then true else b2.
```

1.2.2 Natural numbers

The collection of natural numbers is formalized by a type called `nat`. An inhabitant of this type is either the constant `O` (capital "o" letter) representing zero, or an application to an existing natural number of the function symbol `S` representing the successor:

```
1  Inductive nat := O | S (n : nat).
```

This inductive definition of the expressions of type `nat` has one base case, the constant `O`, and one inductive case, for the natural numbers obtained using the successor function at least once: therefore `O` has type `nat`, `(S O)` has type `nat`, so does `(S (S O))`, and so on, and any natural number has this shape. The constant `O` and the function `S` are the constructors of type `nat`.

When interacting with Coq, we will often see decimal notations, but these are only a parsing and display facility provided to the user for readability: `O` is displayed 0, `(S O)` is displayed 1, etc. Users can also type decimal numbers to describe values in type `nat`: these are automatically translated into terms built (only) with `O` and `S`. In the rest of this chapter, we call such a term a *numeral*.

The Mathematical Components library provides a few notations to make the use of the constructor `S` more intuitive to read. In particular, if `x` is a value of type `nat`, `x.+1` is another way to write `(S x)`. The ".+1" notation binds more strongly than function application, so that this notation makes it possible to avoid some needs for parentheses: assuming that `f` is a function of type `nat -> nat` the expression `(f n.+1)` reads `(f (S n))`.

Queries	Response
`1  Check fun n => f n.+1.`	`fun n : nat => f n.+1 : nat -> nat`

When defining functions that operate on natural numbers, we can proceed by case analysis, as was done in the previous section for boolean values. Here as well, there are two cases: either the natural number used in the computation

is `0` or it is `p.+1` for some `p`, and the value of `p` may be used to describe the computations to be performed. This case analysis can be seen as matching against *patterns*: if the data fits one of the patterns, then the computation proceeds with the expression in the corresponding branch. Such a case analysis is therefore also called *pattern matching*. Here is an example:

```
Definition predn n := if n is p.+1 then p else n.
```

The function `predn` returns the predecessor of a natural number when it exists, and 0 otherwise. In this definition `p.+1` is a pattern. The value bound to the name `p` mentioned in this pattern is not known in advance; it is actually computed at the moment the `n` is matched against the pattern. For instance:

Evaluating predn	Response
`Eval compute in predn 5.`	`= 4 : nat`

The `compute` command rewrites any term of the shape `if 0 is p.+1 then t1 else t2` into `t2` and any term of the shape `if k.+1 is p.+1 then t1 else t2` into `t1` where all occurrences of `p` have been replaced by `k`. In our example, as the value of `k.+1` is `5`, `k` and hence `t2` is `4`.

The symbols that are allowed in a pattern are essentially restricted to the constructors, here `0` and `S`, and to variable names. Thanks to notations however, a pattern can also contain occurrences of the notation "`.+1`" which represents `S`, and decimal numbers, which represent the corresponding terms built with `S` and `0`. When a variable name occurs, this variable can be re-used in the result part. Remark that we did omit the type of the input `n`. We can omit this type because matching `n` against the `p.+1` pattern imposes that `n` has type `nat`, as the `S` constructor belongs to *exactly* one inductive definition, namely the one of `nat`.

The pattern used in the `if` statement can be composed of several nested levels of the `.+1` pattern. For instance, if we want to write a function that returns $n - 5$ for every input n larger than or equal to 5 and 0 for every input smaller than 5, we can write the following definition:

```
Definition pred5 n :=
  if n is u.+1.+1.+1.+1.+1 then u else 0.
```

On the other hand, if we want to describe a different computation for three different cases and use variables in more than one case, we need the more general "`match .. with .. end`" syntax. Here is an example:

```
Definition three_patterns n :=
  match n with
    u.+1.+1.+1.+1.+1 => u
  | v.+1 => v
  | 0 => n
  end.
```

This function maps any number n larger than or equal to 5 to $n - 5$, any number $n \in \{1, \ldots, 4\}$ to $n - 1$, and 0 to 0.

The pattern matching construct "`match .. with .. end`" may contain an arbitrarily large number of *pattern matching rules* of the form "*pattern*`=>`*result*" separated by the `|` symbol. Optionally one can prefix the first pattern matching rule with `|`, in order to make each line begin with `|`. Each pattern matching rule results in a new rewrite rule available to the `compute` command. All the pattern matching rules are tried successively against the input. The patterns may overlap, but the result is given by the first pattern that matches. For instance with the function `three_patterns`, if the input is `2`, in other words `0.+1.+1`, the first rule cannot match, because this would require that `0` matches `u.+1.+1.+1` and we know that 0 is not the successor of any natural number; when it comes to the second rule `0.+1.+1` matches `v.+1`, because the rightmost `.+1` in the value of `2` matches the rightmost `.+1` part in the pattern and `0.+1` matches the `v` part in the pattern.

A fundamental principle is enforced by Coq on case analysis: *exhaustiveness*. The patterns must cover all constructors of the inductive type. For example, the following definition is rejected by Coq.

Non-exhaustive case analysis

```
Definition wrong (n : nat) :=
  match n with 0 => true end.

```

Response

```
Error: Non exhaustive pattern-
    matching:
no clause found for
pattern S _
```

We conclude the section by showing a syntactic facility to scrutinize multiple values at the same time.

```
Definition same_bool b1 b2 :=
  match b1, b2 with
  | true, true => true
  | false, false => true
  | _, _ => false
  end.
```

Here, the reserved word `_` stands for a "throwaway variable", i.e., a variable that we choose to give no name because we are not going to reference it (for example, the constant function `fun (n : nat) => 2` can also be written `fun (_ : nat) => 2`).

The above code defining `same_bool` is parsed as follows:

```
Definition same_bool b1 b2 :=
  match b1 with
  | true => match b2 with true => true | _ => false end
  | false => match b2 with true => false | _ => true end
  end.
```

1.2.3 Recursion on natural numbers

Using constructors and pattern matching, it is possible to add or subtract one, but not to describe the addition or subtraction of arbitrary numbers. For this purpose, we resort to recursive definitions. The addition operation is defined in

the following manner:

```
Fixpoint addn n m :=
  if n is p.+1 then (addn p m).+1 else m.
```

As this example illustrates, the keyword for defining a recursive function in Coq is `Fixpoint`: the function being defined, here called `addn`, is used in the definition of the function `addn` itself. This text expresses that the value of `(addn p.+1 m)` is `(addn p m).+1` and that the value `(addn 0 m)` is `m`. This first equality may seem redundant, but there is progress when reading this equality from left to right: an addition with `p.+1` as the first argument is explained with the help of addition with `p` as the first argument, and `p` is a smaller number than `p.+1`. When considering the expression `(addn 2 3)`, we can know the value by performing the following computation:

`(addn 2 3)`	use the "then" branch, `p = 1`
`(addn 1 3).+1`	use the "then" branch, `p = 0`
`(addn 0 3).+1.+1`	use the "else" branch
`3.+1.+1`	remember that `5 = 3.+1.+1`

When the computation finishes, the symbol `addn` disappears. In this sense, the recursive definition is really a definition. Remark that what the `(addn n m)` program does is simply to pile `n` times the successor symbol on top of `m`.

An alternative way of writing `addn` is to provide explicitly the rules of the pattern-matching at stake instead of relying on an `if` statement. This can be written as follows:

```
Fixpoint addn n m :=
  match n with
  | 0 => m
  | p.+1 => (addn p m).+1
  end.
```

With this way of writing the recursive function, it becomes obvious that pattern-matching rules describe equalities between two symbolic expressions, but these equalities are always used from left to right during computations.

When writing recursive functions, the Coq system imposes the constraint that the described computation must be *guaranteed to terminate*. The reason for this requirement is sketched in section 3.2.3. This guarantee is obtained by analyzing the description of the function, making sure that recursive calls always happen on a given argument that decreases. Termination is obvious when the recursive calls happen only on "syntactically smaller arguments". For instance, in our example `addn`, the function is defined by matching its first argument `n` against the patterns `p.+1` and `0`; in the branch corrsponding to the pattern `p.+1`, the recursive call happens on `p`, a strict subterm of `p.+1`. Had we matched the argument `n` against the pattern `p.+1.+1`, then the recursive call would have been allowed on arguments `p` or `p.+1`, but not `p.+1.+1`.

An erroneous, in the sense of non-terminating, definition is rejected by Coq:

Here and [illegible] we do not [illegible] the fact that `.+1.+1` is displayed `.+2`

Non-terminating program

```
Fixpoint loop n :=
 if n is 0 then loop n else 0.

```

Response

```
Error: Recursive call to loop has
principal argument equal to "n"
instead of a subterm of "n".
```

If addition amounts to repeating the operation of applying `.+1` to one of the arguments, subtraction amounts to repeating the operation of fetching a subterm of the first argument. This is also easily expressed using pattern matching constructs. Here again, subtraction is already defined in the libraries, but we can play the game of re-defining our own version.

```
Fixpoint subn m n : nat :=
  match m, n with
  | p.+1, q.+1 => subn p q
  | _ , _ => m
  end.
```

From a mathematical point of view, this definition can be quite unsettling. The second pattern matching rule indicates that when any argument of the subtraction is 0, then the result is the first argument. This is exactly what one would expect when the second argument is 0; but this rule also covers the case where the second argument is non-zero while the first argument is 0: in that case, the result of the function is zero.

On paper, the expression $m - n$, for $m, n \in \mathbb{N}$, is usually understood as an integer, which is negative when $n > m$. But here the output of `subn` is constraint by its output type to be a natural number. Moreover, any function defined in Coq has to be total, i.e., has to assign a value to any element in the type of its arguments.

Since `subn` has type `nat -> nat -> nat`, an element *in type* `nat` has to be output by the function even in the case when the second argument is non-zero and the first argument is zero. The default value output in this case is `0`, so that in the end `subn` sends two naturals m and n to $\max\{m - n, 0\}$ as opposed to $m - n$.

Thus, mathematically speaking, `subn` sends two naturals m and n to $\max\{m - n, 0\}$ as opposed to $m - n$. This oddity is imposed by the fact that any function defined in Coq has to be total, i.e., has to assign a value to any element in the type of its arguments. Since `subn` has type `nat -> nat -> nat`, an element *in type* `nat` has to be output by the function even in the case when the second argument is non-zero and the first argument is zero. This problem disappears with the introduction of another type of numbers, with negative integers. (Alternatively, it is possible to implement partial functions by extending the output type; see Section 1.3.3.)

The discussion above needs to be improved. And subn is may be not the right example, for its main raison d'tre is the definition of comparison.

We can also write a recursive function with two arguments of type `nat`, that returns `true` exactly when the two arguments are equal:

```
Fixpoint eqn m n :=
  match m, n with
  | 0, 0 => true
  | p.+1, q.+1 => eqn p q
  | _, _ => false
  end.
```

The last rule in the code of this function actually covers two cases : `0, _.+1` and `_.+1, 0`.

For equality test functions, it is useful to add a more intuitive notation. For instance we can attach a notation to `eqn` in the following manner:

```
Notation "x == y" := (eqn x y).
```

Now that we have programmed this equality test function, we can verify that the Coq system really identifies various ways to write the same natural number.

```
Eval compute in 0 == 0.
Eval compute in 1 == S 0.
Eval compute in 1 == 0.+1.
Eval compute in 2 == S 0.
Eval compute in 2 == 1.+1.
Eval compute in 2 == addn 1 0.+1.
```

```
= true : bool
= true : bool
= true : bool
= false : bool
= true : bool
= true : bool
```

In this section, we introduced a variety of functions and notations for operations on natural numbers. In practice, these functions and notations are already provided by the Mathematical Components library. In particular it provides addition (named `addn`, infix notation `+`), multiplication (`muln`, `*`), subtraction (`subn`, `-`), division (`divn`, `%/`), modulo (`modn`, `%%`), exponentiation (`expn`, `^`), equality comparison (`eqn`, `==`), order comparison (`leq`, `<=`) on natural numbers. All these operations output natural numbers: as explained above, subtraction is made to return 0 when the subtrahend exceeds the minuend; similarly, division is integer division. The trailing `n` in the names is chosen to signal that these operations are on the `nat` data type. Postfix notations such as `.-1` and `.*2` are provided for the predecessor and double functions.

We detail here the definition of `leq` since it will be often used in examples.

```
Definition leq m n := m - n == 0.
Notation "m <= n" := (leq m n).
```

Note that this definition crucially relies on the fact that subtraction computes to `0` whenever the first argument is less than or equal to the second argument.

The Mathematical Components library also provides concepts that make sense only for the `nat` data type, like the test functions identifying `prime` and `odd` numbers. In that case, the trailing `n` is omitted in their name.

We strongly advise the reader wanting to explore the Mathematical Components library to browse their source files[1] and not to limit herself to interactive

[1] http://math-comp.github.io/math-comp/htmldoc/libgraph.html

queries to the system. The information provided by the `Print` and `About` commands is useful to understand how to use the objects defined in the libraries once they are known by their name. By contrast, the source files describe what is formalized, under which name and notation.

Advice

Each file in the Mathematical Components library starts with a banner describing all the concepts and associated notations introduced by the file. There is no better way to browse the library than reading these banners.

1.3 Containers

A *container* is a data type which describes a collection of objects grouped together so that they can be manipulated as a single object. For instance, we might want to compute the sequence of all predecessors or all divisors of a number. We could define the following data type for this purpose:

```
Inductive listn := niln | consn (hd : nat) (tl : listn).
```

The elements of this data type are (one possible implementation of) lists of zero or more natural numbers; the first constructor `niln` builds the empty list, whereas the second constructor `consn` builds a nonempty list by combining a natural number `hd` with an already existing list `tl`. For example:

```
Check consn 1 (consn 2 niln).
Check consn true (consn false niln).


```

```
consn 1 (consn 2 niln) : listn
Error: The term "true" has
type "bool" while it is
expected to have type "nat".
```

As expected, `listn` cannot hold boolean values. So if we need to manipulate a list of booleans we have to define a similar data type: `listb`.

```
Inductive listb := nilb | consb (hd : bool) (tl : listb).
```

This approach is problematic for two reasons. First, every time we write a function that manipulates a list, we have to decide a priori if the list holds numbers or booleans, even if the program does not really use the objects held in the list. A concrete example is the function that computes the size of the list; in the current setting such a function has to be written twice. Worse, starting from the next chapter we will prove properties of programs, and given that the two size functions are "different", we would have to prove such properties twice.

However it is clear that the two data types we just defined follow a similar schema, and so do the two functions for computing the size of a list or the

theorems we may prove about these functions. Hence, one would want to be able to write something like the following, where α is a schematic variable:

```
1  Inductive list := nil | cons (hd : α) (tl : list).
```

This may look familiar jargon to some readers. In the present context however, we would rather like to avoid appealing to any notion of schema, that would be somehow added on top of the Coq language. This way, we will make possible the writing of formal sentences with arbitrary quantifications on this parameter α.

1.3.1 The (polymorphic) sequence data type

The Mathematical Components library provides a generic data type to hold several objects of any given type `A`. This is the data type `seq`, defined as follows:

```
1  Inductive seq (A : Type) := nil | cons (hd : A) (tl : seq A).
```

The name `seq` refers to (finite) "sequences", also called "lists". This definition actually describes the type of lists as a *polymorphic type*. This means that there is a different type `(seq A)` for each possible choice of type `A`. For example `(seq nat)` is the type of sequences of natural numbers, while `(seq bool)` is the type of sequences of booleans. The type `(seq A)` has two constructors, named `nil` and `cons`. Constructor `nil` represents the empty sequence. The type of the constructor `cons` is devised specifically to describe how to produce a new list of type `(seq A)` by combining an element of `A`, the *head* of the sequence, and an existing list of type `(seq A)`, the *tail* of the sequence. This also means that this data-type does not allow users to construct lists where the first element would be a boolean value and the second element would be a natural number.

In the declaration of `seq`, the keyword `Type` denotes the *type of all data types*. For example `nat` and `bool` are of type `Type`, and can be used in place of `A`. In other words `seq` is a function of type `(Type -> Type)`, sometimes called a *type constructor*. The symbol `seq` alone does not denote a data type, but if one passes to it a data type, then it builds one. Remark that this also means that `(seq (seq nat))` is a valid data type (namely, the type of lists of lists of natural numbers), and that the construction can be iterated. Types again avoid confusion: it is not licit to form the type `(seq 3)`, since the argument `3` has type `nat`, while the function `seq` expects an argument of type `Type`.[2]

In principle, the constructors of such a polymorphic data type feature a type argument: `nil` is a function that takes a type `A` as argument and returns an empty list of type `(seq A)`. The *type* of the output of this function hence depends on the *value of this input*. The type of `nil` is thus not displayed with the arrow notation ".. -> .." that we have used so far for the type of functions. It is rather written as follows:

[2]For historical reasons Coq may display the type of `nat` or `bool` as `Set` and not `Type`. We beg the reader to ignore this detail, which plays no role in the Mathematical Components library.

```
1    ∀ A : Type, seq A
```

so as to *bind* the value `A` of the argument in the type of the output. The same goes for the other constructor of `seq`, named `cons`. This function actually takes three arguments: a type `A`, a value in this type, and a value in the type `(seq A)`. The type of `cons` is thus written as follows:

```
1    ∀ A : Type, A -> seq A -> seq A
```

Since the type of the output does not depend on the second and third arguments of `cons`, respectively of type `A` and `(seq A)`, the type of `cons` features two arrow separators for these. Altogether, we conclude that the sequence holding a single element `2 : nat` can be constructed as `(cons nat 2 (nil nat))`. Actually, the two occurrences of type `nat` in this term are redundant: the tail of a sequence is a sequence with elements of the same type. Better yet, this type can be *inferred* from the type of the given element `2`. One can thus write the sequence as `(cons _ 2 (nil _))`, using the placeholder `_` to denote a subterm, here a type, to be inferred by Coq. In the case of `cons`, however, one can be even more concise. Let us ask for information about `cons` using the command `About`:

Query	Response

`1 About cons.`
`2`
`3`
	`cons : ∀ A : Type, A -> seq A -> seq A`
	`...`
	`Argument A is implicit and maximally inserted`

The Coq system provides a mechanism to avoid that users need to give the type argument to the `cons` function when it can be inferred. This is the information meant by the message "`Argument A is implicit and ..`". Every time users write `cons`, the system automatically inserts an argument in place of `A`, so that this argument does not need to be written (the argument is *implicit*). It is then the job of the Coq system to guess what this argument is when looking at the first explicit argument given to the function. The same happens to the type argument of `nil` in the built-in version of `seq` provided by the Mathematical Components library (although not in the version we defined above). In the end, this ensures that users can write the following expression.

This comment should disappear thanks to a more timely mention of implicit arguments.

Query	Response
`1 Check cons 2 nil.`	`[:: 2] : seq nat`

This example shows that the function `cons` is only applied explicitly to two arguments (the two arguments effectively declared for `cons` in the inductive type declaration). The first argument, which is implicit, has been guessed so that it matches the actual type of `2`. Also for `nil` the argument has been guessed to match the constraints that it is used in a place where a list of type `(seq nat)` is expected.

To locally disable the status of implicit arguments, one can prefix the name of a constant with `@` and pass all arguments explicitly, as in `(@cons nat 2 nil)` or `(@cons nat 2 (@nil nat))` or even `(@cons _ 2 (@nil _))`.

This example, and the following ones, also show that Coq and the Mathematical Components library provide a collection of notations for lists.

Query

```
Check 1 :: 2 :: 3 :: nil.
Check fun l => 1 :: 2 :: 3 :: l.

```

Response

```
[:: 1; 2; 3] : seq nat
fun l : seq nat => [:: 1, 2, 3 &
    l]
  : seq nat -> seq nat
```

In particular Coq provides the infix notation `::` for `cons`. The Mathematical Components library follows a general pattern for n-ary operations obtained by the (right-associative) iteration of a single binary one. In particular `[::` begins the repetition of `::` and `]` ends it. Elements are separated by `,` (comma) but for the last one separated by `&`. For example, the above `[:: 1; 2; 3 & l]` stands for `1 :: (2 :: (3 :: l))`. For another example, one can write the boolean conjunction of three terms as `[&& true, false & true]`.[3] For sequences that are `nil`-terminated, a very frequent case, the Mathematical Components library provides an additional notation where all elements are separated by `;` (semi-colon) and the last element, `nil`, is omitted.

Pattern matching can be used to define functions on sequences, like the following example which computes the first element of a non-empty sequence, with a default value for the empty case:

```
Definition head T (x0 : T) (s : seq T) := if s is x :: _ then x else x0.
```

1.3.2 Recursion for sequences

Terms of type `(seq A)`, for a type `A`, are finite piles of `cons` constructors, terminated with a `nil`. They are similar to the terms of type `nat`, except that each `cons` constructor carries a datum of type `A`. Here as well, recursion provides a way to process sequences of arbitrary size.

The Coq system provides support for the recursive definition of functions over any inductive type (not just `nat`). The recursive definition of the value of a function at a given constructor can use the value of the function at the arguments of the constructor. This defines the function over the entire type since all values of an inductive type are finite piles of constructors.

The size function counts the number of elements in a sequence:

```
Fixpoint size A (s : seq A) :=
  if s is _ :: tl then (size tl).+1 else 0.
```

During computation on a given sequence, this function traverses the whole sequence, incrementing the result for every `cons` that is encountered. Note that in this definition, the function `size` is described as a two argument function, but the recursive call `(size tl)` is done by providing explicitly only one argument, `tl`. Remember that the Coq system makes arguments of functions that can be

[3] Some n-ary notations use a different, but more evocative, last separator. For example one writes `[|| b1, b2 | b3]` and `[==> b1, b2 => b3]`.

guessed from the type of the following arguments automatically implicit, and `A` is implicit here. This feature is already active in the expression defining `size`.

Another example of recursive function on sequences is a function that constructs a new sequence whose entries are values of a given function applied to the elements of an input sequence. This function can be defined as:

```
Fixpoint map A B (f : A -> B) s :=
  if s is e :: tl then f e :: map f tl else nil.
```

This function provides an interesting case study for the definition of appropriate notations. For instance, we will add a notation that makes it more apparent that the result is *the sequence of all expressions $f(i)$ for i taken from another sequence.*

```
Notation "[ 'seq' E | i <- s ]" := (map (fun i => E) s).
```

For instance, with this notation we write the computation of successors for a given sequence of natural numbers as follows:

Query	Response
`Eval compute in [seq i.+1 | i <- [:: 2; 3]].`	`= [:: 3; 4] : seq nat`

In addition to the function `map` and the associated notation we describe here, the Mathematical Components library provides a large collection of useful functions and notations to work on sequences, as described in the header of the file `seq`. For instance `[seq i <- s | p]` filters the sequence `s` keeping only the values selected by the boolean test `p`. Another useful function for sequences is `cat` (with infix notation `++`) that is used to concatenate two sequences together.

1.3.3 Option and pair data types

Here is another example of polymorphic data type, which represents a box that can be either empty, or contain a single value:

```
Inductive option A := None | Some (a : A).
```

Akin to a pointed version of `A`, type `(option A)` contains a copy of all the elements of `A`, built using the `Some` constructor, plus an extra element given by the constructor `None`. It may be used to represent the output of a partial function or of a filtering operation, using `None` as a default element.

For example, function `only_odd` "filters" natural numbers, keeping the odd ones and replacing the evens by the `None` default value:

```
Definition only_odd (n : nat) : option nat :=
  if odd n then Some n else None.
```

This fails since None has no implicits

Similarly, one can use the `option` type to define a partial function which computes the head of a non-empty list:

```
Definition ohead (A : Type) (s : seq A) :=
  if s is x :: _ then Some x else None.
```

See also the sub-type kit presented in chapter 6, which makes use of the option type to describe a partial injection.

Another typical polymorphic data type is the one of pairs, that lets one put together any two values:

```
Inductive pair (A B : Type) : Type := mk_pair (a : A) (b : B).
Notation "( a , b )" := (mk_pair a b).
Notation "A * B" := (pair A B).
```

The type `pair` has two type parameters, `A` and `B`, so that it can be used to form any instance of pairs: `(pair nat bool)` is the type of pairs with an element of type `nat` in its first component and one of type `bool` in its second, but we can also form `(pair bool nat)`, `(pair bool bool)`, etc. The type `pair` is denoted by an infix `*` symbol, as in `(nat * bool)`. This inductive type has a *single constructor* `mk_pair`. It takes over the two polymorphic parameters, that become its two first, implicit arguments. The constructor `mk_pair` has two more explicit arguments which are the data stored in the pair. This constructor is associated with a notation so that `(a, b)` builds the pair of `a` and `b`, and `(a, b)` has type `(pair A B)`. For a given pair, we can extract its first element, and we can provide a polymorphic definition of this projection:

```
Definition fst A B (p : pair A B) :=
  match p with mk_pair x _ => x end.
```

We leave as an exercise the definition of the projection on the second component of a pair. The Mathematical Components library has a notation for these projections: `c.1` is the first component of the pair `c` and `c.2` is its second.

```
Check (3, false).
Eval compute in (true, false).1.
```

```
(3, false) : nat * bool
 = true : bool
```

As one expects, pairs can be nested. Coq provides a slightly more complex notation for pairs, which makes possible to write `(3,true,4)` for `((3,true),4)`. In this example, the value `true` is the second component of this tuple, or more precisely the second component of its first one: it can thus be obtained as `(3,true,4).1.2`. This may be a drawback, consequence of representing tuples as nested pairs. If tuples of a certain fixed length are pervasive to a development, one may consider defining another specific container type for this purpose. See for instance exercise 1 for the definition of triples.

We conclude this example with a remark on notations. After declaring such a notation for the type of pairs, the expression `(a * b)` becomes "ambiguous", in the sense that the same infix `*` symbol can be used to multiply two natural numbers as in `(1 * 2)` but also to write the type of pairs `(nat * bool)`. Such an ambiguity is somewhat justified by the fact that the pair data type can be seen as the (Cartesian) product of the arguments. The ambiguity can be resolved by annotating the expression with a specific label: `(a * b)%N` interprets the infix `*` as multiplication of natural numbers, while `(a * b)%type` would interpret `*` as the pair data type constructor. The `%N` and `%type` labels are said to be *notation scope delimiters* (for more details see [24, section 12.2]).

1.3.4 Aggregating data in record types

Inductive types with a single constructor, like the type `pair` in section 1.3.3 or the type `triple` of exercise 1, provide a general pattern to define a type which aggregates existing objects into a single packaged one. This need is so frequent that Coq provides a specialized command to declare this class of data type, called `Record`.

For example here is an instance of a type representing triples of natural numbers, which can be used to represent a grid point in a 3-dimensional cone:

```
1  Record point : Type := Point { x : nat; y : nat; z : nat }.
```

This line of code defines an inductive type `point`, with no parameter and with a single constructor `Point`, which has three arguments each of type `nat`. Otherwise said, this type is:

```
1  Inductive point : Type := Point (x : nat) (y : nat) (z : nat).
```

Using the `Record` version of this definition instead of its equivalent `Inductive` allows us to declare names for the projections at the time of the definition. In our example, the record `point` defines three projections, named `x`, `y` and `z` respectively. In the case where they come from a record definition, these projections are also called *fields* of the record. One can thus write:

Query	Response
`1 Eval compute in x (Point 3 0 2).` `2 Eval compute in y (Point 3 0 2).`	`= 3 : nat` `= 0 : nat`

As expected, the code for the `x` projection is:

```
1  Definition x (p : point) := match p with Point a _ _ => a end.
```

When an inductive type has a single constructor, like in the case of `pair` or for records, the name of this constructor is not relevant in pattern matching, as the "case analysis" has a single, irrefutable, branch. There is a specific syntax for irrefutable patterns, letting one rewrite the definition above as follows:

```
1  Definition x (p : point) := let: Point a _ _ := p in a.
```

Record types are as expressive as inductive types with one constructor: they can be polymorphic, they can package data with specifications, etc. In particular, they will be central to the formalization techniques presented in Part II.

1.4 The Section mechanism

When several functions are designed to work on similar data, it is useful to set a working environment where the common data is declared only once. Such a working environment is called a `Section`, and the data that is local to this

section is declared using `Variable` commands. A typical example happens when describing functions that are polymorphic. In that case, definitions rely in a uniform way on a given type parameter plus possibly on existing functions in this type.

```
Section iterators.

Variables (T : Type) (A : Type).
Variables (f : T -> A -> A).

Implicit Type x : T.

Fixpoint iter n op x :=
  if n is p.+1 then op (iter p op x) else x.

Fixpoint foldr a s :=
  if s is y :: ys then f y (foldr a ys) else a.

End iterators.
```

The `Section` and `End` keywords delimit a scope in which the types `T` and `A` and the function `f` are given as parameters: `T`, `A` and `f` are called *section variables*. These variables are used in the definition of `iter` and `foldr`.

The `Implicit Type` annotation tells Coq that, whenever we name an input `x` (or `x'`, or `x1`, ...), its type is supposed to be `T`. Concretely, it lets us omit an explicit type annotation in the definition of programs using `x`, such as `iter`. The `Implicit Type` command is used frequently in the Mathematical Components library; the reader can refer to [24, section 2.7.18] for a more detailed documentation of it.

When the section is closed (using the `End` command), these variables are abstracted: i.e., from then on, they start appearing as arguments to the various functions that use them in the very same order in which they are declared. Variables that are not actually used in a given definition are omitted. For example, `f` plays no role in the definition of `iter`, and thus does not become an argument of `iter` outside the section. This process is called an *abstraction mechanism*.

Concretely, the definitions written inside the section are elaborated to the following ones.

```
Fixpoint iter (T : Type) n op (x : T) :=
  if n is p.+1 then op (iter p op x) else x.
Fixpoint foldr (T A : Type) (f : T -> A -> A) a s :=
  if s is x :: xs then f x (foldr f a xs) else a.
```

Finally, remark that `T` and `A` are implicit arguments; hence they are not explicitly passed to `iter` and `fold` in the recursive calls.

```
About foldr.

```

```
foldr : ∀ T A : Type, (T -> A -> A) -> A -> seq T -> A
Arguments T, A are implicit ...
```

We can now use `iter` to compute, for example, the subtraction of `5` from `7`, or `foldr` to compute the sum of all numbers in `[:: 1; 2]`:

```
Eval compute in iter 5 predn 7.
Eval compute in foldr addn 0 [:: 1; 2].
```

```
= 2 : nat
= 3 : nat
```

1.5 Symbolic computation

As we mentioned in section 1.1, the `Eval compute` command of Coq can be used to normalize expressions, which eventually leads to simpler terms, like numerals. This flavour of computation can however accommodate to the presence of parameters in the expression to be computed.

```
Section iterators.

Variables (T : Type) (A : Type).
Variables (f : T -> A -> A).

Fixpoint foldr a s :=
  if s is x :: xs then f x (foldr a xs) else a.
```

If we ask for the type of `foldr` in the middle of the section, we see that it is not a polymorphic function (yet).

```
About foldr.
```

```
foldr : A -> seq T -> A
```

We hence postulate a term of type `A` and two of type `T` in order to apply `foldr` to a two-element list, and we ask Coq to compute this expression.

```
Variable init : A.
Variables x1 x2 : T.
Eval compute in foldr init [:: x1; x2].
```

```
= f x1 (f x2 init) : A
```

The symbols `f`, `x1`, `x2` and `init` are inert: They represent unknown values, hence computation cannot proceed any further. Still Coq has developed the expression symbolically. To convince ourselves that such expression is meaningful we can try to substitute `f`, `x1`, `x2` and `init` with the values we used at the end of the last section to play with `foldr`, namely `addn`, `1`, `2` and `0`:

```
Eval compute in addn 1 (addn 2 0).
```

```
= 3 : nat
```

The expression, which now contains no inert symbols, computes to the numeral 3, the very same result we obtained by computing `(foldr addn 0 [:: 1; 2])` directly.

The way functions are described as programs has an impact on the way symbolic computations unfold. For example, recall from section 1.2.3 the way we defined an addition operation on elements of type `nat`. It was defined using the `if .. is .. then .. else ..` syntax:

```
Fixpoint addn n m := if n is p.+1 then (addn p m).+1 else m.
```

Now let us consider the following alternative definition:

```
Fixpoint add n m := if n is p.+1 then add p m.+1 else m.
```

Both are sensible definitions, and we can show that the two addition functions compute the same results when applied to numerals. Still, their computational behavior may differ when computing on arbitrary symbolic values. In order to highlight this, we will use another normalization strategy to perform computation, the `simpl` evaluation strategy. One difference between the `simpl` and `compute` strategies is that the `simpl` one usually leaves expressions in nicer forms whenever they contain variables. Here again we point the interested reader to [24, section 8.7.1] for more details.

predn has a notation attached to it

```
Variable n : nat.
Eval simpl in (add  n.+1 7).-1.
Eval simpl in (addn n.+1 7).-1.
```

```
= (add n 8).-1 : nat
= addn n 7 : nat
```

Here we see the impact of the difference in the definitions of `addn` and `add`: the `add` variant transfers the number of pebbles (represented by the successor `S` symbol) given as first argument to its second argument before resorting to its base case, whereas in the `addn` variant, the resulting pile of pebbles is constructed top down. An intermediate expression in the computation of `(addn n m)` exposes as many successors as recursive calls have been performed so far. Since bits of the final result are exposed early, the `predn` function can eventually compute, and it cancels the `.+1` coming out of the sum. On the other hand, `add` does not expose a successor when a recursive call is performed; hence symbolic computation in `predn` is stuck.

As chapter 2 illustrates, symbolic computation plays an important role in formal proofs, and this kind of difference matters. For instance the `addn` variant helps showing that `(addn n.+1 7)` is different from `0` because by computation Coq would automatically expose a `S` symbol and no natural number of the form `(S ..)` is equal to 0. For a worked out example, see also the proof of `muln_eq0` in section 2.2.2.

1.6 Iterators and mathematical notations

Numbers and sequences of objects are so pervasive in the mathematical discourse that we could hardly omit to present them in such an introductory chapter. On the other hand, the reader may wonder what role programs like `foldr` may play in mathematical sentences.

Actually, in order to formalize the left hand side of the two following formulas:

$$\sum_{i=1}^{n}(i*2-1) = n^2 \qquad \sum_{i=1}^{n} i = \frac{n*(n+1)}{2}$$

and, more generally, in order to make explicit the meaning of the capital notations like $\bigcap_{i=1}^{n} A_i, \prod_{i=1}^{n} u_i, \ldots$, we need to decribe an iteration procedure akin

to the `foldr` program. We illustrate this on the example of the summation symbol (for a finite sum):

```
Fixpoint iota m n := if n is u.+1 then m :: iota m.+1 u else [::].
Notation "\sum_ ( m <= i < n ) F" :=
  (foldr (fun i a => F + a) 0 (iota m (n-m))).
```

The `iota` function generates the list `[:: m; m+1; ...; m+n-1]` of natural numbers corresponding to the range of the summation. Then the notation defines expressions with shape `\sum_ (m <= i < n) F`, with `F` a sub-expression featuring variable `i`, the one used for the index. For instance, once the above notation is granted and for any natural number `(n : nat)`, one may write the Coq expression `(\sum_(1 <= i < n) i)` to represent the sum $\sum_{i=1}^{n-1} i$. The name `i` is really used as a *binder name* here and we may substitute `i` with any other name we may find more convenient, as in `(\sum_(1 <= j < n) j)`, without actually changing the expression[4].

The notation represents an instance of the `foldr` program, iterating a function of type `nat -> nat -> nat`: the two type parameters of `foldr` are set to `nat` in this case. The first explicit argument given to `foldr` is hence a function of type `nat -> nat -> nat`, which is created by *abstracting* the variable `i` in the expression `F`, as mentioned in section 1.1.1. The name of the second argument of the function, called `a` here, should not appear in expression `F` in order for the notation to be meaningful. Crucially, the `foldr` iterator takes as input a *functional argument that is able to represent faithfully any general term.* As a second explicit argument to `foldr`, we also provide the neutral element `0` for the addition: this is the initial value of the iteration, corresponding to an empty index range.

Let us perform a few examples of computations with these iterated sums.

Query	Response
`Eval compute in \sum_( 1 <= i < 5 ) (i * 2 - 1).`	`= 16 : nat`
`Eval compute in \sum_( 1 <= i < 5 ) i.`	`= 10 : nat`

Behind the scenes, iteration happens following the order of the list, as we observed in section 1.5. In the present case the operation we iterate, addition, is commutative, so this order does not impact the final result. But it may not be the case, for example if the iterated operation is the product of matrices.

Defining the meaning of this family of notations using a generic program `foldr`, which manipulates functions, is not only useful for providing notations, but it also facilitates the design of the corpus of properties of these expressions. This corpus comprises lemmas derived from the generic properties of `foldr`, which do not depend on the function iterated. By assuming further properties linking the iterated operation to the initial value, like forming a monoid, we will be able to provide a generic theory of iterated operations in section 5.7.

[4] Although this change results in two syntactically different expressions, they have the same meaning for Coq

1.7 Notations, abbreviations

Throughout this chapter, we have used *notations* to display formal terms in a more readable form, closer to the usual conventions adopted on paper. Without notations, formal terms soon get unreadable for humans for they often expose too low level details in mathematical expressions. For this purpose, the Coq system provides a `Notation` command, and we have mentioned it several times already, relying on the reader's intuition to understand roughly how it works. Its actual behavior is quite complex. With this command, it is possible to declare various kind of notations and to specify the way associativity and their precedence to the parsing engine of Coq. It is also possible to provide some hints for printing, like good breaking points. *Scopes* are groups of notation that go together well, and can be activated or deactivated simultaneously. They are usually associated with a *scope delimiter*, which allows the activation of a scope locally in a sub-expression.

For instance the infix notation that we have used so far for the constant `addn` can be declared as follows:

```
1  Notation "m + n" := (addn m n) (at level 50, left associativity).
```

For infix notations, which are meant to be printed between two arguments of the operator (like addition in `2 + 3`), we advise to always include space around the infix operator, so that notations don't get mixed up with potential notations occurring in the arguments.

Similarly, the comparison relation `leq` on type `nat` comes with an infix notation `<=` which can be defined as:

```
1  Notation "m <= n" := (leq m n) (at level 70, no associativity).
```

where the rightmost annotations, between parentheses, indicate a precedence level and an associativity rule so as to avoid parsing ambiguities. A lower level binds more than a higher level. A comprehensive description of the `Notation` command goes behind the scope of this book; the interested reader shall refer to [24, chapter 12].

A notation can denote an arbitrary expression, and not only to a single constant. Here is for instance the definition of the infix notation `<`, for the strict comparison of two natural numbers: it denotes the composition of the comparison `_ <= _` (which refers to the constant `leq`) with the successor `_.+1` (which refers to the constructor `S`) on its first argument:

```
1  Notation "m < n"  := (m.+1 <= n).
```

Warning

There is no function testing if a natural number is strictly smaller than another one. `(a < b)` is just an alternative syntax for `(a.+1 <= b)`

The converse relation `(n > m)` is defined as a notation for `(m < n)`. However, it is only accepted in input, and is always printed as `(m < n)`, thanks to the following declaration:

```
Notation "n > m"  := (m.+1 <= n) (only parsing).
```

Another frequently used form of notation is called syntactic abbreviation. It simply lets one specify a different name for the same object. For example the `S` constructor of natural numbers can also be accessed by writing `succn`. This is useful if `S` is used in the current context to, say, denote a ring.

```
Notation succn := S.
```

The notation `_.+1` we have been using so far is defined on top of this abbreviation, as:

```
Notation "n .+1" := (succn n) (at level 2, left associativity): nat_scope.
```

The `Locate` command can be used to reveal the actual term represented by a notation. In order to understand the meaning of an unknown notation like for instance `(a <= b <= c)`, one can use the `Locate` command to uncover the symbols it involves.

```
Locate "<=".
```

```
Notation                                    Scope
"m <= n <= p" := andb (leq m n) (leq n p)    : nat_scope
"m <= n < p" := andb (leq m n) (leq (S n) p) : nat_scope
"m <= n" := leq m n                          : nat_scope
```

The only difficulty in using `Locate` comes from the fact that one has to provide a complete *symbol*. A symbol is composed of one or more non-blank characters and its first character is necessarily a non-alphanumerical one. The converse is not true: a sequence of characters is recognized as a symbol only if it is used in a previously declared notation. For example `3.+1.+1` is parsed as a number followed by two occurrences of the `.+1` symbol, even if `.+1.+1` could, in principle, be a single symbol.

Moreover substrings of a symbol are not necessarily symbols. As a consequence `Locate "="` does not find notations like the ones above, since `<=` is a different symbol even if it contains `=` as a substring. For the same reason `Locate ".+"` returns an empty answer since `.+` is not a (complete) symbol.

We mention here some notational conventions adopted throughout the Mathematical Components library.

- Concepts that are typically denoted with a letter, like $N(G)$ for the normalizer of the group G, are represented by symbols beginning with `'` as in `'N(G)`, where `'N` is a symbol.
- At the time of writing, notations for numerical constants are specially handled by the system. The algebraic part of the library overrides specific cases, like binding `1` and `0` to ring elements.
- Postfix notations begin with `.`, as in `.+1` and `.-group` to let one write `(p.-group G)`. There is one exception for the postfix notation of the factorial, which starts with `` ` ``, as in `m`!`.
- Taking inspiration from LaTeX some symbols begin with `\`, like `\in`, `\matrix`, `\sum`, ...
- Arguments typically written as subscripts appear after a symbol which ends with an underscore like `'N_` in `'N_G(H)`.
- N-ary notations begin with `[` followed by the symbol of the operation being repeated, as in `[&& true, false & false]`.
- Bracket notations are also used for operations building data, as in `[seq .. | .. ]`.
- Curly braces notations like `{poly R}` are used for data types with parameters.
- Curly braces are also used to write localized statements such as `{in A, injective f}`, which means that the restriction of `f` to `A` is injective.

Each file in the Mathematical Components library comes with a header documenting all concepts and associated notations it provides.

1.8 Exercises

Exercise 1. *The triple data type*

Define the triple data type such that the following notation applies. Also define the projections.

```
Notation "( a , b , c )" := (mk_triple a b c).
Notation "p .1" := (fst p) (at level 2).
Notation "p .2" := (snd p) (at level 2).
Notation "p .3" := (thrd p) (at level 2).
Eval compute in (4, 5, 8).1.
Eval compute in (true, false, 1).2.
Eval compute in (2, true, false).3.
```

```
     = 4 : nat
     = false : bool
     = false : bool
```

Remark that the triple data type has to be polymorphic in order to be applicable to each of these combinations of natural numbers and booleans.

Exercise 2. *Addition with iteration*

Define a program computing the sum of two natural numbers using `iter`.

Exercise 3. *Multiplication with iteration*

Define a program computing the product of two natural numbers using `iter`.

Exercise 4. *Find the n-th element*

Define a program taking a default value, a list and a natural number n. The program should return the n-th element of the list (counting from 0) if n is smaller than the size of the list. It should return the default value otherwise.

```
Eval compute in
  nth 99 [:: 3; 7; 11; 22] 2.
Eval compute in
  nth 99 [:: 3; 7; 11; 22] 7.
```

```
     = 11
     : nat
     = 99
     : nat
```

Exercise 5. *List reversal*

Define the program `rev` that reverses the order of the elements of a list.

```
Eval compute in
  rev [:: 1; 2; 3].
```

```
  = [:: 3; 2; 1]
  : seq nat
```

Exercise 6. * *List flattening*

Define the program `flatten` that takes a list of lists and returns their concatenation. Don't write a recursive function; just reuse the concatenation function and one of the higher-order iterators seen so far.

```
Eval compute in
  flatten [:: [:: 1; 2; 3]; [:: 4; 5] ].
```

```
= [:: 1; 2; 3; 4; 5]
: seq nat
```

Exercise 7. ** *All words of size n*

Define the `all_words` program that takes in input a length `n` and a sequence of symbols `alphabet`. The program has to generate a list of all words (i.e., lists of symbols) of size `n` using the symbols from `alphabet`.

```
Eval compute in
  all_words 2 [:: 1; 2; 3].


```

```
= [:: [:: 1; 1]; [:: 1; 2]; [:: 1; 3];
      [:: 2; 1]; [:: 2; 2]; [:: 2; 3];
      [:: 3; 1]; [:: 3; 2]; [:: 3; 3]]
: seq (seq nat)
```

1.8.1 Solutions

Answer of Exercise 1

```
Inductive triple (A B C : Type) := mk_triple (a : A) (b : B) (c : C).
Notation "( a , b , c )" := (mk_triple a b c).
Definition fst  A B C (p : triple A B C) := let: (a, _, _) := p in a.
Definition snd  A B C (p : triple A B C) := let: (_, b, _) := p in b.
Definition thrd A B C (p : triple A B C) := let: (_, _, c) := p in c.
```

Answer of Exercise 2

```
Definition addn n1 n2 := iter n1 S n2.
```

Answer of Exercise 3

```
Definition muln n1 n2 := iter n1 (addn n2) 0.
```

Answer of Exercise 4

```
Fixpoint nth T (def : T) (s : seq T) n :=
  if s is x :: tl then if n is u.+1 then nth def tl u else x else def.
```

Answer of Exercise 5

```
Fixpoint catrev T (s1 s2 : seq T) :=
  if s1 is x :: xs then catrev xs (x :: s2) else s2.

Definition rev T (s : seq T) := catrev s [::].
```

Answer of Exercise 6

```
Definition flatten T (s : seq (seq T)) := foldr cat [::] s.
```

Answer of Exercise 7

```
Definition all_words n T (alphabet : seq T) :=
  let prepend x wl := [seq x :: w | w <- wl] in
  let extend wl := flatten [seq prepend x wl | x <- alphabet] in
  iter n extend [:: [::] ].
```

Another possible solution [5]:

```
Fixpoint all_words_alt (T : Type) (n : nat) (xs : seq T) :=
  if n is m.+1 then
    flatten [seq [seq x :: w | w <- (all_words_alt m xs)] | x <- xs]
  else nil :: nil.
```

[5] suggested by Darij Grinberg.

Chapter 2

First Steps in Formal Proofs

In this chapter, we explain how to use the Coq system to state and to prove theorems, focusing on simple statements and basic proof commands. In the course of this book, we will see that choosing the right way to state a proposition formally can be a rather delicate matter. For equivalent wordings of one and the same proposition, some can be much simpler to prove, and some can be more convenient to invoke inside the proof of another theorem. This chapter emphasizes the use of computable definitions and equational reasoning whenever possible, an approach that will be developed fully in chapter 4.

2.1 Formal statements

In this section, we illustrate how to state elementary candidate theorems, starting with *identities*.

2.1.1 Ground equalities

Coq provides a binary predicate named `eq` and equipped with the infix notation `=`. This predicate is used to write sentences expressing that two objects are *equal*, like in $2+2 = 4$. Let us start with examples of Coq *ground* equality statements: *ground* means that these statements do not feature parameter variables. For instance $2 + 2 = 4$ is a ground statement, but $(a + b)^2 = a^2 + 2ab + b^2$ has two parameters a and b: it is not ground.

The `Check` command can be used not only to verify the type of some expression, but also to check whether a formal statement is well formed or not:

```
Check 3 = 3.
Check false && true = false.
```

```
3 = 3 : Prop
false && true = false : Prop
```

Let's anatomize the two above examples. Indeed, just like Coq's type system prevents us from applying functions to arguments of a wrong nature, it also enforces a certain nature of well-formedness at the time we enunciate sentences that are candidate theorems. Indeed, formal statements in Coq are themselves *terms* and as such they have a *type* and their subterms obey type constraints. An equality statement in particular is obtained by applying the constant `eq` to two arguments *of the same type*. This application results in a well-formed term of type `Prop`, for *proposition*.

Throughout this book, we will use the word *proposition* for a term of type `Prop`, typically something one wants to prove.

The `About` vernacular command provides information on a constant: its type, the list of arguments of the constant that are implicit, ... For instance we can learn more about the constant `eq`:

The polymorphic equality predicate

```
Locate "=".

About eq.

```

Response

```
"x = y" := eq x y

eq : ∀ A : Type, A -> A -> Prop
Argument A is implicit ...
```

The constant `eq` is a *predicate*, i.e., a function that outputs a proposition. The equality predicate is polymorphic: exactly like we have seen in the previous chapter, the `forall` quantifier (represented by the ∀ symbol in Coq's output, at least as long as the symbol is available) is used to make the (implicit) parameter `A` range over types. Both examples `3 = 3` and `false && true = false` thus use the *same* equality constant, but with different values (respectively, `nat` and `bool`) for the type parameter `A`. Since the first argument of `eq` is implicit, it is not part of the infix notation and its value is not provided by the user. This value can indeed be inferred from the type of the two sides of the identity: `(3 = 3)` unfolds to `(eq _ 3 3)`, and the missing value must be `nat`, the type of `3`. Similarly, `(false && true = false)` unfolds to `(eq _ (false && true) false)` and the missing value is `bool`, the common type of `false` and `(false && true)`.

As the Coq system checks the well-typedness of statements, the two sides of a well-formed equality should have the same type:

```
Check 3 = [:: 3].

```

```
Error: The term "[:: 3]" has type "seq nat"
 while it is expected to have type "nat".
```

Yet it does not check the provability of the statement!

```
Check 3 = 4.
```

```
3 = 4 : Prop
```

In order to establish that a certain equality holds, the user should first announce that she is going to prove a sentence, using a special command like `Lemma`. This command has several variants `Theorem`, `Remark`, `Corollary`,... which are all synonyms for what matters here. A `Lemma` keyword is followed by the name

chosen for the lemma and then by the statement itself. Command `Lemma` and its siblings are in fact a variant of the `Definition` syntax we used in chapter 1: everything we mentioned about it also applies here. The `Proof` command marks the beginning of the proof text, which ends either with `Qed` or `Admitted`. After the command `Proof` is executed, the system displays the current state of the formal proof in a dedicated window.

Stating a lemma and starting its proof

```
1 Lemma my_first_lemma : 3 = 3.
2 Proof.
3 (* your proof text *)
4
```

Response after line 2: proof state

```
1 subgoal

  ======================
   3 = 3
```

Indeed, Coq is now in its so-called *proof mode*: we can execute new commands to construct a proof and inspect the current state of a proof in progress, but some other commands, like opening sections, are no longer available. At any stage of the proof construction, Coq displays the current state of the (sub)proof currently pending: a list of named hypotheses forms the current context and is printed on top of the horizontal bar (empty here), whereas the statement of the current goal (the conjecture to be proved) is below the bar.

We will explain how to proceed with such a proof in section 2.2.1. For now, let us just admit this result, using the `Admitted` command.

```
1 Lemma my_first_lemma : 3 = 3.
2 Proof.
3 Admitted.
```

Although we have not (yet) provided a proof for this lemma, a new definition has been added to our environment:

```
1 About my_first_lemma.
```

```
my_first_lemma : 3 = 3
```

In the rest of the chapter, we will often omit the `Admitted` proof terminator, and simply reproduce the statement of some lemmas in order to discuss their formulation.

2.1.2 Identities

Ground equalities are a very special case of mathematical statements called *identities*. An *identity* is an equality relation $A = B$ that holds regardless of the values that are substituted for the variables in A and B. Let us state for instance the identity expressing the associativity of the addition operation on natural numbers:

```
1 Lemma addnA (m n k : nat) : m + (n + k) = m + n + k.
```

Note that in the statement of `addnA`, the right hand side does not feature any parentheses but should be read `((m + n) + k)`: This is due to the left-associativity of the infix `+` notation, which was prescribed back when this notation was defined (see section 1.7). Command `Lemma`, just like `Definition`, allows for dropping the

type annotations of parameters if these types can be inferred from the statement itself:

```
Lemma addnA n m k : m + (n + k) = m + n + k.
```

Boolean identities play a central role in the Mathematical Components library. They state equalities between boolean expressions (possibly with parameters). For instance, the `orbT` statement expresses that `true` is right absorbing for the boolean disjunction operation `orb`. Recall from section 1.2.1 that `orb` is equipped with the `||` infix notation:

```
Lemma orbT b : b || true = true.
```

More precisely, lemma `orbT` expresses that the truth table of the boolean formula `(b || true)` coincides with the (constant) one of `true`: otherwise said, that the two propositional formulas are equivalent, or that `(b || true)` is a propositional tautology. Below, we provide some other examples of such propositional equivalences stated as boolean identities.

```
Lemma orbA b1 b2 b3 : b1 || (b2 || b3) = b1 || b2 || b3.
Lemma implybE a b : (a ==> b) = ~~ a || b.
Lemma negb_and (a b : bool) : ~~ (a && b) = ~~ a || ~~ b.
```

Proving these identities (once we have learned to provide proofs) is left as an exercise (see exercise 8).

2.1.3 From boolean predicates to formal statements

A *boolean predicate* means a function to `bool`. A boolean predicate can indeed be seen as an effective truth table, which can be used to form a proposition in a systematic way by equating its result to `true`. More generally, boolean identities are equality statements in type `bool`, which may involve arbitrary boolean predicates and boolean connectives. They can feature variables of an arbitrary type, not only of type `bool`.

For instance, the boolean comparison function `(leq : nat -> nat -> bool)` is a boolean binary predicate on natural numbers. Lemma `leq0n` is a proposition asserting that a certain comparison always holds, by stating that the truth value of the boolean `(0 <= n)` is `true`, whatever term of type `nat` is substituted for the parameter `n`.

Stating that a boolean predicate holds

```
Lemma leq0n (n : nat) : 0 <= n = true.
```

The Mathematical Components library makes an extensive use of boolean predicates, and of the associated propositions. For the sake of readability, the default behavior of the Mathematical Components library is to omit the "`.. = true`" part in these boolean identities. Coq is actually able to insert automatically and silently this missing piece whenever it fits and is non-ambiguous, thanks to its *coercion* mechanism. We postpone further explanation of this mechanism to section 4.4, but from now on, we stop displaying the `.. = true`

parts of the statement that are silently inserted this way. For instance, lemma `leq0n` is displayed as:

```
Lemma leq0n (n : nat) : 0 <= n.
```

As a general fact, boolean identities express that two boolean statements are equivalent. We already encountered special cases of such equivalence with propositional tautologies in section 2.1.2. Here are a few more examples involving boolean predicates on natural numbers that we have defined in chapter 1: the boolean equality `==` and its negation `!=`, the order relation `<` and its large version `<=`, and the divisibility predicate `%|`, with `(a %| b)` meaning `a` *divides* `b`. Note that we omit the type of the parameters; they are all of type `nat`, as enforced by the type of the operators involved in the statements:

```
Lemma eqn_leq m n : (m == n) = (m <= n) && (n <= m).
Lemma neq_ltn m n : (m != n) = (m < n) || (n < m).
Lemma leqn0 n : (n <= 0) = (n == 0).
Lemma dvdn1 d : (d %| 1) = (d == 1).
Lemma odd_mul m n : odd (m * n) = odd m && odd n.
```

2.1.4 Conditional statements

In the previous sections, we have seen statements of unconditional identities: either equalities between ground terms, or identities that hold for *any* value of their parameters. A property that holds only when its parameters satisfy some condition is stated using an *implication*, and the Coq syntax for this connective is "`->`". For instance:

Implication

```
Lemma leq_pmull m n : n > 0 -> m <= n * m.
Lemma odd_gt0 n : odd n -> n > 0.
```

This arrow `->` is the same as the one we have used in chapter 1 in order to represent function types. This is no accident, but we postpone further comments on the meaning of this arrow to section 3.1. For now let us only stress that `->` is right-associative, and therefore a succession of arrows expresses a conjunction of conditions:

```
Lemma dvdn_mul d1 d2 m1 m2 : d1 %| m1 -> d2 %| m2 -> d1 * d2 %| m1 * m2.
```

Replacing conjunctions of hypotheses by a succession of implications is akin to replacing a function taking a tuple of arguments by a function with a functional type ("currying"), as described in section 1.1.2.

2.2 Formal proofs

We shall now explain how to turn a well-formed statement into a machine-checked theorem. Let us come back to our first example, that we left unproved:

```
1 Lemma my_first_lemma : 3 = 3.
2 Proof.
3 Admitted.
```

In the Coq system, the user builds a formal proof by providing, interactively, instructions to the Coq system that describe the gradual construction of the proof she has in mind. This list of instructions is called a *proof script*, and the instructions it is made of are called proof commands, or more traditionally *tactics*. The language of tactic we use is called Ssreflect.

Scheme of a complete proof

```
1 Lemma my_first_lemma : 3 = 3.
2 Proof.
3 (* your finished proof script comes here *)
4 Qed.
```

Once the proof is complete, we can replace the `Admitted` command by the `Qed` one. This command calls the proof checker part of the Coq system, which validates a posteriori that the formal proof that has been built so far is actually a complete and correct proof of the statement, here `3 = 3`.

In this section, we will review different kinds of proof steps and the corresponding tactics.

2.2.1 Proofs by computation

Here is now a proof script that validates the statement $3 = 3$.

Reflexivity of equality

```
1 Lemma my_first_lemma : 3 = 3.
2 Proof. by [].
```

Proof finished

```
No more subgoals.
```

Indeed, this statement holds trivially, because the two sides of the equality are syntactically the same. The tactic "`by []`" is the command that implements this nature of *trivial* proof step. The proof command `by` typically prefixes another tactic (or a list thereof): it is a *tactical*. The `by` prefix checks that the following tactic trivializes the goal. But in our case, no extra work is needed to solve the goal, so we pass an empty list of tactics to the tactical `by`, represented by the empty bracket `[]`.

The system then informs the user that the proof looks complete. We can hence confidently conclude our first proof by the `Qed` command:

```
1 Lemma my_first_lemma : 3 = 3.
2 Proof. by []. Qed.
3 About my_first_lemma.
```

```
No more subgoals.
my_first_lemma is defined
my_first_lemma : 3 = 3
```

Just like when it was `Admitted`, this script results in a new definition being added in our context, which can then be reused in future proofs under the name `my_first_lemma`. Except that this time we have *a machine checked proof* of the statement of `my_first_lemma`. By contrast `Admitted` happily accepts false statements...

What makes the `by []` tactic interesting is that it can be used not only when both sides of an equality coincide syntactically, but also when they are equal *modulo the evaluation of programs* used in the formal sentence to be proved. For instance, let us prove that $2 + 1 = 3$.

```
Lemma my_second_lemma : 2 + 1 = 3.
Proof. by []. Qed.
```

Indeed, this statement holds because the two sides of the equality are the same, once the definition of the `addn` function, hidden behind the infix `+` notation, is unfolded, and once the calculation is performed. In a similar way, we can prove the statement `(0 <= 1)`, or `(odd 5)`, because both expressions *compute* to `true`.

As we have seen in chapter 1, computation is not limited to ground terms; it is really about using the rules of the pattern matching describing the code of the function. For instance the proof of the `addSn` identity:

Reflexivity by symbolic computation

```
Lemma addSn m n : m.+1 + n = (m + n).+1. Proof. by []. Qed.
```

is trivial as well because it is a direct consequence of the definition of the `addn` function: This function is defined by pattern matching, with one of the branches stating exactly this identity. Statements like `(0 + n = n)` or `(0 < n.+1)` can be proved in a similar way, but also `(2 + n = n.+2)`, which requires several steps of computation.

Last, the `by` tactical turns its argument into a *terminating tactic* — and thus `by []` is such a terminating tactic. A tactic is said to be terminating if, whenever it does not solve the goal completely, it fails and stops Coq from processing the proof script. A terminating tactic is colored in red so that the eye can immediately spot that a proof, or more commonly a subproof, ends there.

2.2.2 Case analysis

Let us now consider the tautology `~~ (~~ b) = b`. The "proof by computation" technique of section 2.2.1 fails in this case:

Double negation elimination

```
Lemma negbK (b : bool) : ~~ (~~ b) = b.
Proof. by [].
```

Failing proof script

```
Error: No applicable
tactic.
```

Indeed, proving this identity requires more than a simple unfolding of the definition of `negb`:

```
Definition negb (b : bool) : bool := if b then false else true.
```

One also needs to perform a *case analysis* on the boolean value of the parameter `b` and notice that the two sides coincide in both cases. The tactic `case` implements this action:

Reasoning by cases

```
1  Lemma negbK b : ~~ (~~ b) = b.
2  Proof.
3  case: b.
4
5
6
```

Case analysis

```
2 subgoals

  ============================
   ~~ ~~ true = true
subgoal 2 is:
 ~~ ~~ false = false
```

More precisely, the tactic "`case: b`" indicates that we want to perform a case analysis on term `b`, whose name follows the separator `:`. The Coq system displays the state of the proof after this command: The proof now has two subcases, treated in two parallel branches, one in which the parameter `b` takes the value `true` and one in which the parameter `b` takes the value `false`. More generally, the `case: t` tactic performs a case analysis on (the shape of) a term `t` which should be of an inductive type. As any inhabitant of an inductive type is necessarily built from one of its constructors, this tactic creates as many branches in the proof as the type has constructors, in the order in which they appear in the definition of the type. In our example, the branch for `true` comes first, because this constructor comes first in the definition of type `bool`.

We shall thus provide two distinct pieces of script, one per each subproof to be constructed, starting with the branch associated with the `true` value. In order to signal that we are starting a piece of script for a sub-proof, it is a good practice to indent the corresponding script.[1]

Once the case analysis has substituted a concrete value for the parameter `b`, the proof becomes trivial, in both cases: We are in a similar situation as in the proofs of section 2.2.1 and the tactic `by []` applies successfully:

First trivial case

```
1  Lemma negbK b : ~~ (~~ b) = b.
2  Proof.
3  case: b.
4    by [].
```

Second goal

```
1 subgoal

 ============================
   ~~ ~~ false = false
```

Once the first goal is solved, we have only one subgoal left, and we solve it using the same tactic.

Second trivial case

```
1  Lemma negbK b : ~~ (~~ b) = b.
2  Proof.
3  case: b.
4    by [].
5  by [].
6  Qed.
```

Finished proof

```
No more subgoals.
negbK is defined
```

Yet, as we mentioned earlier, we can also use the `by` tactical as a prefix for

[1] A "sub-proof" means a proof of a sub-goal; i.e., a proof whose completion does not immediately conclude the proof of the lemma. Thus, for example, if the `case:` tactic splits a proof into two goals, then only the first case (i.e., the proof of the first of these goals) counts as a sub-proof. The second case is not a sub-proof (as it concludes the proof of the lemma), and thus is not indented.

any tactic (not just an empty list of tactics), and have the system check that after the `case` tactic, the proof actually becomes trivial, in both branches of the case analysis. This way, the proof script becomes a one-liner:

Double negation elimination: final script

```
Lemma negbK b : ~~ (~~ b) = b.
Proof. by case: b. Qed.
```

Case analysis with naming

The boolean equivalence `leqn0` is another example of statement that cannot be proved by computation only:

```
Lemma leqn0 n : (n <= 0) = (n == 0).
Proof.
```

```
n : nat
======================
  (n <= 0) = (n == 0)
```

Both comparison operations `<=` and `==` are defined by case analysis on their *first* argument, independently of the shape of the second. The proof of `leq0n` thus goes by case analysis on term `(n : nat)`, as it appears as a first argument to both these comparison operators. Remember that the inductive type `nat` is defined as:

```
Inductive nat := O | S (n : nat).
```

with two constructors, `O` which has no argument and `S` which has one (recursive) argument. A case analysis on term `(n : nat)` thus has two branches: one in which `n` is `O` and one in which `n` is `(S k)`, denoted `k.+1`, for some `(k : nat)`. We hence need a variant of the `case` tactic, in order to *name* the parameter `k` that pops up in the second branch as the argument of the `S` constructor of type `nat`:

case with naming

```
Lemma leqn0 n : (n <= 0) = (n == 0).
Proof.
case: n => [| k].


```

```
2 subgoals

======================
  (0 <= 0) = (0 == 0)

subgoal 2 is:
 (k < 0) = (k.+1 == 0)
```

The tactic "`case: n => [|k]`" can be decomposed into two components, separated by the arrow `=>`. The left block "`case: n`" indicates that we perform a case analysis action, on term `(n : nat)`, while the right block "`[|k]`" is an *introduction pattern*. The brackets surround slots separated by vertical pipes, and each slot allows to name the parameters to be introduced in each subgoal created by the case analysis, in order.

As type `nat` has two constructors, the introduction pattern `[|k]` of our case analysis command uses two slots: the last one introduces the name `k` in the second subgoal and the first one is empty. Indeed, in the first subgoal (first branch of the case analysis), `n` is substituted with `0`. In the second one, we can observe that `n` has been substituted with `k.+1`. As hinted in the first chapter,

the term `(k.+1 <= 0)` is displayed as `(k < 0)`.

The first goal can be easily solved by computation, as both sides of the equality evaluate to `true`.

```
Lemma leqn0 n : (n <= 0) = (n == 0).
Proof.
case: n => [| k].
  by [].
```

```
k : nat
=========================
  (k < 0) = (k.+1 == 0)
```

The second and now only remaining goal corresponds to the case when `n` is the successor of `k`. Note that `(k < 0)` is a superseding notation for `(k.+1 <= 0)`, as mentioned in section 1.7. This goal can also be solved by computation, as both sides of the equality evaluate to `false`. The final proof script is hence:

```
Lemma leqn0 n : (n <= 0) = (n == 0).
Proof.  by case: n => [| k].  Qed.
```

We will use a last example of boolean equivalence to introduce more advanced proof techniques, leading to less verbose proof scripts. Remember from chapter 1 that the product of two natural numbers is defined as a function `(muln : nat -> nat -> nat)`. From this definition, we prove that the product of two (natural) numbers is zero if and only if one of the numbers is zero:

Specifying muln: a double case analysis attempt

```
Fixpoint muln (m n : nat) : nat :=
  if m is p.+1 then n + muln p n else 0.

Lemma muln_eq0 m n : (m * n == 0) = (m == 0) || (n == 0).
```

In the case when `m` is zero (whatever value `n` takes), both sides of the equality evaluate to `true`: the left hand side is equal modulo computation to `(0 == 0)`, which itself computes to `true`, and the right hand side is equal modulo computation to `((0 == 0) || (n == 0))`, hence to `(true || (n == 0))` and finally to `true` because the boolean disjunction `(_ || _)` is defined by case analysis on its first argument.

```
Lemma muln_eq0 m n :
  (m * n == 0) = (m == 0) || (n == 0).
Proof.
case: m => [|m].
  by [].
```

One goal left

```
n, m : nat
=========================
(m.+1 * n == 0) =
  (m.+1 == 0) || (n == 0)
```

In this script, we used the name `m` for the argument of the constructor in the second branch of the case analysis. There is no ambiguity here and this proof step reads: either `m` is zero, or it is of the form `m.+1` (for a new `m`).

By default, the successor case is treated in the second subgoal, according to the order of constructors in the definition of type `nat`. If we want to treat it first, we can use the "`; last first`" tactic suffix:

```
Lemma muln_eq0 m n : (m * n == 0) = (m == 0) || (n == 0).
Proof.
case: m => [|m].
  by [].
case: n => [|k]; last first.
```

Two goals left (in reverse order)

```
2 subgoals

m, k : nat
============================
(m.+1 * k.+1 == 0) = (m.+1 == 0) || (k.+1 == 0)

subgoal 2 is:
 (m.+1 * 0 == 0) = (m.+1 == 0) || (0 == 0)
```

Indeed it is a good practice to get rid of the easiest subgoals as early as possible. And here the successor case is such an easy subgoal: when `n` is of the form `k.+1`, it is easy to see that the right hand side of the equality evaluates to `false`, as both arguments of the boolean disjunction do. Now the left hand side evaluates to `false` too: by the definition of `muln`, the term `(m.+1 * k.+1)` evaluates to `(k.+1 + (m * k.+1))`, and by definition of the addition `addn`, this in turn reduces to `(k + (m * k.+1)).+1`. The left hand side term hence is of the form `t.+1 == 0`, where `t` stands for `(k + (m * k.+1))`, and this reduces to `false`. In consequence, the successor branch of the case analysis is trivial by computation.

```
Lemma muln_eq0 m n :
  (m * n == 0) = (m == 0) || (n == 0).
Proof.
case: m => [|m].
  by [].
case: n => [|k]; last first.
  by [].
```

```
1 subgoal

m : nat
==========================
(m.+1 * 0 == 0) =
  (m.+1 == 0) || (0 == 0)
```

This proof script can actually be made more compact and, more importantly, more linear by using extra features of the introduction patterns. It is indeed possible, although optional, to inspect the subgoals created by a case analysis and to solve the trivial ones on the fly, as the `by []` tactic would do, except that in this case no failure happens in the case some, or even all, subgoals remain. For instance in our case, we can add the optional `//` *simplify* switch to the introduction pattern of the first case analysis:

A simplify intro pattern

```
Lemma muln_eq0 m n :
  (m * n == 0) = (m == 0) || (n == 0).
Proof.
case: m => [|m] //.
```

Proof state

```
n, m : nat
==========================
(m.+1 * n == 0) =
  (m.+1 == 0) || (n == 0)
```

Only the first generated subgoal is trivial: Thus, it has been closed and we are left with the second one. Similarly, we can get rid of the second goal produced by the case analysis on `n`:

```
Lemma muln_eq0 m n :
  (m * n == 0) = (m == 0) || (n == 0).
Proof.
case: m => [|m] //.
case: n => [|k] //.
```

```
m : nat
=========================
(m.+1 * 0 == 0) =
  (m.+1 == 0) || (0 == 0)
```

This `//` switch can be used in more general contexts than just this special case of introduction patterns: It can actually punctuate more complex combinations of tactics, avoiding spurious branching in proofs in a similar manner [13, section 5.4].

The last remaining goal cannot be solved by computation. The right hand side evaluates to `true`, as the left argument of the disjunction is `false` (modulo computation) and the right one is `true`. However, we need more than symbolic computation to show that the left hand side is `true` as well: the fact that `0` is a right absorbing element for multiplication indeed requires reasoning by *induction* (see section 2.3.4).

To conclude the proof we need one more proof command, the `rewrite` tactic, that lets us appeal to an already existing lemma.

2.2.3 Rewriting

This section explains how to locally replace certain subterms of a goal with other terms during the course of a formal proof. In other words, we explain how to perform a *rewrite* proof step, thanks to the eponymous `rewrite` tactic. Such a replacement is licit when the original subterm is equal to the final one, up to computation or because of a proved identity. The `rewrite` tactic comes with several options for an accurate specification of the operation to be performed.

Let us start with a simple example and come back to the proof that we left unfinished at the end of the previous section:

```
Lemma muln_eq0 m n :
  (m * n == 0) = (m == 0) || (n == 0).
Proof.
case: m => [|m] //.
case: n => [|k] //.
```

```
m : nat
========================
(m.+1 * 0 == 0) =
  (m.+1 == 0) || (0 == 0)
```

At this stage, if we replace subterm `(m.+1 * 0)` by `0`, the subgoal becomes:

```
(0 == 0) = (m.+1 == 0) || (0 == 0)
```

which is equal modulo computation to `(true = true)`, hence trivial. But since the definition of `muln` proceeded by pattern matching on its *first* argument, `(m.+1 * 0)` does not evaluate symbolically to `0`: This equality holds but requires a proof by induction, as explained in section 2.3.4. For now, let us instead derive `(m.+1 * 0 = 0)` from a lemma. Indeed, the Mathematical Components library provides a systematic review of the properties of the operations it defines. The lemma we need is available in the library as:

```
Lemma muln0 n : n * 0 = 0.
```

As a side remark, being able to find the "right" lemma is of paramount importance for writing modular libraries of formal proofs. See section 2.5 which is dedicated to this topic.

Back to our example, we use the `rewrite` tactic with lemma `muln0`, in order to perform the desired replacement.

First rewrite

```
Lemma muln_eq0 m n :
  (m * n == 0) = (m == 0) || (n == 0).
Proof.
case: m => [|m] //.
case: n => [|k] //.
rewrite muln0.
```

Proof state

```
m : nat
========================
(0 == 0) =
  (m.+1 == 0) || (0 == 0)
```

The `rewrite` tactic uses the `muln0` lemma in the following way: It replaces an instance of the left hand side of this identity with the corresponding instance of the right hand side. The left hand side of `muln0` can be read as a *pattern* `(_ * 0)`, where `_` denotes a wildcard: The identity is valid for any value of its parameter `n`. The tactic automatically finds where in the goal the replacement should take place, by searching for a subterm matching the pattern `(_ * 0)`. In the present case, there is only one such subterm, `(m.+1 * 0)`, for which the parameter (or the wild-card) takes the value `m.+1`. This subterm is hence replaced by `0`, the right hand side of `muln0`, which does not depend on the value of the pattern. We can now conclude the proof script, using the prenex `by` tactical[2]:

```
Lemma muln_eq0 m n : (m * n == 0) = (m == 0) || (n == 0).
Proof.
case: m => [|m] //.
case: n => [|k] //.
by rewrite muln0.
Qed.
```

Arguments to the `rewrite` tactic are typically called *rewrite rules* and can be prefixed by flags tuning the behavior of the tactic.

Rewriting many identities in one go

The boolean identity `muln_eq0` that we just established expresses a logical equivalence that can in turn be used in proofs via the `rewrite` tactic. For instance, let us consider the case of lemma `leq_mul2l`, which provides a necessary and sufficient condition for the comparison `(m * n1 <= m * n2)` to hold:

Another example: product vs. order

```
Lemma leq_mul2l m n1 n2 : (m * n1 <= m * n2) = (m == 0) || (n1 <= n2).
```

The proof goes as follows: The left hand side can equivalently be written as `(m * n1 - m * n2 == 0)`, which factors into `(m * (n1 - n2) == 0)`. But this is

[2] Passing a single tactic to `by` requires no brackets; i.e., we can write `by rewrite muln0` instead of `by [rewrite muln0]`.

equivalent to one of the arguments of the product being zero. And `(n1 - n2 == 0)` means `(n1 <= n2)`.

The first step is performed using the following equation:

```
Lemma leqE m n : (m <= n) = (m - n == 0).
Proof. by []. Qed.
```

The proof of this identity is trivial, as the right hand side is the definition of the `leq` relation, denoted by the `<=` infix notation. Rewriting with this equation turns the left hand side of our goal into a subtraction:

Unfolding leq

```
Lemma leq_mul2l m n1 n2 :
 (m * n1 <= m * n2) = (m == 0) || ...
Proof.
rewrite leqE.
```

Proof state

```
m, n1, n2 : nat
============================
(m * n1 - m * n2 == 0) =
  (m == 0) || (n1 <= n2)
```

The command `rewrite leqE` only affects the first occurrence of `<=`, but we would like to substitute both. In order to rewrite *all* the possible instances of the rule in the goal, we may use a repetition flag, which is `!`:

```
Lemma leq_mul2l m n1 n2 :
 (m * n1 <= m * n2) = (m == 0) || ...
Proof.
rewrite !leqE.
```

```
m, n1, n2 : nat
============================
(m * n1 - m * n2 == 0) =
  (m == 0) || (n1 - n2 == 0)
```

Now the definition of `<=` has been exposed *everywhere* in the goal, i.e., at both its occurrences in the initial goal. We can now factor out `m` on the left, according to the appropriate distributivity property:

```
Lemma mulnBr n m p : n * (m - p) = n * m - n * p.
```

This time we need to perform a right-to-left rewriting of the `mulnBr` lemma (instead of the default left-to-right). The rewriting step first finds in the goal an instance of pattern `(_ * _ - _ * _)`, where the terms matched by the first and the third wildcards coincide. The syntax for right-to-left rewriting consists in prefixing the name of the rewrite rule with a minus `-`:

Rewrite right-to-left

```
Lemma leq_mul2l m n1 n2 :
 (m * n1 <= m * n2) = (m == 0) || ...
Proof.
rewrite !leqE. rewrite -mulnBr.
```

Proof state

```
m, n1, n2 : nat
============================
(m * (n1 - n2) == 0) =
  (m == 0) || (n1 - n2 == 0)
```

Consecutive rewrite steps can be chained as follows:

```
Lemma leq_mul2l m n1 n2 :
 (m * n1 <= m * n2) = (m == 0) || ...
Proof.
rewrite !leqE -mulnBr.
```

```
m, n1, n2 : nat
============================
(m * (n1 - n2) == 0) =
  (m == 0) || (n1 - n2 == 0)
```

The last step of the proof uses lemma `muln_eq0` to align the left and the right hand sides of the identity.

```
Lemma leq_mul2l m n1 n2 :
 (m * n1 <= m * n2) = (m == 0) || ...
Proof.
rewrite !leqE -mulnBr muln_eq0.
```

```
m, n1, n2 : nat
============================
(m == 0) || (n1 - n2 == 0) =
  (m == 0) || (n1 - n2 == 0)
```

The proof can now be completed by prefixing the tactic with the `by` tactical.

We only provided here some hints on the basic features of the `rewrite` tactic. Section 2.4.1 gives more details on the matching algorithm and on the flags supported by `rewrite`. The complete description of the features of this tactic is found in the manual [13].

2.3 Quantifiers

2.3.1 Universal quantification, first examples

Let us compare an example of a function we defined in chapter 1:

```
Definition leq n m := m - n == 0.
```

with an example of a parametric statement we have used in the present chapter:

```
Lemma leqn0 n : (n <= 0) = (n == 0).
```

We recall, as seen in chapter 1, that this concise syntax for defining `leq` stands for:

```
Definition leq := fun (n m : nat) => m - n == 0.
```

and that the type of the constant `leq` is:

```
About leq.
```

```
leq : nat -> nat -> bool
```

The curious reader might already have tested the answer of the `About` command on some parametric lemmas:

Inspecting lemma leqn0

```
About leqn0.
```

Universal quantification

```
leqn0 : ∀ n : nat, (n <= 0) = (n == 0)
```

She has thus observed that Coq's output features a prenex `forall` quantifier. This universal quantifier binds a natural number, and expresses — as expected — that the equation holds for *any* natural number. In fact, the types of the lemmas and theorems with parameters all feature prenex universal quantifiers:

```
About muln_eq0.

```

```
muln_eq0 : ∀ m n : nat,
  (m * n == 0) = (m == 0) || (n == 0)
```

Quantifiers may also occur elsewhere in a statement, and not only in prenex position. In the following example, we use the function `nth`, extracting the

element of a sequence at a given position, which was the object of exercise 4. This statement expresses that two sequences with the same size and whose n-th elements coincide for any n are the same. The second hypothesis, about the elements, is itself a quantified formula:

```
Lemma seq_eq_ext (s1 s2 : seq nat) :
  size s1 = size s2 ->
  (∀ i : nat, nth 0 s1 i = nth 0 s2 i) ->
  s1 = s2.
```

Quantifiers are also allowed to range over functions:

```
Lemma size_map (T1 T2 : Type) :
  ∀ (f : T1 -> T2) (s : seq T1), size (map f s) = size s.
```

Observe that in the above statement of `size_map`, we have used a compact notation for successive universal quantifications: "`∀ (f : T1 -> T2) (s : seq T1), ...`" is syntactic sugar for "`∀ f : T1 -> T2, ∀ s : seq T1, ...`". However, in this case of prenex quantification, we could just as well write:

```
Lemma size_map (T1 T2 : Type) (f : T1 -> T2) (s : seq T1) :
  size (map f s) = size s.
```

as all quantifiers are in prenex positions.

Quantifiers may also occur in the body of definitions, which is useful to define predicates expressing standard properties on objects. For instance, the commutativity property of a binary operator is defined as:

```
Definition commutative (S T : Type) (op : S -> S -> T) :=
  ∀ x y, op x y = op y x.
```

and the lemma stating the commutativity of the `addn` operation is in fact:

```
Lemma addnC : commutative addn.
```

The Mathematical Components library defines several such predicates, which are used as molds in order to state standard properties in a consistent and compact way. We provide below a few examples:

```
Section StandardPredicates.
Variable T : Type.
Implicit Types (op add : T -> T -> T) (R : rel T).
Definition associative op := ∀ x y z, op x (op y z) = op (op x y) z.
Definition left_distributive op add :=
  ∀ x y z, op (add x y) z = add (op x z) (op y z).
Definition left_id e op := ∀ x, op e x = x.
End StandardPredicates.
```

where `(rel T)` is an abbreviation for the type `(T -> T -> bool)`.

Beside the standardization of the statements through these predicates, the Mathematical Components library uses a systematic naming policy for the lemmas that are instances of these predicates. A common suffix `C` is used for commutativity properties like `addnC` or `mulnC`. Such naming conventions are also useful to search the library, as detailed in section 2.5.

Another class of predicates typically describes usual properties of functions; these usually feature quantifiers in their definitions:

```
Section MoreStandardPredicates.
Variables rT aT : Type.
Implicit Types (f : aT -> rT).
Definition injective f := ∀ x1 x2, f x1 = f x2 -> x1 = x2.
Definition cancel f g := ∀ x, g (f x) = x.
Definition pcancel f g := ∀ x, g (f x) = Some x.
End MoreStandardPredicates.
```

The types of these predicates deserve a few comments:

```
About commutative.
```

```
commutative : ∀ S T : Type, (S -> S -> T) ->
    Prop
```

The constant `commutative` has a polymorphic parameter `T`, takes a binary operation as argument and builds a *proposition*. It is hence a polymorphic unary predicate on a certain class of functions, the binary functions with both their arguments and their result having the same type. Just like the polymorphic binary predicate `eq`, the predicate `commutative` can be used to form propositions:

```
Check 3 = 3.
Check (commutative addn).
```

```
3 = 3 : Prop
commutative addn : Prop
```

2.3.2 Organizing proofs with sections

The `Section` mechanism presented in Section 1.4 can be used to factor not only the parameters but also the hypotheses of a corpus of definitions and properties. For instance, the proof of the Chinese Remainder Theorem is stated within such a section. It uses a self-explanatory notation for congruences:

```
Section Chinese.

Variables m1 m2 : nat.
Hypothesis co_m12 : coprime m1 m2.

...

Lemma chinese_remainder x y :
  (x == y %[mod m1 * m2]) = (x == y %[mod m1]) && (x == y %[mod m2]).
Proof.
...
End.

End Chinese.
```

The part of this excerpt up to the "..." corresponds to a mathematical sentence of the form: *In this section, m_1 and m_2 are two coprime natural numbers...*. Within the scope of this section, the parameters `m1` and `m2` are fixed and the hypothesis `co_m12` is assumed to hold. Outside the scope of the section (i.e., after the `End Chinese` command), these variables and the hypotheses are *generalized*, so that the statement of `chinese_remainder` becomes:

```
Lemma chinese_remainder m1 m2 (co_m12 : coprime m1 m2) x y :
  (x == y %[mod m1 * m2]) = (x == y %[mod m1]) && (x == y %[mod m2]).
```

In general, when a section ends, the types of the constants and the statements of the lemmas change to include those section variables and hypotheses that are actually used in their definitions or proofs.

2.3.3 Using lemmas in proofs

In order to use a known lemma, one should provide the values of its parameters that specify the instance relevant to the current proof. Fortunately, Coq can assist its user in describing these values, and the `apply:` tactic, like the `rewrite` one in section 2.4, finds the appropriate instance by comparing the lemma to the current goal:

```
Lemma leqnn n : n <= n. Proof. Admitted.

Lemma example a b : a + b <= a + b.
Proof. by apply: leqnn. Qed.
```

The lookup performed by the `apply` tactic works up to computation:

```
Lemma example a b : a.+1 + b <= (a + b).+1.
Proof. by apply: leqnn. Qed.
```

In order to save the effort of explicitly mentioning trivial steps in the proof script, we can extend the power of the `by` terminator to make it aware of some lemmas available in the library. The `Hint Resolve` command is used to tag these lemmas, as in:

```
(* This line belongs to the file where the lemma leqnn is stated and
     proved.*)
Hint Resolve leqnn.
Lemma example a b : a + b <= a + b.
Proof. by [].  Qed.
```

Observe that the goal is now closed without a mention of `leqnn`, although it has been used by the system to conclude the proof.

In order to illustrate more proof techniques related to the use of lemmas inside proofs, let us scrutinize a formal proof that a prime number which divides $m! + 1$ for a certain integer m has to be greater than m. This lemma is a key step in a proof that there are infinitely many primes, which will be studied in section 4.3.1. The proof of the lemma goes by contraposition: If p is a prime number smaller than m, then it divides $m!$ and thus it cannot divide $m! + 1$ as it does not divide 1. We first state this lemma as follows:

```
Lemma example m p : prime p -> p %| m `! + 1 -> m < p.
```

where `p %| m`! stands for "`p` divides the factorial of `m`".

The first step of our formal proof will be to give a name to the hypothesis `(prime p)`, which means that we add it to the current context of the goal. The

dedicated tactic for this naming step is `move=>` followed by the name given to the hypothesis, because the hypothesis *moves* from under the bar to above the bar:

```
Lemma example m p : prime p ->
  p %| m `! + 1 -> m < p.
Proof.
move=> prime_p.
```

```
m, p : nat
prime_p : prime p
====================
p %| m`! + 1 -> m < p
```

The second step of the proof is to transform the current goal into its contrapositive. This means that we use the lemma

```
Lemma contraLR (c b : bool) :  (~~ c -> ~~ b) -> (b -> c).
```

which describes (one direction of) the contraposition law (namely, that an implication between booleans can be derived from its contraposition). The `apply: contraLR` tactic finds the appropriate values of the premise and conclusion and instantiates the law, leaving us with the task of proving that `p` is not a divisor of `(m `! + 1)` under the assumption that `p` is not greater than `m`:

```
Lemma example m p : prime p ->
  p %| m `! + 1 -> m < p.
Proof.
move=> prime_p.
apply: contraLR.
```

```
m, p : nat
prime_p : prime p
====================
 ~~ (m < p) -> ~~ (p %| m`! + 1)
```

More precisely, the values chosen by the tactic for the two parameters `c, b` of lemma `contraLR` are `(m < p)` and `(p %| m `! + 1)`. They have been found by comparing the statement to be proved with the conclusion `(b -> c)` of the statement of the lemma `contraLR`. The new statement of the goal is the corresponding instance of the premise `(~~ c -> ~~ b)` of lemma `contraLR`.

The next steps in our formal proof are to improve the shape of the hypothesis `~~ (m < p)` (using `rewrite -leqNgt`) and to give it a name (using `move=> leq_p_m`):

```
Lemma example m p : prime p ->
  p %| m `! + 1 -> m < p.
Proof.
move=> prime_p.
apply: contraLR.
rewrite -leqNgt.
move=> leq_p_m.
```

```
m, p : nat
prime_p : prime p
leq_p_m :  p <= m
====================
 ~~ (p %| m`! + 1)
```

And the next step uses the following lemma:

```
Lemma dvdn_addr m d n : d %| m -> (d %| m + n) = (d %| n).
```

This is a conditional equivalence, expressed as a conditional identity. We can replace our current goal with `~~ (p %| 1)` by rewriting it using (the appropriate instance of) this identity. This operation will open an extra goal requiring a proof of (the corresponding instance of) the side condition `p %| m`!`.

```
Lemma example m p : prime p ->
  p %| m `! + 1 -> m < p.
Proof.
move=> prime_p.
apply: contraLR.
move=> leq_p_m.
rewrite dvdn_addr.
```

```
2 subgoals

  m, p : nat
  prime_p : prime p
  leq_p_m : p <= m
  ============================
   ~~ (p %| 1)

subgoal 2 is:
 p %| m`!
```

Observe the second goal at the bottom of the buffer, which displays the statement of the side condition to be proved later. The context of this subgoal is omitted but we do not really need to see it: We know that statement `p %| m`!` holds because `p <= m` and because we can combine the following lemmas:

```
Lemma dvdn_fact m n : 0 < m -> m <= n -> m %| n`!.
Lemma prime_gt0 p : prime p -> 0 < p.
```

The first goal is also easy to solve, using the following basic facts:

```
Lemma gtnNdvd n d : 0 < n -> n < d -> (d %| n) = false.
Lemma prime_gt1 p : prime p -> 1 < p.
```

Finally, the resulting script would be:

```
Lemma example m p : prime p -> p %| m `! + 1 -> m < p.
Proof.
move=> prime_p.
apply: contraLR.
rewrite -leqNgt.
move=> leq_p_m.
rewrite dvdn_addr.
  rewrite gtnNdvd.
    by [].  (* ~~ false *)
    by [].  (* 0 < 1 *)
  by apply: prime_gt1.  (* 1 < p *)
apply: dvdn_fact.
  rewrite prime_gt0.  (* 0 < p *)
  rewrite leqNgt.
  by [].  (* true && ~~ (m < p) *)
by [].   (* prime p *)
Qed.
```

For brevity, we record the goal solved by a tactic in a comment after this tactic.[3]

We shall improve this script in two steps. First, we take advantage of `rewrite` simplification flags. It is quite common for an equation to be conditional, hence for `rewrite` to generate side conditions. We have already suggested that a good practice consists in proving the easy side conditions as soon as possible. Here, the first two side conditions are indeed trivial, and, just as with the introduction

[3] Some comments might still be helpful. After `apply: dvdn_fact.`, our goal became `0 < p <= m`, which is really an abbreviation for `(0 < p) && (p <= m)`. The two subsequent `rewrite` commands rewrote `(0 < p)` as a `true` and rewrote `(p <= m)` as `~~ (m < p)` (which is an assumption); after that, the goal fell prey to the `by []` tactic.

patterns of the `case` tactic, we can use a simplification switch `//` to prove them. We also combine on the same line the first three steps, using the semicolon[4]. The proof script (up to the end of the proof of the first goal) then looks like this:

```
Lemma example m p : prime p -> p %| m `! + 1 -> m < p.
Proof.
move=> prime_p; apply: contraLR; rewrite -leqNgt; move=> leq_p_m.
rewrite dvdn_addr.
  rewrite gtnNdvd //.
  by apply: prime_gt1.  (* 1 < p *)
```

A careful comparison of the conclusions of `gtnNdvd` and `prime_gt1` reveals that they are both rewriting rules. While the former features an explicit "`.. = false`", in the latter one the "`.. = true`" part is hidden, but is there. This means both lemmas can be used as identities.

Advice

All boolean statements can be rewritten as if they were regular identities. The result is that the matched term is replaced with `true`.

Rewriting with `prime_gt1` leaves open the trivial goal `true` (i.e., `(true = true)`), and the side condition `(prime p)`. Both are trivial, hence solved by prefixing the line with `by`.

```
Lemma example m p : prime p -> p %| m `! + 1 -> m < p.
Proof.
move=> prime_p; apply: contraLR; rewrite -leqNgt; move=> leq_p_m.
rewrite dvdn_addr.
  by rewrite gtnNdvd // prime_gt1.
```

The same considerations hold for the last goal.

```
Lemma example m p : prime p -> p %| m `! + 1 -> m < p.
Proof.
move=> prime_p; apply: contraLR; rewrite -leqNgt; move=> leq_p_m.
rewrite dvdn_addr.
  by rewrite gtnNdvd // prime_gt1.
by rewrite dvdn_fact // prime_gt0.
Qed.
```

[4] From this example, one might take away the wrong impression that a semicolon is synonymous to a dot. In general, it is not. The semicolon is actually a tactical (like `by`) that allows chaining tactics. A tactic of the form `t1; t2` (where `t1` and `t2` are two tactics) amounts to first applying `t1`, and then applying `t2` to *each* of the subgoals spawned by the application of `t1`. When `t1` spawns exactly one subgoal, this is indeed equivalent to `t1. t2`. (Note that the tactic `t2` has to work in each of these subgoals in order for the notation to compile!) A slight variation of this syntax is `t0; [t1 | t2 | ... | tn]` for any tactics `t0`, `t1`, ..., `tn`; this amounts to applying tactic `t0`, and then applying tactic `t1` to the first subgoal spawned, `t2` to the second, and so on.

To sum up, both `apply:` and `rewrite` are able to find the right instance of a quantified lemma and to generate subgoals for its eventual premises. Hypotheses can be named using `move=>`.

The proof script given above for `example m p` can be further reduced in size. One simple improvement is to replace the chained tactic `rewrite -leqNgt; move=> leq_p_m` by the equivalent `rewrite -leqNgt => leq_p_m`. Indeed, as we will see later (in subsection 3.2.2), the `move` tactic does nothing; it is the `=>` that is responsible for the naming of the hypothesis.

In section 2.4.1, we shall describe some further ways to shrink the proof script.

2.3.4 Proofs by induction

Let us take the well known induction principle for Peano's natural numbers and let us formalize it in the language of Coq. It reads: let $\mathcal{P}$ be a property of natural numbers; if $\mathcal{P}$ holds on 0 and if, for each natural number n, the property $\mathcal{P}$ holds on $n+1$ as soon as it holds on n, then $\mathcal{P}$ holds for any n. Induction is typically regarded as a schema, where the variable $\mathcal{P}$ stands for any property we could think about.

In the language of Coq, it is possible to use a quantification to bind the parameter $\mathcal{P}$ in the schema, akin to the universal quantification of polymorphic parameters in data types like `seq`. Induction principles, instead of being "schemas", are regular lemmas with a prenex quantification on predicates:

```
About nat_ind.
```

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

Here `P` is quantified exactly as `n` is, but its type is a bit more complex and deserves an explanation. As we have seen in the first chapter, the `->` denotes the type of functions; hence `P` is a function from `nat` to `Prop`. Recall that `Prop` is the type of *propositions*, i.e., something we may want to prove. In the light of that, `P` is a function producing a proposition out of a natural number. For example, the property of being an odd prime can be written as follows:

```
(fun n : nat => (odd n && prime n) = true)
```

Indeed, if we take such function as the value for `P`, the first premise of `nat_ind` becomes

P 0

```
(fun n : nat => (odd n && prime n) =
     true) 0
```

Equivalent by computation

```
odd 0 && prime 0 = true
```

Remark the similarity between the function argument to `foldr` that is used to describe the general term of an iterated sum in section 1.6 and the predicate `P` here used to describe a general property.

Coq defines an induction principle for every inductive type the user defines, with standard names formed by adding a suffix `_ind` to the name of the type. The statement of a generated induction principle is shaped by the

structure of the definition of the inductive type. For example, if we define an inductive type `bintree` (representing unlabeled planar binary trees) as follows:

```
Inductive bintree := leaf | graft (u v : bintree).
```

then we automatically are granted an induction principle called `bintree_ind`:

```
About bintree_ind.
```

```
bintree_ind :
∀ P : bintree -> Prop,
P leaf ->
(∀ u : bintree, P u -> ∀ v : bintree, P v -> P
     (graft u v)) ->
∀ b : bintree, P b

Argument P is implicit
```

For another example, here is the induction principle for sequences (although denoted by `list_ind` rather than `seq_ind`, as `seq` is defined merely as a synonym for `list`), which has some similarities with the one for natural numbers:

```
About list_ind.
```

```
list_ind : ∀ (A : Type) (P : seq A -> Prop),
  P [::] ->
  (∀ (a : A) (l : seq A), P l -> P (a :: l)) ->
  ∀ l : seq A, P l
```

To sum up: reasoning by induction on a term `t` means finding the induction lemma associated to the type of `t` *and* synthesizing the right predicate `P`. The `elim:` tactic has these two functionalities, while `apply:` does not. Thus, while both `elim:` and `apply:` can be used to formalize a proof by induction, the user would have to explicitly specify both `t` and `P` in order to make use of the `apply:` tactic, whereas the `elim:` tactic does the job of determining these parameters itself: The induction principle to be used is guessed from the type of the argument of the tactic. Let us illustrate on an example how the value of the parameter `P` is guessed by the `elim:` tactic and let us prove by induction on `m` that `0` is neutral on the right of `addn`.

```
Lemma addn0 m : m + 0 = m.
Proof.
elim: m => [ // |m IHm].
```

```
m : nat
IHm : m + 0 = m
=======================
m.+1 + 0 = m.+1
```

The `elim:` tactic is used here with an introduction pattern similar to the one we used for `case:`. It has two slots, because of the

two constructors of type `nat` (corresponding naturally to what is commonly called the "induction base" and the "induction step"), and in the second branch we give a name not only to the argument `m` of the successor, but also to the induction hypothesis. We also used the `//` switch to deal with the

base case because if `m` is `0`, both sides evaluate to zero. The value of the parameter `P` synthesized by `elim:` for us is `(fun n : nat => n + 0 = n)`. It has been obtained by *abstracting* the term `m` in the goal (see section 1.1.1). The proof concludes by using lemma `addSn` to pull the `.+1` out of the sum, so that the induction hypothesis `IHm` can be used for rewriting.

Unfortunately proofs by induction do not always run so smooth. To our aid the `elim:` tactic provides two additional services. The first one is to let one *generalize* the goal. It is typically needed when the goal mentions a recursive function that uses an accumulator: its value is going to change during recursive calls; hence the induction hypothesis must be general.

Another service provided by `elim:` is specifying an alternative induction principle. For example, one may reason by induction on a list starting from its end, using the following induction principle:

```
Lemma last_ind A (P : list A -> Prop) :
  P [::] -> (∀ s x, P s -> P (rcons s x)) -> ∀ s, P s.
```

where `rcons` is the operation of concatenating a sequence with an element, as in `(s ++ [::x])`.

For example `last_ind` can be used to relate the `foldr` and `foldl` iterators as follows:

```
Fixpoint foldl T R (f : R -> T -> R) z s :=
  if s is x :: s' then foldl f (f z x) s' else z.

Lemma foldl_rev T R f (z : R) (s : seq T) :
  foldl f z (rev s) = foldr (fun x z => f z x) z s .
```

The proof uses the following lemmas:

Tools

```
Lemma cats1 T s (z : T) : s ++ [:: z] = rcons s z.
Lemma foldr_cat T R f (z0 : R) (s1 s2 : seq T) :
  foldr f z0 (s1 ++ s2) = foldr f (foldr f z0 s2) s1.
Lemma rev_rcons T s (x : T) : rev (rcons s x) = x :: rev s.
```

The complete proof script follows:

```
Lemma foldl_rev T A f (z : A) (s : seq T) :
  foldl f z (rev s) = foldr (fun x z => f z x) z s .
Proof.
elim/last_ind: s z => [|s x IHs] z //.
by rewrite -cats1 foldr_cat -IHs cats1 rev_rcons.
Qed.
```

Here "`elim/last_ind: s z`" performs the induction using the `last_ind` lemma on `s` after having generalized the initial value of the accumulator `z`. The resulting value for `P` hence features a quantification on `z`:

```
(fun s => ∀ z, foldl f z (rev s) = foldr (fun x z => f z x) z s)
```

The "`z`" that comes *after* "`elim/last_ind: s z => [|s x IHs]`" deserves further explanation. Had we stopped at "`elim/last_ind: s z => [|s x IHs] //`", our goal would look like this:

```
∀ z : A,
  foldl f z (rev (rcons s x)) = foldr (fun x0 : T => f^~ x0) z (rcons s x)
```

(ignore the "`f^~`", which is coming out of the blue here, but is merely the standard abbreviation for `fun x z => f z x` in the Mathematical Components library). The additional "`z`" after the "`=> [|s x IHs]`" is responsible for pulling the "`∀ z : A`" out of the goal again, back into the heap of variables and assumptions. (See subsection 3.2.2 for more detail about this kind of operations.)

We have thus generalized `z` only to pull it out of the goal again soon after. It might appear as if this was a useless detour. However, it was not. Thanks to the generalization, the induction hypothesis `IHs` states:

```
1  IHs : ∀ z : A, foldl f z (rev s) = foldr (fun x z => f z x) z s
```

which is more general than what we had obtained if we had not generalized `z` beforehand. The quantification on `z` is crucial since the goal in the induction step, just before we use `IHs`, is the following one:

```
1  foldl f z (rev (s ++ [:: x])) =
2    foldr (fun y w => f w y) (foldr (fun y w => f w y) z [:: x]) s
```

The instance of the induction hypothesis that we need is one where `z` takes the value `(foldr (fun y w => f w y) z [:: x])`. The generalization of `z` gave us the freedom to substitute a different value for `z`.

2.4 Rewrite, a Swiss army knife

Approximately one third of the proof scripts in the Mathematical Components library is made of invocations of the `rewrite` tactic. This proof command provides many features we cannot extensively cover here. We just sketch a very common idiom involving conditional rewrite rules and we mention the `RHS` pattern for the casual reader. The interested reader can find more about the pattern language in section 2.4.1 or in the dedicated chapter of the Ssreflect language user manual [13].

We have seen before that applying the `rewrite` tactic can create side conditions which themselves need to be proven (i.e., they are sub-goals). (For example, we obtained the side condition `p %| m`!` when we used the `rewrite dvdn_addr` tactic in proving `example m p`, since the lemma `dvdn_addr` had a `d %| m` condition.) These side conditions (by default) become the second, third and higher subgoals in a proof script, so their proofs are usually postponed to after the first sub-goal (the main one) is proven. This is not always desirable; therefore, it is helpful to have a way to prove side conditions right away, on the same line where they arise. One way to do this (which we have already seen in action) is using the simplification item `//`. When this does not suffice, one can invoke another rewrite rule using the optional iterator `?`. A rule prefixed by `?` is applied to all goals zero-or-more times. For example, recall our proof of `example m p`:

```
Lemma example m p : prime p -> p %| m `! + 1 -> m < p.
Proof.
move=> prime_p; apply: contraLR; rewrite -leqNgt => leq_p_m.
rewrite dvdn_addr.
  by rewrite gtnNdvd // prime_gt1. (* ~~ (p %| 1) *)
by rewrite dvdn_fact // prime_gt0. (* p %| m`! *)
Qed.
```

The side condition `p %| m`!` spawned by `rewrite dvdn_addr` was proven in the last line of the script. Instead, we could have solved it right away by rewriting using `?dvdn_fact ?prime_gt0`. In fact, optionally rewriting with `dvdn_fact` on all goals affects only the side condition, since the main goal mentions no "divides" predicate. The same holds for `prime_gt0`. The resulting proof script is:

```
Lemma example m p : prime p -> p %| m `! + 1 -> m < p.
Proof.
move=> prime_p; apply: contraLR; rewrite -leqNgt => leq_p_m.
rewrite dvdn_addr ?dvdn_fact ?prime_gt0 //.
by rewrite gtnNdvd // prime_gt1.
Qed.
```

The prefix `!` works similarly to `?`, but instead of applying the rewrite rule zero-or-more times, it applies it one-or-more times.

Another functionality offered by `rewrite` is the possibility to focus the search for the term to be replaced by providing a context. For example, the most frequent context is `RHS` (for Right Hand Side) and is used to force `rewrite` to operate only on the right hand side of an equational goal.

```
Lemma silly_example n : n + 0 = (n + 0) + 0.
Proof. by rewrite [in RHS]addn0. Qed.
```

The last rewrite flag worth mentioning is the `/=` simplification flag. It performs computations in the goal to obtain a "simpler" form.

```
Lemma simplify_me : size [:: true] = 1.
Proof.
rewrite /=.
```

```
  ============================
  1 = 1
```

The `/=` flag simply invokes the Coq standard `simpl` tactic. Whilst being handy, `simpl` tends to oversimplify expressions, hence we advise using it with care. In section 4.3.3 we propose a less risky alternative. The sequence "`// /=`" can be collapsed into `//=`.

Another version of the `rewrite` tactic allows *unfolding* a definition – i.e., replacing an object by its definition. For example, the lemma `leqE` that we used in the proof of `leq_mul2l` back in section 2.2.3 does not exist in the library, and there is no name associated to this equation. It is simply the definition of `leq`, and we actually do not need to state a lemma in order to relate the name of a definition, like `leq`, to its body `fun n m => n - m == 0`. This unfolding operation can be performed by calling the `rewrite` tactic, prefixing the name of the object with `/`, as in `rewrite /leq`. Unfolding a definition is indeed not a deductive

operation but an instance of computation, as made more precise in chapter 3. Our proof of `leq_mul2l` thus takes the form

```
Lemma leq_mul2l m n1 n2 :
 (m * n1 <= m * n2) = (m == 0) || (n1 <= n2).
Proof.
by rewrite /leq -mulnBr muln_eq0.
Qed.
```

Notice that `rewrite /leq` unfolded the definition of `leq` both in the expression `(m * n1 <= m * n2)` and in the expression `(n1 <= n2)`. In general, each appearance of the object is being unfolded. Sometimes it is desirable to unfold only some of them; see [13, §7.1] for various ways to fine-tune the tactic for such purposes. One option is to tell Coq what instances of the object are to be unfolded. For example, had we used `rewrite /(leq n1 n2)` instead of `rewrite /leq`, we would only have gotten `(n1 <= n2)` unfolded.

A less frequently used variant of the `rewrite` tactic is the `-/` prefix; it allows *folding* a definition, i.e., the reverse of unfolding. (For instance, `rewrite /(leq n1 n2)` could be undone by `rewrite -/(leq n1 n2)`.)

2.4.1 Rewrite contextual patterns (⋆)

The example `leq_mul2l` illustrates how the `rewrite` tactic, provided a rewrite rule like `mulnBr` or `muln_eq0`, is able to identify a subterm in the goal to be substituted. The usability of the tactic crucially relies on an appropriate combination of automation and control. The user should be able to predict which subterm will be substituted and to drive the tactic if needed, with enough control options, but not too much verbosity. A key ingredient of the `rewrite` tactic is hence the *matching* algorithm that elects this subterm from the arguments provided to the tactic. Let us provide some insights on the power and on the limitations of this algorithm, as well as on the control primitives that can drive it.

First, remember that our first attempt, using the simple `rewrite leqE` command, only affected the left hand side of the initial goal because of the behavior of this matching algorithm. Indeed, the matching algorithm traverses the entire goal left-to-right, looking for the first subterm matching pattern `(_ <= _)`, and hence picks the subterm `(m * n1 <= m * n2)`. Now suppose we want to pick the other instance of a subterm matching this pattern in the goal. We can use the command `rewrite [n1 <= _]leqE`: the pattern given by the user overrides the one inferred from the rewrite rule and is used to select the subterm to be rewritten. In this case, term `(m * n1 <= m * n2)` is ruled out because the first argument of `<=`, namely `(m * n1)`, does not match the first argument `n1` required in the user-given pattern. Therefore, `rewrite` picks the term `(n1 <= n2)`, in the right hand side.

User provided pattern

```
Lemma leq_mul2l m n1 n2 :
(m * n1 <= m * n2) = (m == 0) || (n1 <= n2).
Proof.
rewrite [n1 <= _]leqE.
```

Proof state

```
m, n1, n2 : nat
============================
(m * n1 <= m * n2) =
(m == 0) || (n1 - n2 == 0)
```

The same result could have been achieved by explicitly specifying the parameters in the matching subterm to be rewritten: viz., by using `rewrite (leqE n1 n2)` instead of `rewrite leqE`.[5]

Another way of driving the matching algorithm is by providing a *context*, restricting the part of the goal to be explored. For instance, in this case, the instance we want to pick is on the right hand side of the identity to be proved. We can implement this specification using the pattern `[in RHS]`:

RHS pattern

```
Lemma leq_mul2l m n1 n2 :
(m * n1 <= m * n2) = (m == 0) || (n1 <= n2).
Proof.
rewrite [in RHS]leqE.
```

Proof state

```
m, n1, n2 : nat
============================
(m * n1 <= m * n2) =
(m == 0) || (n1 - n2 == 0)
```

More generally, one can provide context patterns like `[in X in T]` where `X` is a variable name, bound in `T`. For instance pattern `[in RHS]` is just syntactic sugar for the context pattern `[in X in _ = X]`. We invite the interested reader to check the reference manual [13, section 8] for more variants of patterns and for a more precise description of the different phases in the matching algorithm used by this tactic.

As we have said, the lemma `leqE` does not in fact exist in the library, and instead is just the definition of `leq`. However if we try to omit the first `rewrite !leqE` command, then the next one, namely `rewrite -mulnBr`, fails:

```
Lemma leq_mul2l m n1 n2 :
(m * n1 <= m * n2) = (m == 0) || (n1 <= n2).
Proof.
rewrite -mulnBr.
```

```
Error: The RHS of mulnBr
  (_ * _ - _ * _)
does not match any subterm
of the goal
```

This indicates in particular that, although the term `(m * n1 <= m * n2)` is equal up to computation to the term `(m * n1 - m * n2 == 0)`, the matching algorithm is not able to see it. This is due to the compromise that has been chosen, between predictability and cleverness. Indeed the algorithm looks for a verbatim occurrence of the head symbol of the pattern: in this case it hence looks for an occurrence of `(_ - _)`, which is not found. As a consequence, we need

[5] Sometimes, an even more precise specification is needed. Namely, it can happen that the term `(n1 <= n2)` appears twice in the goal: for example, when the goal is `(n1 <= n2) || (n1 <= n2)`. In this case, we would need `rewrite {1}(leqE n1 n2)` to rewrite only the first appearance of the term, and `rewrite {2}(leqE n1 n2)` to rewrite only the second. In general, the `{k}` prefix (for a positive integer `k`) has the effect that the tactic is being applied in the k-th of the possible positions where it is applicable; for instance, in the proof of `silly_example`, we could have used `rewrite {2}(addn0 n)` instead of `rewrite [in RHS]addn0`. Care must be taken when the `{k}` prefix is applied without fully specifying the parameters, e.g., in the form `rewrite {1}leqE`; the behavior of Coq in this situation is counterintuitive (it starts by finding one subterm of the goal matching `leqE`, then infers its parameters, and then rewrites the k-th among the appearances of the subterm with these parameters, rather than the k-th of all appearances).

an explicit step in the proof script in order to expose the subtraction before being able to rewrite right to left with `mulnBr`. However if we tackle the proof in reverse, starting from the right hand side, the first `-muln_eq0` step will succeed:

```
Lemma leq_mul2l m n1 n2 :
(m * n1 <= m * n2) = (m == 0) || (n1 <= n2).
Proof.
rewrite -[_ || _]muln_eq0.
```

```
m, n1, n2 : nat
============================
(m * n1 <= m * n2) =
  (m * (n1 - n2) == 0)
```

Indeed, the `[_ || _]` pattern identifies term `(m == 0) || (n1 <= n2)`, as their head symbols coincide. Now that we have selected a subterm, the `rewrite` tactic is able to identify it with the term `(m == 0) || (n1 - n2 == 0)`, itself an instance of the right hand side of `muln_eq0`. Indeed, while matching only sees syntactic occurrences of the head symbols of patterns, it is able to compare the other parts of the pattern up to symbolic computation. Note that the `[_ || _]` pattern is redundant here; there is no other location in the goal where the right hand side of `muln_eq0` could appear.

Patterns can not only be used in combination with a rewriting rule, but also with a simplification step `/=` or an unfolding step like `/leq`. For example:

Focused unfold

```
Lemma leq_mul2l m n1 n2 :
(m * n1 <= m * n2) = (m == 0) || (n1 <= n2).
Proof.
rewrite [in LHS]/leq.
```

Proof state

```
m, n1, n2 : nat
============================
(m * n1 - m * n2 == 0) =
  (m == 0) || (n1 <= n2)
```

One can also re-fold a definition, but in such a case one has to specify, at least partially, its folded form.

Refold

```
Lemma leq_mul2l_modified m n1 n2 :
  (m * n1 - m * n2 == 0) =
    (m == 0) || (n1 <= n2).
Proof.
rewrite -/(leq _ _).
```

Proof state

```
m, n1, n2 : nat
============================
(m * n1 <= m * n2) =
  (m == 0) || (n1 <= n2)
```

More generally, the `rewrite` tactic can be used to replace a certain subterm of the goal by another one, which is equal to the former modulo computation:

Replacing computationally equal terms

```
Lemma leq_mul2l m n1 n2 :
(m * n1 <= m * n2) = (m == 0) || (n1 <= n2).
Proof.
rewrite -[n1]/(0 + n1).
```

Proof state

```
m, n1, n2 : nat
============================
(m * (0 + n1) <= m * n2) =
(m == 0) || (0 + n1 <= n2)
```

Last, an equation local to the proof context, like an induction hypothesis, can be disposed of after using it by prefixing its name with `{}`. For example `rewrite -{}IHn` rewrites with `IHn` right to left and drops `IHn` from the context.

2.5 Searching the library

Finding the name of the "right" lemma in a library that contains thousands of them may be quite a challenge. In spite of their digital nature, formal libraries are not so easy to browse and the state of the art of search tools for formal libraries is far from being as advanced as what exists for instance for the world wide web.

In order to help the users find their needle in the haystack, the Mathematical Components library follows uniform naming policies, and the Ssreflect proof language provides a `Search` command which displays lists of items filtered using patterns, like `(_ * _ + _)` or `(addn _ _)`, and substrings of the names, like `"rev" "cons"`.

2.5.1 Search by pattern

The `Search` command takes a list of filters and prints the lemmas that do match all the criteria.

The *first* pattern provided is special, since it is required to match the conclusion of a lemma, while all other patterns can match anywhere.

For example "`Search (odd _)`" only prints one lemma:

```
dvdn_odd  ∀ m n : nat, m %| n -> odd n -> odd m
```

Indeed the conclusion matches the pattern. Note that one is not forced to use wildcards; `odd` alone is a perfectly valid pattern. Many more lemmas are found by leaving the conclusion unspecified, as in "`Search _ odd`".

If we require the lemma to be an equation, as in "`Search eq odd`", we find the following two lemmas (among many other things):

```
dvdn2  ∀ n : nat, (2 %| n) = ~~ odd n
coprime2n  ∀ n : nat, coprime 2 n = odd n
```

If we want to rule out all lemmas about coprimality we can refine the search by writing "`Search eq odd -coprime`".

2.5.2 Search by name

Being acquainted to the naming policy followed by the Mathematical Components library provides one of the more effective ways of finding lemmas in the loaded libraries. The name `my_first_lemma` we chose in section 2.1.1 is a very bad name, as it gives no insight about what the lemma says. Most of the time, we refrain from naming lemmas with numbers, as is typically done in standard mathematical texts. Finding an appropriate name for a lemma can be a delicate task. It should convey as much information as possible, while striving to remain short and handy. In particular, bureaucratic lemmas that are frequently used but represent no deep mathematical step should have a short name: this way they are both easy to type and easy to disregard when skimming through a proof script.

Partial names can be used as filters by the `Search` command. For example `Search "C"` prints, among other things, `addnC` and `mulnC`, the commutativity properties of addition and multiplication. Multiple strings can be specified, for example `Search "1" "muln"`. This time we find `muln1` but also `muln_eq1`, the equation saying that the product of two natural numbers is 1 if and only if they are both 1.

Here are the general principles governing the names of lemmas in the Mathematical Components library:

- **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 historical names, 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**
  - If the conclusion of a lemma is a predicate or an equality for a predicate, then that predicate is a suffix of the lemma name, like in `addn_eq0` or `rev_uniq`.
  - If the conclusion of a lemma is a standard property such as `\char`, `<|`, etc.[6]: the property should be indicated by a suffix (like `_char`, `_normal`, etc), so the lemma name should start with 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

[6]These examples are taken from libraries in the Mathematical Component distribution.

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, addition
  - `E` : definition elimination (often conversion lemmas)
  - `F` : boolean false, finite type variant (as in `canF_eq`), or group functor
  - `G` : group argument
  - `I` : set intersection, injectivity for binary operators
  - `J` : group conjugation
  - `K` : cancellation lemmas
  - `L` : left hand side (as in `canLR`)
  - `M` : group multiplication
  - `N` : boolean negation, additive opposite
  - `P` : characteristic properties (often reflection lemmas)
  - `R` : group commutator, or right hand side (as in `canRL`)
  - `S` : subset argument, or integer successor
  - `T` : boolean truth and Type-wide sets
  - `U` : set union
  - `V` : group or ring multiplicative inverse
  - `W` : weakening
  - `X` : exponentiation, or set Cartesian product
  - `Y` : group join
  - `Z` : module/vector space scaling

2.6 Exercises

Exercise 8. *Truth tables*

Prove the following boolean tautologies:

```
Lemma orbA b1 b2 b3 : b1 || (b2 || b3) = b1 || b2 || b3.
Lemma implybE a b : (a ==> b) = ~~ a || b.
Lemma negb_and (a b : bool) : ~~ (a && b) = ~~ a || ~~ b.
```

Exercise 9. *Rewriting*

Prove the following lemma by rewriting:

```
Lemma subn_sqr m n : m ^ 2 - n ^ 2 = (m - n) * (m + n).
```

The `(_ ^ _)` notation is attached to the `expn` function.

Exercise 10. * *Induction*

Prove the following lemma by induction:

```
Lemma odd_exp m n : odd (m ^ n) = (n == 0) || odd m.
```

Recall that a local equation can be disposed of (after use) by prefixing its name with `{}`.

Exercise 11. ** *Multiple induction*

Prove the following lemma by induction.

```
Definition all_words n T (alphabet : seq T) :=
  let prepend x wl := [seq x :: w | w <- wl] in
  let extend wl := flatten [seq prepend x wl | x <- alphabet] in
  iter n extend [:: [::] ].
Lemma size_all_words n T (alphabet : seq T) :
  size (all_words n alphabet) = size alphabet ^ n.
```

(where `all_words` is as defined in the solution of exercise 7). This requires two inductions: first on the length of words, then on the alphabet. In this last case, a non-trivial sub expression has to be generalized just before starting the induction.

2.6.1 Solutions

Answer of Exercise 8

```
Lemma orbA b1 b2 b3 : b1 || (b2 || b3) = b1 || b2 || b3.
Proof. by case: b1; case: b2; case: b3. Qed.
Lemma implybE a b : (a ==> b) = ~~ a || b.
Proof. by case: a; case: b. Qed.
Lemma negb_and (a b : bool) : ~~ (a && b) = ~~ a || ~~ b.
Proof. by case: a; case: b. Qed.
```

Answer of Exercise 9

```
Lemma subn_sqr m n : m ^ 2 - n ^ 2 = (m - n) * (m + n).
Proof. by rewrite mulnBl !mulnDr addnC [m * _]mulnC subnDl !mulnn. Qed.
```

Answer of Exercise 10

```
Lemma odd_exp m n : odd (m ^ n) = (n == 0) || odd m.
Proof.
elim: n => // n IHn.
by rewrite expnS odd_mul {}IHn orKb.
Qed.
```

Answer of Exercise 11

```
Definition all_words n T (alphabet : seq T) :=
  let prepend x wl := [seq x :: w | w <- wl] in
  let extend wl := flatten [seq prepend x wl | x <- alphabet] in
  iter n extend [:: [::] ].

Lemma size_all_words n T (alphabet : seq T) :
  size (all_words n alphabet) = size alphabet ^ n.
Proof.
elim: n => [|n IHn]; first by rewrite expn0.
rewrite expnS -{}IHn [in LHS]/all_words iterS -/(all_words _ _).
elim: alphabet (all_words _ _) => //= w ws IHws aw.
by rewrite size_cat IHws size_map mulSn.
Qed.
```

The second elim is a bit too much for ch2

Chapter 3

Type Theory

The authors made the deliberate choice to postpone the presentation of the mathematical foundations of the Coq system and the formal definition of its language. Instead, the previous chapters have insisted on the definition of (typed) programs and on the way these programs can be used to describe computable functions and decidable predicates. This take on calculations is indeed both at the core of the type theory underlying Coq and one of the crucial ingredients to the methodology at the base of the Mathematical Components library. The present chapter is devoted to a more in-depth presentation of these mathematical foundations, although an exhaustive description shall remain out of the scope of the present book. For more comprehensive presentations of type theory for formalized mathematics, the reader shall refer to more specialized references like [17], or the shorter survey in the Handbook of Automated Reasoning [2, Volume 2, chapter 18].

3.1 Terms, types, proofs

The Coq proof assistant provides a formal language to describe mathematical statements and proofs, called *Gallina* and based on a type theory called the *Calculus of Inductive Constructions* (CIC) [8, 9]. In this section, we provide some hints on the main features of this formalism. The Calculus of Inductive Constructions is described formally in chapter 4 of the reference manual of Coq [24]. The reference book on homotopy type theory [25] describes a very close formalism.

3.1.1 Propositions as types, proofs as programs

There exist several flavors of *type theory* that can be used as a foundational language for mathematics, alternative to the flavors of set theory usually cited

as the formal language of reference [4]. The main difference between a type-theoretic framework and a set-theoretic one is the status of the deductive system used to represent reasoning in a formal way. Informally speaking, a set-theoretic framework has two stages. First order logic provides the former layer: it describes the language of logical sentences and how these statements can be combined and proved. This language is then used in the second layer, to formulate the axioms of the particular theory of interest. On the contrary a type-theoretic framework is monolithic and the same language of types is used in a uniform way to describe mathematical objects, statements and proofs. *Logical statements* are identified with some *types* and their *proofs* with *terms*, or programs, having this type. Thus proving a statement consists in fact in constructing a term of the corresponding type. In set theory, the rules which govern the construction of grammatically correct statements are rather loose, and the goal of the game is to construct a proof for a given proposition, using first-order logic to combine the axioms of the theory. The analogue of the meta-statement that a given proposition has a proof is the meta-statement that a given term has a given type. Such a meta-statement is called a *typing judgment*. It is written:

$$\Gamma \vdash t : T$$

which means: *in the context* Γ*, the term* t *has the type* T. Typing judgments are justified from typing rules, and these justifications form trees similar to the proof trees of natural deduction. Before getting more precise about what contexts, terms and types are, let us emphasize a couple of important differences with set theory here: first, the typing judgment features explicit expressions for the term and the type, whereas the proof of a statement in first-order logic is seldom mentioned. Second, the term t shall represent an arbitrary object of the mathematical discourse, like a number or a group, and not only a proof.

3.1.2 First types, terms and computations

So every expression we have manipulated so far is a *term* of the Calculus of Inductive Constructions, and any term of this formalism has a type in the context in which it occurs.

We put aside inductive types until section 3.1.3, and define inductively what the terms of the formalism are. We suppose that we

have a countable infinite set of symbols available for names, so that it is always possible to exhibit a name distinct from a given arbitrary finite collection of names. The simplest, atomic, terms are the *variables*, and the *sorts* `Prop` and $\texttt{Type}_i$, for $i \in \mathbb{N}$[1]. We shall also use constants (for instance to introduce definitions). Constants and variables are represented using the pre-existing set of names. If `x` is a variable, and `T` and `U` are two terms, then `∀ x : T, U` is also a term. In fact this term will be used as a type, and it represents a family of types `U` indexed by elements of type `T`. Importantly, if `x` does not occur in the term `U` as a free variable, then this term is usually denoted `T -> U`: it is used for

[1] Coq has another sort `Set` but we do not use it here.

the type of functions having their argument of type `T` and their results of type `U`. For instance, there is no difference between `(nat -> bool)` and `(∀ x : nat, bool)`, but the notation. In the case where `x` does occur in term `U`, the term `∀ x : T, U` is called a *dependent type*. If `x` is a variable and `T`, `u`

are terms, then `(fun x:T => u)` is a term, which represents a function

with argument `x` and body `u`. If `t` and `u` are terms then `(t u)` is also a term, which represents the application of (function) `t` to (its argument) `u`. Finally `let x := t in u` is a term if `t, u` are terms and `x` is a variable.

A typing judgment $\Gamma \vdash t : T$ relates a context Γ and two terms t and T. As its name suggests, the context logs all the facts assumed at the moment when the term occurs. It can be empty but otherwise contains typed variables and typed definitions. A typed variable is a pair of a name and a type: for instance, after executing the command `Variable n : nat.`, the context contains the pair (`n : nat`). A typed definition is a triple of a name, a term, and a type: for instance, after executing the command `Definition a : T := t.`, the context contains the triple (`a : T := t`). We have not yet defined what a type is, because it is a special case of term, defined using typing judgments. Actually, well-formed contexts and typing judgments are defined inductively and simultaneously.

Typing judgments with an empty context express the axiom rules of the formalism, like for instance the types of sorts:

$$\vdash \texttt{Prop} : \texttt{Type}_1 \qquad \vdash \texttt{Type}_i : \texttt{Type}_{i+1}.$$

From now on and as the default display mode of Coq does, we will leave the index of the sorts $\texttt{Type}_i$ implicit and just write `Type`, as these indices do not play a significant role in what is presented in this book. A *type* is a term which can be typed by a sort in a given context:

$$\Gamma \vdash \texttt{T} : \texttt{Prop} \quad \text{or} \quad \Gamma \vdash \texttt{T} : \texttt{Type}.$$

Sorts are thus particular types and they are types of types. A *well-formed context* is either an empty context, or the extension of an existing context with a pair `(x : T)`, where `x` is a fresh name and `T` is a type. As expected, a well-formed context provides a type to the variables it contains:

$$\text{if } \Gamma \text{ is well-formed and } \texttt{(x : T)} \in \Gamma \text{ then } \Gamma \vdash \texttt{x} : \texttt{T}.$$

Contexts not only store global assumptions: they are also used to forge more complex typing judgments, for more complex types. For instance, the following rule explains how to construct non-atomic types:

$$\text{if} \quad \Gamma \vdash T : \texttt{Type} \quad \text{and} \quad \Gamma, x : T \vdash U : \texttt{Type} \quad \text{then} \quad \Gamma \vdash \forall x : T, U : \texttt{Type}.$$

The term $\forall x : T, U : \texttt{Type}$ is the type of functions, as described by the rule:

$$\text{if } \Gamma \vdash \forall x : T, U : \texttt{Type} \text{ and } \Gamma, x : T \vdash t : U \text{ then } \Gamma \vdash \texttt{fun x => t} : \forall x : T, U$$

and the application of a function to an argument is well-typed only if their types agree:

$$\text{if } \Gamma \vdash u : \forall x : T, U \text{ and } \Gamma \vdash t : T \text{ then } \Gamma \vdash u\ t : U[t/x]$$

where $U[t/x]$ denotes the term obtained by substituting the variable x by the term t in the term U.

Consider the simple tautology $A \to A$. The implication symbol $\to$ can be understood as the logical symbol of implication but also as the symbol we used to represent the type of functions, in the non-dependent case. The term `(fun x : A => x)` represents the identity function for terms of type A and is a program of type $A \to A$. If we consider that A is a proposition, and that terms of type A are proofs of A, this function is the identity function on proofs of A. More generally, the logical rules of implicative and universal statements can be decorated with terms, so that proofs of implications can be seen as functions transforming an arbitrary proof of the premise into a proof of the conclusion. The introduction rules respectively for the implication and the universal quantification can be written:

$$\frac{\begin{array}{c}A\\ \vdots \\ B\end{array}}{A \to B} \to_I \qquad \frac{B}{\forall x, B} \forall_I \ (x \text{ fresh})$$

This presentation, in the style of natural deduction, means that in order to prove $A \to B$ one can prove B under the assumption A, and in order to prove $\forall x, B$ one can prove B for a fresh variable x. Let us now annotate these rules with terms whose typing rules coincide with these proof rules:

$$\frac{\begin{array}{c}\texttt{x} : A\\ \vdots \\ \texttt{b} : B\end{array}}{\texttt{(fun x : A => b)} : A \to B} \to_I \qquad \frac{\texttt{b} : B}{\texttt{(fun x : A => b)} : \forall x, B} \forall_I$$

Actually, the term `(fun .. => ..)` serves as a proof for both rules and in both cases the term `b` is a proof of B. The only difference is that x is allowed to occur in B only in the second rule.

We now play the same game with the corresponding elimination rules:

$$\frac{A \to B \qquad A}{B} \to_E \qquad \frac{\forall x, B}{B[t/x]} \forall_E \ (t \text{ is a term})$$

This time function application serves as a proof in both cases:

$$\frac{\texttt{f} : A \to B \qquad \texttt{t} : A}{\texttt{(f t)} : B} \to_E \qquad \frac{\texttt{f} : \forall x : A, B \qquad \texttt{t} : A}{\texttt{(f t)} : B[t/x]} \forall_E$$

Universal quantification is not only useful to form types that play the role of propositions. It can come handy to form data types too. In the second part of this book (section 6.8) we shall see how the matrix data type and the type of its multiplication function benefit from it.

```
matrix : Type -> nat -> nat -> Type.
mulmx : ∀ R : Type, ∀ m n p : nat,
  matrix R m n -> matrix R n p -> matrix R m p.
```

This time the data type of matrices exposes the size, and matrix multiplication can be given a type that rules out incompatible matrices and also describes the size of the resulting matrix in terms of the size of the input ones.[2]

This is in fact a classic example of what is called a *dependent type*: a data type depending on data, two natural numbers here.

Note that in this formalism, quantification can range on the proofs of a given statement, just like it can range on numbers. In this way, a statement may express a property shared by the proofs of a given statement. Such capability finds a rare, but crucial use in Mathematical Components that we discuss in chapter 6: All the proofs of a given equality statement between two booleans are indistinguishable.

```
Lemma bool_irrelevance (P Q : bool) : ∀ e1 e2 : P = Q, e1 = e2.
```

The proofs-as-programs correspondence has a visible impact in the proofs part of the Mathematical Components library. In particular, quantified lemmas, being programs, can be instantiated by simply passing arguments to them. Exactly as one can pass `3` to `addn` and obtain `(addn 3)`, the function adding three, one can "pass" `3` to the lemma `addnC` and obtain a proof of the statement `(∀ y, 3 + y = y + 3)`. Remark that the argument passed to `addnC` shows up in the type of the resulting term `(addnC 3)`: The type of the `addnC` program depends on the value the program is applied to. That is the difference between the *dependent function space* (∀) and the standard function space (→).

Advice

Lemma names can be used as functions, and you can pass arguments to them. For example, `(addnC 3)` is a proof that `(∀ y, 3 + y = y + 3)`, and `(prime_gt0 p_pr)` is a proof that `(0 < p)` whenever `(p_pr : prime p)`.

We refer the reader to the reference manual of Coq [24] for the other rules of the system, which are variants of the ones we presented or which express subtleties of the type system that are out of the scope of the present book, like the difference between the sorts `Prop` and `Type`.

We conclude this section by explaining a last rule of CIC, called the *conversion rule*, which describes the status of computation in this framework. Computation is modeled by rewrite rules explaining how to apply functions to their argument. For instance, the so-called β-reduction rule rewrites an application

[2] To be precise, `mulmx` also needs to take in input operations to add and multiply elements of `R`. Such detail plays no role in the current discussion.

of the form (`fun x => t`)u into $t[u/x]$: the formal argument x is substituted by the actual argument u in the body t of the function. A similar computation rule models the computation of a term of the shape (`let x := u in t`) into $t[u/x]$. Two terms t_1 and t_2 that are equal modulo computation rules are said to be *convertible*, and these terms are indistinguishable to the type system: If T_1 and T_2 are two convertible types and if a term t has type T_1 in the context Γ, then t also has type T_2 in the context Γ. This is the feature of the formalism that we have used in section 2.2.1: The proofs of the statements `2 + 1 = 3` and `3 = 3` are the same, because the terms `2 + 1 = 3` and `3 = 3` are convertible. In chapter 1 and 2 we used boolean programs to express predicates and connectives exactly to take advantage of convertibility: Also the compound statement `(2 != 7 && prime 7)` is convertible to `true`. Finally, as illustrated in section 1.5, computation is not limited to terms without variables: The term `(isT : true)` is a valid proof of `(0 < n.+1)`, as well as a proof of `(0 != p.+1)`.

One can also benefit from convertibility whenever one applies a lemma; otherwise said, whenever the $\to_E$ and $\forall_E$ rules apply. These rules check that the argument has the "right type", i.e. the type of the premise or of the bound variable respectively. At the cost of being more verbose we could have made explicit in, say, the typing for $\to_E$ that types are compared modulo computation as follows:

$$\frac{\texttt{f} : A \to B \qquad \texttt{a} : A' \qquad A \equiv A'}{\texttt{(f a)} : B} \to_E$$

where $A \equiv A'$ denotes that A and A' are convertible.

3.1.3 Inductive types

The formalism described in section 3.1.2 is extended by the possibility of introducing *inductive definitions* [9, 19]. We only provide a very brief overview of this subtle feature and again refer the reader to the reference manual for a precise definition. In chapter 1 we have used several examples of inductive data types like the data type `nat` and its constructors `O` and `S`, or the data type `option`, or various others. An inductive definition simultaneously introduces several new objects into the context: a new type, in a given sort, and new terms for the constructors, with their types. For instance, the command:

```
1  Inductive nat : Type := O : nat | S (n : nat).
```

introduces a new type `nat : Type` and two new terms `(O : nat)` and `(S : nat -> nat)`. Constructors are functions, possibly of zero arguments like `O`, and the codomain of a constructor of the inductive type `T` is always `T`. An inductive type may occur as the type of certain arguments of its constructors, like in the case of `S`. The only way to construct an inhabitant of an inductive type is to apply a constructor to sufficiently many arguments. Constructors are by definition injective functions. Moreover, two distinct constructors construct distinct terms; this is why a function with an argument of an inductive type can be described by pattern matching on this argument. For instance, in chapter 1, we have defined the predecessor function by:

```
Definition predn n := if n is p.+1 then p else n.
```

where `if ... then ... else ...` is a notation for the special case of pattern matching with only two branches and one pattern. The definition of the terms of the formalism is in fact extended with the `match ... with ... end` construction described in section 1.2.2. A special reduction rule expresses that pattern matching a term which features a certain constructor in head position reduces to the term in the corresponding branch of the case analysis.

The definition of the terms of CIC also includes so-called *guarded fixpoints*, which represent functions with a recursive definition. We have used these fixpoints in chapter 1, for instance when defining the addition of two natural numbers as:

```
Fixpoint addn n m :=
  match n with
  | 0 => m
  | p.+1 => (addn p m).+1
  end.
```

Note that guarded fixpoints always terminate, as a non-terminating term would allow proofs of absurdity (see section 3.2.3).

3.1.4 More connectives

With inductive definitions it is possible to describe more data structures than the mere functions described in section 3.1.2. These data structures and their typing rules are used to model logical connectives, like functions were used to model the proofs of implications and universal quantifications.

For instance, the introduction and elimination rule of the conjunction connective are:

$$\frac{A \qquad B}{A \wedge B} \wedge_I \qquad\qquad \frac{A \wedge B}{A} \wedge_E \text{ (left)}$$

The first rule reads: to prove $A \wedge B$ one needs to prove both A and B. The second states that one proves A whenever one is able to prove the stronger statement $A \wedge B$.

In Coq this connective is modeled by the following inductive definition:

```
Inductive and (A B : Prop) : Prop := conj (pa : A) (pb : B).
Notation "A /\ B" := (and A B).
```

Remark that the "data" type `and` is tagged as `Prop`, i.e., we declare the intention to use it as a logical connective rather than a data type. The single constructor `conj` takes two arguments: a proof of `A` and a proof of `B`. Moreover `and` is polymorphic: `A` and `B` are parameters standing for arbitrary propositions. As a consequence it can model faithfully the logical rule $\wedge_I$.

Note that the definition of the pair data type, in section 1.3.3, is almost identical to the one of `and`:

```
Inductive prod (A B : Type) := pair (a : A) (b : B).
```

The striking similarity illustrates how programs (and data) are used in Coq to represent proofs.

Pattern matching provides a way to express the elimination rule for conjunction as follows:

```
Definition proj1 A B (p : A /\ B) : A :=
  match p with conj a _ => a end.
```

Now recall the similarity between $\rightarrow$ and $\forall$, where the former is the simple, non-dependent case of the latter. If we ask for the type of the `conj` constructor:

```
About conj.
```

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

we may wonder what happens if the type of the second argument (i.e., `B`) becomes dependent on the value of the first argument (of type `A`). What we obtain is actually the inductive definition corresponding to the existential quantification.

```
Inductive ex (A : Type) (P : A -> Prop) : Prop :=
  ex_intro (x : A) (p : P x).
Notation "'exists' x : A , p" := (ex A (fun x : A => p)).
```

As `ex_intro` is the only constructor of the `ex` inductive type, it is the only means to prove a statement like `(exists n, prime n)`. In such a — constructive — proof, the first argument would be a number `n` of type `nat` while the second argument would be a proof `p` of type `(prime n)`. The parameter `P` causes the dependency of the second component of the pair on the first component. It is a function representing an arbitrary predicate over a term of type `A`. Hence `(P x)` is the instance of the predicate, for `x`. E.g., the predicate of being an odd prime number is expressed as `(fun x : nat => (odd x) && (prime x))`, and the statement expressing the existence of such a number is

`(ex nat (fun x : nat => (odd x) && (prime x)))`, which (thanks to the `Notation` mechanism of Coq) is parsed and

printed as the more familiar `(exists x : nat, odd x && prime x)`.

It is worth summing up the many features of type theory that intervene in the type of `ex` and `ex_intro`:

```
ex : ∀ A : Type, (A -> Prop) -> Prop.
ex_intro : ∀ A : Type, ∀ P : A -> Prop, ∀ a : A, P a -> ex A P.
```

The first quantification on `A` is the first one we encountered in this book, and expresses polymorphism. Then we find an higher order quantification of a predicate `P` that makes the `ex` inductive declaration work for any formula describing a property of the witness. Finally the dependent quantification `∀ a : A` binds a term variable `a` in the type that follows to express that the property `P` must hold on the witness `a`.

Let us also observe the inductive definition of the disjunction `or` and its two

constructors `or_introl` and `or_intror`.

```
Inductive or (A B : Prop) : Prop := or_introl (a : A) | or_intror (b : B).
Notation "A \/ B" := (or A B).
```

The elimination rule can again be expressed by pattern matching:

```
Definition or_ind (A B P : Prop)
  (aob : A \/ B) (pa : A -> P) (pb : B -> P) : P :=
  match aob with or_introl a => pa a | or_intror b => pb b end.
```

The detail worth noting here is that the pattern match construct has two branches, and each branch represents a distinct sub proof. In this case, in order to prove `P` starting from `A \/ B`, one has to deal with all cases: i.e., to prove `P` under the assumption `A`, and to prove `P` under the assumption `B`.

Usual constants and connectives such as $\top$, $\bot$ and $\neg$ can be defined as follows.

```
Inductive True : Prop := I.
Inductive False : Prop := .
Definition not (A : Prop) := A -> False.
Notation "~ A" := (not A).
```

Hence, in order to prove `True`, one just has to apply the constructor `I`, which requires no arguments. So proving `True` is trivial, and as a consequence eliminating it provides little help (i.e., no extra knowledge is obtained by pattern matching over `I`). Contrarily, it is impossible to prove `False`, since it has no constructor, and pattern matching on `False` can inhabit any type, since no branch has to be provided:

```
Definition exfalso (P : Prop) (f : False) : P :=
  match f with end.  (* no constructors, no branches *)
```

The only base predicate we still haven't described is equality. The reason we left it as the last one is that it has a tricky nature. In particular, equality, as we have seen in the previous chapters, is an open notion in the following sense. Terms that compute to the same syntactic expression are considered as equal, and this is true for any program the user may write. Hence such notion of equality needs to be somewhat primitive, as `match` and `fun` are. One also expects such notion to come with a substitutivity property: replacing equals by equals must be licit.

The way this internal notion is exposed is via the concept of index on which an inductive type may vary.

```
Inductive eq (A:Type) (x:A) : A -> Prop := erefl : eq A x x.
Notation "x = y" := (@eq _ x y).
```

This is the first time we see a function type after the `:` symbol in an inductive type declaration. The `eq` type constructor takes three arguments: a type `A`

and two terms of that type (the former is named `x`). Hence one can write `(a = b)` whenever `a` and `b` have the same type. The `erefl` constructor takes no arguments, as `I`, but its type annotation says it can be used to inhabit only the type `(x = x)`. Hence one is able to prove `(a = b)` only when `a` and `b` are convertible (i.e., indistinguishable from a logical standpoint). Conversely, by eliminating a term of type `(a = b)` one discovers that `a` and `b` are equal and `b` can be freely replaced by `a`.

```
1  Definition eq_ind A (P : A -> Prop) x (px : P x) y (e : x = y) : P y :=
2    match e with erefl => px end.
```

The notion of equality is one of the most intricate aspects of type theory; an in-depth study of it is out of the scope of this book. The interested reader finds an extensive study of this subject in [25]. Later in this chapter we define and use other inductive types to take advantage of the "automatic" substitution of the implicit equations we see here: While `px` has type `(P x)`, it is accepted as an inhabitant of `(P y)` because inside the `match` the term `y` is automatically replaced by `x`.

3.2 A script language for interactive proving

Since proofs are just terms one could, in principle, use no proof language and directly input proof terms instead. Indeed this was the modus operandi in the pioneering work of De Bruijn on Automath (automating mathematics) in the seventies [18]. The use of a dedicated proof language enables a higher level description of the formal proof being constructed. Importantly, it provides support for taming the bookkeeping part of the activity of interactive proving.

3.2.1 Proof commands

The proof commands we have mentioned in chapter 2 can all be explained in terms of the proof terms they produce behind the scenes. For example, `case: n` provides a much more compact syntax for `(match .. with .. end)` and it produces a `match` expression with the right shape by looking at the type of `n`. If `n` is a natural number, then there are two branches; the one for the `S` constructor carries an argument of type `nat`, the other one is for `0` and binds no additional term. The `case:` tactic is general enough to work with any inductive data type and inductive predicate. Note that this tactic is known as *destruction*, since it transforms an object of an inductive type back into the arguments that this object was constructed from (using the constructors of the type).

The `apply:` tactic generates an application. For example, `apply: addnC` generates the term `(addnC t1 t2)` by figuring out the correct values of `t1` and `t2`, or opening new goals when this cannot be done, i.e., if the lemma takes in input proofs, like `contraL`.

There is a list of proof commands that are shorthands for `apply:` and are only worth mentioning here briefly. `split` proves a conjunction by applying

the `conj` constructor; `left` and `right` prove a disjunction by applying `or_introl` and `or_intror` respectively; `exists t` proves an existentially quantified formula by providing the witness `t` and, later, a proof that `t` validates the predicate. Finally `reflexivity` proves an equality by applying `erefl`.

The only primitive constructor that remains without an associated proof command is `(fun .. => ..)`. Operationally, what the $\rightarrow_I$ and $\forall_I$ logical rule do is to introduce into the proof context a new entry. So far we either expressed this step at the beginning of proofs by placing such items just after the name of the lemma being proved, or just after a `case:` or `elim:` proof command. The current section expands on this subject covering the full management of the proof context.

3.2.2 Goals as stacks

The presentation we gave so far of proof commands like `case: n => [|m]` is over-simplified. While `case` is indeed the proof command in charge of performing case analysis, the "`: n`" and "`=> [|m]`" parts are decorators to prepare the goal and post-process the result of the proof command. These decorators perform what we typically call *bookkeeping*: actions that are necessary in order to obtain readable and robust proof scripts but that are too frequent to benefit from a more verbose syntax. Bookkeeping actions do convey a lot of information, like where names are given to assumptions, but also let one deal with annoying details using a compact, symbolic, language. Note that all bookkeeping actions correspond to regular, named, proof commands. It is the use one makes of them that may be twofold: a case analysis in the middle of a proof may start two distinct lines of reasoning, and hence it is worth being noted explicitly with the `case` word; conversely, de-structuring a pair to obtain the two components can hardly be a relevant step in a proof, so one may prefer to perform such bookkeeping action with a symbolic, compact, notation corresponding to the same `case` functionality.

Pulling from the stack

Let's start with the post-processing phase, called *introduction pattern*. The postfix "`=> ...`" syntax can be used in conjunction with any proof command, and it performs a sequence of actions on the first assumption or variable that appears in the goal (i.e., on `A` if the goal has the form `A -> B -> C -> ...`, or on `x` if the goal has the form $\forall$ `x, ...`). With these looking glasses, the goal becomes a *stack*. Take for example this goal:

```
========================
∀ xy, prime xy.1 -> odd xy.2 -> 2 < xy.2 + xy.1
```

Before accessing the assumption `(prime xy.1)`, one has to name the bound variable `xy`, exactly as one can only access a stack from its top. The execution of `=> xy pr_x odd_y` is just the composition of `=> xy` with `=> pr_x` and finally `=> odd_y`.

Each action pulls an item out of the stack and names it. The `move` proof command does nothing, so we use it as a placeholder for the postfix `=>` bookkeeping action:

```
move=> xy pr_x odd_y.
```

```
xy : nat * nat
pr_x : prime xy.1
odd_y : odd xy.2
========================
2 < xy.2 + xy.1
```

Now, en passant, we would like to decompose `xy` into its first and second component. Instead of the verbose `=> xy; case: xy => x y`, we can use the symbolic notation `[]` to perform such action.

```
move=> [x y] pr_x odd_y.
```

```
x, y : nat
pr_x : prime (x,y).1
odd_y : odd (x,y).2
========================
2 < (x,y).2 + (x,y).1
```

We can place the `/=` switch to force the system to reduce the formulas on the stack, before introducing them in the context, and obtain:

```
move=> [x y] /= pr_x odd_y.
```

```
x, y : nat
pr_x : prime x
odd_y : odd y
========================
2 < y + x
```

We can also process an assumption through a lemma; when a lemma is used in this way, it is called a *view*. This feature is of general interest, but it is especially useful in the context of boolean reflection, as described in section 4.1. For example, `prime_gt1` states `(prime p -> 1 < p)` for any `p`, and we can use it as a function to obtain a proof of `(1 < x)` from a proof of `(prime x)`.

```
move=> [x y] /= /prime_gt1-x_gt1 odd_y.
```

```
x, y : nat
x_gt1 : 1 < x
odd_y : odd y
========================
2 < y + x
```

The leading `/` makes `prime_gt1` work as a function instead of as a name to be assigned to the top of the stack. The `-` has no effect but to visually link the function with the name `x_gt1` assigned to its output. Indeed `-` can be omitted.

One could also examine `y`: it can't be `0`, since it would contradict the assumption saying that `y` is `odd`.

```
move=> [x [//|y]] /= /prime_gt1-x_gt1.
```

```
x, y : nat
x_gt1 : 1 < x
========================
~~ odd y -> 2 < y.+1 + x
```

This time, the destruction of `y` generates two cases for the two branches; hence the `[ .. | .. ]` syntax. In the first one, when `y` is `0`, the `//` action solves the goal, by the trivial means of the `by []` terminator. In the second branch we

name `y` the new variable (which is legitimate, since the old `y` has been disposed of).

Now, the fact that `y` is even is not needed to conclude, so we can discard it by giving it the `_` dummy name.

```
by move=> [x [//|y]] /= /prime_gt1-x_gt1 _; apply: ltn_addl x_gt1.
```

The way to discard an already named assumption is to mention its name in curly braces, as `=> {x_gt1}`.

We finally conclude with the `apply:` command. In the example just shown, we have used it with two arguments: a function and its last argument. In fact, the lemma `ltn_addl` looks as follows:

```
About ltn_addl.
```

```
ltn_addl : ∀ m n p : nat, m < n -> m < p + n

Arguments m, n are implicit
```

`apply:` automatically fills in the blanks between the function `ltn_addl` (the lemma name) and the argument `x_gt1` provided. Since we are passing `x_gt1`, the variable `m` picks the value `1`. The conclusion of `ltn_addl` hence unifies with `(2 < y.+1 + x)` because both `+` and `<` are defined as programs that compute: Namely, addition exposes a `.+1`, thus reducing to `2 < (y+x).+1`; then `<` (or, better, the underlying `<=`) eats a successor from both sides, leading to `1 < y + x`, which looks like the conclusion of the lemma we apply.

Here we have shown all possible actions one can perform in an intro pattern, squeezing the entire proof into a single line. This has to be seen both as an opportunity and as a danger: one can easily make a proof unreadable by performing too many actions in the bookkeeping operator `=>`. At the same time, a trivial sub-proof like this one should take no more than a line, and in that case one typically sacrifices readability in favor of compactness: what would you learn by reading a trivial proof? Of course, finding the right balance only comes with experience.

Warning

The case intro pattern `[..|..]` obeys an exception: when it is the first item of an intro pattern, it does not perform a case analysis, but only branch on the subgoals. Indeed in `case: n => [|m]` only one case analysis is performed.

Working on the stack

The stack can also be used as a workplace. Indeed, there is no need to pull all items from the stack. If we take the previous example:

```
  ========================
  ∀ xy, prime xy.1 -> odd xy.2 -> 2 < xy.2 + xy.1
```

and we stop just after applying the view, we end up in a valid state:

```
1  move=> [x y] /= /prime_gt1.
```

```
  x, y : nat
  ========================
  1 < x -> odd y -> 2 < y + x
```

One can also chain multiple views on the same stack item:

```
1  move=> [x y] /= /prime_gt1/ltnW.
```

```
  x, y : nat
  ========================
  0 < x -> odd y -> 2 < y + x
```

Two other operations are available on the top stack item: specialization and substitution. Let's take the following conjecture.

```
  ========================
  (∀ n, n * 2 = n + n) -> 6 = 3 + 3
```

The top stack item is a quantified assumption. To specialize it to, say, `3` one can write as follows:

```
1  move=> /(_ 3).
```

```
  ========================
  3 * 2 = 3 + 3 -> 6 = 3 + 3
```

The idea behind the syntax here is that when we apply a view `v` to the top stack item (say, `top`), by writing `/v`, we are forming the term `(v top)`, whereas when we specialize the top stack item `top` to an object `x`, by writing `/(_ x)`, we are forming the term `(top x)`. The `_` is a placeholder for the top item, and is omitted in `/(v _)`.

When the top stack item is an equation, one can substitute it into the rest of the goal, using the tactics `<-` and `->` for right-to-left and left-to-right respectively.

```
1  move=> /(_ 3) <-.
```

```
  ========================
  6 = 3 * 2
```

In other words, the arrows are just a compact syntax for rewriting, as in the `rewrite` tactic, with the top assumption.

Pushing to the stack

We have seen how to pull items from the stack to the context. Now let's see the so called *discharging* operator `:`, performing the converse operation. Such operator decorates proof commands as `move`, `case` and `elim` with actions to be performed before the command is actually run.

Warning

The colon symbol in `apply:` is not the discharging operator. It is just a marker to distinguish the `apply:` tactic of Ssreflect from the `apply` tactic of Coq. Indeed the two tactics, while playing similar roles, behave very differently.

Imagine we want to perform case analysis on `y` at this stage:

```
x, y : nat
x_gt1 : 1 < x
odd_y : odd y
========================
2 < y + x
```

The command `case: y` is equivalent to `move: y; case.` where `move` once again is a placeholder, `: y` pushes the `y` variable onto the stack, and `case` operates on the top item of the stack. Pushing items on the stack is called *discharging*.

Just before running `case`, the goal would look like this:

```
x : nat
x_gt1 : 1 < x
odd_y : odd y
========================
∀ y, 2 < y + x
```

However, this is not actually a well-defined state. Indeed, the binding for `y` is needed by the `odd_y` context item, so `move: y` fails. One has to push items onto the stack in a valid order: first, all properties of a variable, then the variable itself. The correct invocation, `move: y odd_y`, pushes first `odd_y` and only then `y` onto the stack, leading to the valid goal

```
x : nat
x_gt1 : 1 < x
========================
∀ y, odd y -> 2 < y + x
```

Via the execution of `case` one obtains:

```
2 subgoals

  x : nat
  x_gt1 : 1 < x
  ========================
   odd 0 -> 2 < 0 + x

subgoal 2 is:
 ∀ n : nat, odd n.+1 -> 2 < n.+1 + x
```

An alternative to discharging `odd_y` would be to clear it, i.e., purge it from the context. Listing context entry names inside curly braces has this effect, like in the case of `case: y {odd_y}`.

One can combine : and => around a proof command, to first prepare the goal for its execution and finally apply the necessary bookkeeping to the result. For example:

```
1  case: y odd_y => [|y']
```

```
2 subgoals

  x : nat
  x_gt1 : 1 < x
  ========================
   odd 0 -> 2 < 0 + x

subgoal 2 is:
 odd y'.+1 -> 2 < y'.+1 + x
```

At the left of the : operator one can also put a name for an equation that links the term at the top of the stack before and after the execution of the tactic. For example, `case E: y odd_y => [|y']` leads to the following two subgoals:

```
  x, y : nat
  x_gt1 : 1 < x
  E : y = 0
============================
 odd 0 -> 2 < 0 + x
```

```
  x, y : nat
  x_gt1 : 1 < x
  y' : nat
  E : y = y'.+1
========================
 odd y'.+1 -> 2 < y'.+1 + x
```

Last, one can push any term onto the stack – whether or not this term appears in the context. For example, "`move: (leqnn 7)`" pushes on the stack the additional assumption `(7 <= 7)` (and thus "`move: (leqnn 7) => A`" brings this assumption into the context under the name `A`). This will come handy in section 3.2.4.

3.2.3 Inductive reasoning

In chapter 1 we have seen how to build and use (call or destruct) anonymous functions and data types. All these constructions have found counterparts in the Curry-Howard correspondence. The only missing piece is recursive programs. For example, `addn` was written by recursion on its first argument, and is a function taking as input two numbers and producing a third one. We can write programs by recursion that take as input, among regular data, proofs and produce other proofs as output. Let's look at the induction principle for natural numbers trough the looking glasses of the Curry-Howard isomorphism.

```
1  About nat_ind.
```

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

`nat_ind` is a program that produces a proof of `(P n)` for any `n`, proviso a proof for the base case `(P 0)`, and a proof of the inductive step `(∀ n : nat, P n -> P n.+1)`. Let us write such a program by hand.

```
Fixpoint nat_ind (P : nat -> Prop)
  (p0 : P 0) (pS : ∀ n : nat, P n -> P n.+1) n : P n :=
  if n is m.+1 then
    let pm (* : P m *) := nat_ind P p0 pS m in
    pS m pm (* : P m.+1 *)
  else p0.
```

The Coq system generates this program automatically, as soon as the `nat` data type is defined. Recall that recursive functions are checked for termination: Through the lenses of the proofs-as-programs correspondence, this means that the induction principle just coded is sound, i.e., based on a well-founded order relation.

If non-terminating functions are not ruled out, it is easy to inhabit the `False` type, even if it lacks a proper constructor.

```
Fixpoint oops (n : nat) : False := oops n.
Check oops 3.  (* : False *)
```

Of course Coq rejects the definition of `oops`. To avoid losing consistency, Coq also enforces some restrictions on inductive data types. For example the declaration of `hidden` is rejected.

```
Inductive hidden := Hide (f : hidden -> False).
Definition oops (hf : hidden) : False := let: Hide f := hf in f hf.
Check oops (Hide oops).  (* : False *)
```

Note how `oops` calls itself, as in the previous example, even if it is not a recursive function. Such restriction, called *positivity condition*, is out of scope for this book. (Roughly speaking, it says that constructors for an inductive data type can only depend on maps *to* the data type but not on maps *from* it.) The interested reader shall refer to [24].

3.2.4 Application: strengthening induction

As an exercise we show how the `elim` tactic combined with the bookkeeping operator `:` lets one perform, on the fly, a stronger variant of induction called "course of values" or "strong induction".

Claim: every amount of postage that is at least 12 cents can be made from 4-cent and 5-cent stamps. The proof in the inductive step goes as follows. There are obvious solutions for a postage between 12 and 15 cents, so we can assume it is at least 16 cents. Since the postage amount is at least 16, by using a 4-cents stamp we are back at a postage amount that, by induction, can be obtained as claimed. □

The tricky step is that we want to apply the induction hypothesis not on $n-1$, as usual, but on $n-4$, since we know how to turn a solution for a stamping amount problem n to one for a problem of size $n+4$ (by using a 4-cent stamp). The induction hypothesis provided by `nat_ind` is not strong enough. However we can use the `:` operator to load the goal before performing the induction.[3]

[3] See further below for explanations.

```
1  Lemma stamps n : 12 <= n -> exists s4 s5, s4 * 4 + s5 * 5 = n.
2  Proof.
3  elim: n {-2}n (leqnn n) =>[|n IHn]; first by case.
4  do 12! [ case; first by [] ]. (* < 12c *)
5  case; first by exists 3, 0.   (* 12c = 3 * 4c *)
6  case; first by exists 2, 1.   (* 13c = 2 * 4c + 1 * 5c *)
7  case; first by exists 1, 2.   (* 14c = 1 * 4c + 2 * 5c *)
8  case; first by exists 0, 3.   (* 15c = 3 * 5c *)
9  move=> m'; set m := _.+1; move=> mn m11.
10 case: (IHn (m-4) _ isT) => [|s4 [s5 def_m4]].
11   by rewrite leq_subLR (leq_trans mn) // addSnnS leq_addl.
12 by exists s4.+1, s5; rewrite mulSn -addnA def_m4 subnKC.
13 Qed.
```

Line 3 requires some explanations. First of all, "`elim: n {-2}n (leqnn n).`" is shorthand for "`move: (leqnn n). move: {-2}n. move: n. elim.`" (the terms after the colon `:` are pushed to the stack, from right to left; and the `elim` tactic is applied afterwards to the top of the stack, which of course is the last term pushed to the stack). Let us see how these four steps transform the goal. At the beginning, the context and the goal are

```
  n : nat
  ============================
  11 < n -> exists s4 s5 : nat, s4 * 4 + s5 * 5 = n
```

The first of our four steps ("`move: (leqnn n)`") pushes the additional assumption `n <= n` onto the stack (since `(leqnn n)` provides its proof); we are thus left with

```
1 subgoal

  n : nat
  ============================
  n <= n -> 11 < n -> exists s4 s5 : nat, s4 * 4 + s5 * 5 = n
```

The second step ("`move: {-2}n`") is more interesting. Recall that `move:` usually "generalizes" a variable (i.e., takes a variable appearing in the context, and binds it by the universal quantifier, so that the goal becomes a for-all statement). The prefix `{-2}` means "all except for the 2nd occurrence in the goal"; so the idea is to generalize "all except for the 2nd occurrence of n". Of course, this implies that n still has to remain in the context (unlike for "`move: n`"), so the bound variable of the universal quantifier has to be a fresh variable, picked automatically by Coq. For example, Coq might pick `n0` for its name, and so the state after the second step will be:

```
  n : nat
  ============================
  ∀ n0 : nat,
  n0 <= n -> 11 < n0 -> exists s4 s5 : nat, s4 * 4 + s5 * 5 = n0
```

Notice how the `n` in "`n0 <= n`" has remained un-generalized, since it was the 2nd occurrence of n before the second step.

The third step ("`move: n`") merely moves the `n` from the context to the goal.

Thus, after the three "move" steps, we are left with proving the following claim:

```
1  ∀ n n0 : nat,
2    n0 <= n -> 11 < n0 -> exists s4 s5 : nat, s4 * 4 + s5 * 5 = n0
```

The elim now applies induction on the top of the stack, which is n. The corresponding induction hypothesis IHn is:

```
1  IHn : ∀ n0 : nat,
2        n0 <= n ->
3        11 < n0 -> exists s4 s5 : nat, s4 * 4 + s5 * 5 = n0
```

(Of course, the n here is not the original n, but the new n introduced in the =>[|n IHn] pattern.)

Lines 4, 9 and 10 deserve a few comments.

- Line 4 repeats a tactic 12 times using the do 12! tactical. This deals with the 12 cases where n0 is not greater than 11.

- Line 9 uses the set proof command, which is used to define a new constant in the context. For example, "set a := 24 * 11." would create a new "a := 24 * 11 : nat" item in the context. The command also tries to substitute the newly-defined constant for its appearances in the goal; for example, "set a := 11." would not only create a new "a := 11 : nat" in the context, but also replace the "11 < m'.+4.+4.+4.+4" in the goal[4] by an "a < m'.+4.+4.+4.+4". In our example above ("set m := _.+1"), we are using the set command with a wildcard; this captures the first term of the form _.+1 appearing in the goal and denotes it by m, replacing it by m on the goal. In our case, this first term is m'.+4.+4.+4.+4 (which is just syntactic sugar for m'.+1.+1..... with 16 appearances of .+1)[5], and so the name m is given to this term. We could have achieved the same goal using "set m := m'.+4.+4.+4.+4".

  Further details about the set tactic can be found in [13, §4.2]; let us only mention the (by now) habitual variant set a := {k}(pattern), which defines a new constant a to equal the first subterm of the goal that matches the pattern pattern, but then replaces *only the k-th* appearance of this subterm in the goal by a. As usual, if the pattern-matching algorithm keeps finding the wrong subterms, it is always possible to completely specify the subterm, leaving no wildcards.

- Line 10 instantiates the induction hypothesis with the value (m-4), with a placeholder for a missing proof of (m-4 < n), and with a proof that (11 < m-4). The proof given for (11 < m-4) is just a simple application of isT; this is accepted because the term (11 < m-4) *computes* to true (thanks

[4] Don't be surprised by the fact that an addition of 16 is circumscribed by four additions of 4. By default, Mathematical Components has the notations .+1, .+2, .+3 and .+4 pre-defined, but not .+5 and higher.

[5] This is slightly counterintuitive, as you might instead believe it to be m'.+3.+4.+4.+4. But keep in mind that m'.+3.+4.+4.+4 < n.+1 is just syntactic sugar for m'.+4.+4.+4.+4 <= n.+1.

to the computational definitions of `<` and of subtraction, and thanks to `m := m'.+4.+4.+4.+4`). The missing proof of `(m-4 < n)` is not automatically inferred; it becomes the first subgoal. The induction hypothesis yields `exists s4 s5 : nat, s4 * 4 + s5 * 5 = m-4`. The introduction pattern in line 10 gives names to the values of `s4` and `s5` whose existence is thus guaranteed, and to the statement that `s4 * 4 + s5 * 5 = m-4`. In other words, it gives names `s4` and `s5` to the quantities of 4-cent and 5-cent stamps needed to cover the amount of postage `(m-4)`, as well as to the fact that they do cover this exact amount of postage.

3.3 On the status of Axioms

Not all valid reasoning principles can be represented by programs. For example, excluded middle — i.e., the claim that `A \/ ~~ A` for a given logical statement `A` — can be proved by a program only when `A` is decidable, i.e., when we can write in Coq a program to `bool` that tests if `A` holds or not. Excluded middle, in its generality, can only be *axiomatized*, i.e., assumed globally.

The Mathematical Components library is axiom free. This makes the library compatible with any combination of axioms that is known to be consistent with the Calculus of Inductive Constructions. Some formal developments in Coq tend to assume the excluded middle axiom, even if the proofs formalized do not require it, to avoid (or postpone) proving the decidability of the predicates at play while still being able to reason by case analysis on their validity. By contrast, the Mathematical Components library provides the decidability proofs when they exist. Moreover, the boolean reflection methodology, object of chapter 4, provides tools to manipulate decidable predicates in a convenient way, with the same ease as in a formalism based on a classical logic.

Yet sometimes it is not possible to provide a constructive proof of a statement. In this case, axioms like the excluded middle axiom can still be confined into "boxes". The purpose of these boxes is to delimit a local context in which the required axioms are available. Such a box is typically called a *monad* in the theory of programming languages. Here is an example, for the axiom of excluded middle:

```
1  Definition classically P : Prop := ∀ b : bool, (P -> b) -> b.
2  Lemma classic_EM P : classically (decidable P).
3  Lemma classicW P : P -> classically P.
4  Lemma classic_bind P Q :
5    (P -> classically Q) -> classically P -> classically Q.
```

The `classically` box can only be opened when the statement to be proved in the current goal is a boolean, hence an instance of a decidable predicate. Inside such a box the excluded middle is made available by combining `classic_EM` and `classic_bind`. Nevertheless, when proving a statement that is not a boolean, like `exists n, ...`, one cannot access assumptions in the `classically` box.

In other cases, axioms can be avoided by rephrasing the mathematics in a

weaker setting. A notable example in the Mathematical Components library is the construction of the real closure of an Archimedean field [6].

Chapter 4

Boolean Reflection

At this stage, we are in the presence of one of the main issues in the representation of mathematics in a formal language: Very often, several data structures can be used to represent one and the same mathematical definition or statement. The choice between them may have a significant impact on the upcoming layers of formalized theories. So far, we have seen two ways of expressing logical statements: using boolean predicates and truth values on one hand, and using logical connectives and the `Prop` sort on the other. For instance, in order to define the predicate "the sequence `s` has at least one element satisfying the (boolean) predicate `a`", we can either use a boolean predicate:

```
Fixpoint has T (a : T -> bool) (s : seq T) : bool :=
  if s is x :: s' then a x || has a s' else false.
```

or we can use an alternate formula, like for instance:

```
Definition has_prop  T (a : T -> bool) (x0 : T) (s : seq T) :=
   exists i, i < size s /\ a (nth x0 s i)
```

(where we are assuming that `x0` is a given element of `T`; otherwise, we can, e.g., quantify the whole statement by `exists (x0 : T)`).

Term `(has a s)` is a boolean value. It is hence easy to use it in a proof to perform a case analysis on the fact that sequence `s` has an element such that `a` holds, using the `case` tactic:

```
case: (has a s).
```

As we already noted, computation provides some automation for free. For example in order to establish that `(has odd [:: 3; 2; 7]) = true`, we only need to observe that the left hand side *computes* to `true`.

It is not possible to perform a similar case analysis in a proof using the alternative version `(s_has_aP : has_prop a x0 s)`, since, as sketched in section 3.3, excluded middle holds in Coq only for boolean tests. On the other hand, this

phrasing of the hypothesis easily gives access to the value of the index at which the witness is to be found:

```
case: s_has_aP => [n [n_in_s asn]].
```

introduces in the context of the goal a natural number `n : nat` and the fact `(asn : a (nth x0 s n))`. In order to establish that `(has_prop a x0 s)`, we cannot resort to computation. Instead, we can prove it by providing the index at which a witness is to be found — plus a proof of this fact — which may be better suited for instance to an abstract sequence `s`.

In summary, boolean statements are especially convenient for excluded middle arguments and its variants (reductio ad absurdum, ...). They furthermore provide a form of small-step automation by computation.[1] Specifications in the `Prop` sort are structured logical statements, that can be "destructed" to provide witnesses (of existential statements), instances (of universal statements), subformulae (of conjunctions), etc.. They are proved by deduction, building proof trees made with the rules of the logic. Formalizing a predicate by means of a boolean specification requires implementing a form of decision procedure and possibly proving a specification lemma if the code of the procedure is not a self-explanatory description of the mathematical notion. For instance a boolean definition `(prime : nat -> bool)` implements a complete primality test, which requires a companion lemma proving that it is equivalent to the usual definition in terms of proper divisors. Postulating the existence of such a decision procedure for a given specification is akin to assuming that the excluded middle principle holds on the corresponding predicate.

The boolean reflection methodology proposes to avoid committing to one or the other of these options, and provides enough infrastructure to ease the bureaucracy of navigating between the two. The `is_true` coercion, which was being used silently in the background since the early pages of chapter 2, is in fact one piece of this infrastructure.

The is_true coercion

```
Definition is_true (b : bool) : Prop := b = true.
Coercion is_true : bool >-> Sortclass.
```

The `is_true` function is automatically inserted by Coq to turn a boolean value into a `Prop`, i.e. into a regular statement of a theorem. More on this mechanism will be told in section 4.4.

[1] They moreover allow for proof-irrelevant specifications. This feature is largely used throughout the Mathematical Components library but beyond the scope of the present chapter: it will be the topic of chapter 6.

4.1 Reflection views

4.1.1 Relating statements in `bool` and `Prop`

How to best formalize the equivalence between a boolean value `b` and a statement `P : Prop`? The most direct way would be to use the conjunction of the two converse implications:

```
Definition bool_Prop_equiv (P : Prop) (b : bool) := b = true <-> P.
```

where `(A <-> B)` is defined as `((A -> B) /\ (B -> A))`.

Yet, as we shall see in this section, we can improve the phrasing of this logical sentence, in order to improve its usability. For instance, although `(bool_Prop_equiv P b)` implies that the excluded middle holds for `P`, it does not provide directly a convenient way to reason by case analysis on the fact that `P` holds or not, or to use its companion version `(b = false <-> ~ P)`. The following proof script illustrates the kind of undesirable bureaucracy entailed by this wording:

```
Lemma test_bool_Prop_equiv b P :
    bool_Prop_equiv P b -> P \/ ~ P.
Proof.
case: b; case => hlr hrl.
  by left; apply: hlr.
by right => hP; move: (hrl hP).
Qed.
```

Last goal

```
1 subgoal
P : Prop
hlr : false = true -> P
hrl : P -> false = true
========================
P \/ ~ P
```

We could try alternative formulations based on the connectives seen in section 3, like for instance `(b = true /\ P) \/ (b = false /\ ~ P)`, but again the bureaucracy would be non-negligible.

A better solution is to use an ad hoc inductive definition that resembles a disjunction of conjunctions: we inline the two constructors of a disjunction and each of these constructors has the two arguments of the conjunction's single constructor:

```
Inductive reflect (P : Prop) (b : bool) : Prop :=
| ReflectT (p : P)    (e : b = true)
| ReflectF (np : ~ P) (e : b = false).
```

We can prove that the statement `reflect P b` is actually equivalent to the double implication `bool_Prop_equiv`; see exercise 12.

Let us illustrate the benefits of this alternate specialized double implication:

```
Lemma test_reflect b P :
  reflect P b -> P \/ ~ P.
Proof.
case.
```

```
  b : bool
  P : Prop
  ============================
   P -> b = true -> P \/ ~ P

subgoal 2 is:
 ~ P -> b = false -> P \/ ~ P
```

A simple case analysis on the hypothesis `(reflect P b)` exposes in each branch both versions of the statement. Note that the actual `reflect` predicate defined

in the ssrbool library is slightly different from the one we give here: this version misses an ultimate refinement that will be presented in section 4.2.1.

We start our collection of links between boolean and Prop statements with the lemmas relating boolean connectives with their Prop versions:

```
Lemma andP (b1 b2 : bool) : reflect (b1 /\ b2) (b1 && b2).
Proof.  by case: b1; case: b2; [ left | right => //= [[l r]] ..]. Qed.

Lemma orP (b1 b2 : bool) : reflect (b1 \/ b2) (b1 || b2).
Proof.
case: b1; case: b2; [ left; by [ move | left | right ] .. |].
by right=> // [[l|r]].
Qed.

Lemma implyP (b1 b2 : bool) : reflect (b1 -> b2) (b1 ==> b2).
Proof.
by case: b1; case: b2; [ left | right | left ..] => //= /(_ isT).
Qed.
```

In each case, the lemma is proved using a simple inspection by case analysis of the truth table of the boolean formula. The case analysis generates several branches and we use a special syntax to describe the tactics which should be applied to some specific branches, and the tactic which should be applied in the general case. The "`;[ t1 | t2 .. | tn ]`" syntax indeed corresponds to the application of the tactic `t1` to the first subgoal generated by what comes before `;`, and the application of the tactic `tn` to the last subgoal, and the application of the tactic `t2` to all the branches in between. See [24, section 9.2]) for a complete description of this feature.

More generally, a theorem stating an equivalence between a boolean expression and a Prop statement is called a *reflection view*, since it is used to view an assumption from a different perspective.

Advice

The name of a reflection view always ends with a capital P.

The next section is devoted to the proof and usage of more involved views.

4.1.2 Proving reflection views

Reflection views are also used to specify types equipped with a *decidable equality*, by showing that the equality predicate eq (seen in section 3.1.3) is implemented by a certain boolean equality test. For instance, we can specify the boolean equality test on type nat implemented in chapter 1 as:

```
Lemma eqnP (n m : nat) : reflect (n = m) (eqn n m).
```

Each implication can be proved by a simple induction on one of the natural numbers, but we still need to generate the two subgoals corresponding to these implications, as the `split` tactic is of no help here.

In order to trigger this branching in the proof tree, we resort to the bridge between the `reflect` predicate and a double implication. The `ssrbool` library provides a general version of this bridge:

```
1  About iffP.
```

```
iffP : ∀ (P Q : Prop) (b : bool),
  reflect P b -> (P -> Q) -> (Q -> P) -> reflect Q b
```

Lemma `iffP` relates two equivalences `(reflect P b)` and `(reflect Q b)` involving one and the same boolean `b` but different `Prop` statements `P` and `Q`, as soon as one provides a proof of the usual double implication between `P` and `Q`.

The trivial reflection view is called `idP` and is seldom used in conjunction with `iffP`.

```
1  Lemma idP {b : bool} : reflect b b.
```

We can now come back to the proof of lemma `eqnP`, and start its proof script by applying `iffP`.

```
1  Lemma eqnP {n m : nat} :
2    reflect (n = m) (eqn n m).
3  Proof.
4  apply: (iffP idP).
```

```
n : nat
m : nat
====================
 m = n -> eqn m n

subgoal 2 is:
 eqn m n -> m = n
```

The proof is now an easy induction, and is left as exercise 13.

Let us now showcase the usage of the more general form of `iffP` by proving that a type equipped with an injection in type `nat` has a decidable equality:

```
1  Lemma nat_inj_eq T (f : T -> nat) x y :
2    injective f -> reflect (x = y) (eqn (f x) (f y)).
```

The equality decision procedure just consists in pre-applying the injection `f` to the decision procedure `eqn` available on type `nat`. Since we already know that `eqn` is a decision procedure for equality, we just need to prove that `(x = y)` if and only if `(f x = f y)`, which follows directly from the injectivity of `f`. Using `iffP`, a single proof command splits the goal into two implications, replacing on the fly the evaluation `(eqn (f x) (f y))` by the `Prop` equality `(f x = f y)`:

```
1  Lemma nat_inj_eq T (f : T -> nat) x y :
2    injective f ->
3      reflect (x = y) (eqn (f x) (f y)).
4  Proof.
5  move=> f_inj.
6  apply: (iffP eqnP).
```

```
T : Type
f : T -> nat
f_inj : injective f
x, y : T
====================
x = y -> f x = f y

subgoal 2 is:
 f x = f y -> x = y
```

Note that `eqn`, being completely specified by `eqnP`, is not anymore part of the

picture. Finishing the proof is left as exercise 14.

The latter example illustrates the convenience of combining an action on a goal, here breaking an equivalence into one subgoal per implication, with a change of viewpoint, here by the means of the `eqnP` view. This combination of atomic proof steps is pervasive in a library designed using the boolean reflection methodology: the Ssreflect tactic language lets one use view lemmas freely in the middle of intro-patterns.

4.1.3 Using views in intro patterns

Reflection views are typically used in the bookkeeping parts of formal proofs, and thus often appear as views in intro patterns, as described in section 3.2.2. Actually, view intro patterns are named after reflection views because this feature of the tactic language was originally designed for what is now the special case of reflection views. For instance, suppose that one wants to access the components of a conjunctive hypothesis, stated as a boolean conjunction. We can use lemma `andP` in a view intro-pattern in this way:

```
Lemma example n m k : k <= n ->
  (n <= m) && (m <= k) -> n = k.
Proof.
move=> lekn /andP.
```

```
n, m, k : nat
lekn : k <= n
============================
 n <= m /\ m <= k -> n = k
```

The view intro-pattern `/andP` has *applied* the reflection view `andP` to the top entry of the stack `(n <= m) && (m <= k)` and transformed it into its equivalent form `(n <= m) /\ (m <= k)`. Note that strictly speaking, lemma `andP` does not have the shape of an implication, which can be fed with a proof of its premise: it is (isomorphic to) the conjunction of *two* such implications. The *view mechanism* implemented in the tactic language has automatically guessed and inserted a term, called *hint view*, which plays the role of an adapter.

More precisely the `/andP` intro pattern has wrapped the top stack item, called `top` here, of type `((n <= m) && (m <= k))` into `(elimTF andP top)` obtaining a term of type `((n <= m) /\ (m <= k))`.

```
Lemma elimTF (P Q : Prop) (b c : bool) :
  reflect P b -> b = c -> if c then P else ~ P.
```

Term `(elimTF andP top)` hence has type

```
if true then (n <= m) /\ (m <= k) else ~ ((n <= m) /\ (m <= k))
```

which reduces to `((n <= m) /\ (m <= k))` since `c` is `true` (recall the hidden "`.. = true`" in the type of the top stack entry).

Going back to our example: we can then chain this view with a case intro-pattern to break the conjunction and introduce its components:

```
Lemma example n m k : k <= n ->
  (n <= m) && (m <= k) -> n = k.
Proof.
move=> lekn /andP[lenm lemk].
```

```
n, m, k : nat
lekn : k <= n
lenm : n <= m
lemk : m <= k
============================
n = k
```

As `(n <= m)` is by definition `(n - m == 0)`, we can use the reflection view `eqnP` in order to transform this hypothesis into a proper equation. Observe the new shape of the `lenm` hypothesis:

```
Lemma example n m k : k <= n ->
  (n <= m) && (m <= k) -> n = k.
Proof.
move=> lekn /andP [/eqnP lenm lemk].
```

```
n, m, k : nat
lekn : k <= n
lenm : n - m = 0
lemk : m <= k
============================
n = k
```

Advice

Combining wisely the structured reasoning of inductive predicates in `Prop` with the ease to reason by equivalence via rewriting of boolean identities leads to concise proofs.

Let us now move to a non-artificial example to see how the Ssreflect tactic language supports the combination of views with the `apply`, `case` and `rewrite` tactics.

4.1.4 Using views with tactics

We dissect the proof that `<=` is a total relation. As usual the statement is expressed as a boolean formula:

```
Lemma leq_total m n : (m <= n) || (m >= n).
```

The first step of the proof is to view this disjunction as an implication, using the classical equivalence and a negated premise:

```
Lemma leq_total m n : (m <= n) || (m >= n).
Proof.
rewrite -implyNb.
```

```
m, n : nat
======================
~~ (m <= n) ==> (n <= m)
```

This premise can be seen as `n < m`:

```
Lemma leq_total m n : (m <= n) || (m >= n).
Proof.
rewrite -implyNb -ltnNge.
```

```
m, n : nat
======================
(n < m) ==> (n <= m)
```

This is now an instance of the weakening property of the comparison, except that it is expressed with a boolean implication. But the view mechanism not only exists in intro-patterns: it can also be used in combination with the `apply` tactic, to apply a view to a given goal with a minimal amount of bureaucracy:

```
Lemma leq_total m n : (m <= n) || (m >= n).
Proof.
rewrite -implyNb -ltnNge; apply/implyP.
```

```
m, n : nat
=====================
(n < m) -> (n <= m)
```

We can now conclude the proof:

```
Lemma leq_total m n : (m <= n) || (m >= n).
Proof. by rewrite -implyNb -ltnNge; apply/implyP; apply: ltnW. Qed.
```

The `case` tactic also combines well with the view mechanism, which eases reasoning by cases along a disjunction expressed with a boolean statement, like the just proved `leq_total`. For example we may want to start the proof of the following lemma by distinguishing two cases: `n1 <= n2` and `n2 <= n1`.

```
Lemma leq_max m n1 n2 :
  (m <= maxn n1 n2) = (m <= n1) || (m <= n2).
Proof.
case/orP: (leq_total n2 n1) => [le_n21 | le_n12].
```

That results in:

```
m, n1, n2 : nat
le_n21 : n2 <= n1
============================
(m <= maxn n1 n2) = (m <= n1) || (m <= n2)

subgoal two is:
 (m <= maxn n1 n2) = (m <= n1) || (m <= n2)
```

Even if it is not displayed here, subgoal two has `(n1 <= n2)` in its context.

Finally, the `rewrite` tactic also handles views that relate an equation in `Prop` with a boolean formula.

```
Lemma maxn_idPl {m n} : reflect (maxn m n = m) (m >= n).

Lemma leq_max m n1 n2 :
  (m <= maxn n1 n2) = (m <= n1) || (m <= n2).
Proof.
case/orP: (leq_total n2 n1) => [le_n21 | le_n12].
  rewrite (maxn_idPl le_n21).
```

The tactic sees `(maxn_idPl le_n21)` as the equation corresponding to the boolean formula `le_n21`, namely `(maxn n1 n2 = n1)`, and rewrites with it obtaining:

```
  m : nat
  n1 : nat
  n2 : nat
  le_n21 : n2 <= n1
  ============================
   (m <= n1) = (m <= n1) || (m <= n2)

subgoal 2 is:
 (m <= maxn n1 n2) = (m <= n1) || (m <= n2)
```

The full proof of `leq_max` is quite interesting and will be detailed in section 4.3.2

4.2 Advanced, practical, statements

As we have already hinted previously, the shape of a lemma is very important. There are many ways to express the same concept, but, unsurprisingly, one can be easier to access than the others. In particular there are classes of lemmas that *specify* a concept, and as a consequence shape the proofs involving such notion.

4.2.1 Inductive specs with indices

What we did for `reflect`, an ad hoc connective, to model a line of reasoning, is a recurrent pattern in the Mathematical Components library. Such class of inductive predicates is called "spec", for *specification*.

Spec predicates are inductive families with indexes, exactly as the `eq` predicate seen in section 3.1.3. In particular their elimination rule encapsulates the notion of substitution, and that operation is performed automatically by the logic.

If we look at `reflect`, and its use, one is very likely to substitute `b` for its value in each branch of the case.

```
1  Inductive reflect (P : Prop) (b : bool) : Prop :=
2  | ReflectT (p : P)    (e : b = true)
3  | ReflectF (np : ~ P) (e : b = false)
```

This alternative formulation makes the equation implicitly stated and also automatically substituted during case analysis.

```
1  Inductive reflect (P : Prop) : bool -> Prop :=
2  | ReflectT (p : P)    : reflect P true
3  | ReflectF (np : ~ P) : reflect P false
```

Here the second argument of `reflect` is said to be an *index* and it is allowed to vary depending on the constructor: `ReflectT` always builds a term of type `(reflect P true)` while `ReflectF` builds a term of type `(reflect P false)`. When one reasons by cases on a term of type `(reflect P b)` he obtains two proof branches, in

the first one `b` is replaced by `true`, since it corresponds to the `ReflectT` constructor. Conversely, `b` is replaced by `false` in the second branch.

Let's take an example where we reason by cases on the `andP` lemma, that states `(∀ a b, reflect (a /\ b) (a && b))`.

```
Lemma example a b :
  a && b ==> (a == b).
Proof.
case: andP => [ab|nab].
```

```
a, b : bool
ab : a /\ b
========================
true ==> (a == b)

subgoal 2 is:
false ==> (a == b)
```

Remark how the automatic substitution trivializes the second goal. The first one can be solved by replacing both `a` and `b` by their truth values, once `ab` is destructed. Hence the full proof script is:

```
Lemma example a b : a && b ==> (a == b).
Proof. by case: andP => [[-> ->] |]. Qed.
```

Note that we have not specified a value for the variables quantified in the statement of `andP`. Such a view is in fact accompanied by an implicit argument declaration as follows:

```
Arguments andP {a b}.
```

We recall that this makes `a` and `b` implicit, hence writing `andP` is equivalent to `(@andP _ _)` whose type is `(reflect (_ /\ _) (_ && _))`. The value of the index of the inductive family to be replaced by `true` or `false` is here a pattern `(_ && _)` and the goal is searched for an instance of such pattern by the same matching algorithm the `rewrite` tactic uses for rewrite rules. Tuning of implicit arguments is key to obtaining easy to use lemmas, even more the "spec" ones.

If one needs to override the pattern inferred for the index of `andP`, he can provide one by hand as follows:

```
Lemma example a b : (a || ~ a) && (a && b ==> (a == b)).
Proof. by case: (a && _) / andP => [[-> ->] |] //; rewrite orbN. Qed.
```

A more detailed explanation of this syntax can be found in [13, section 5.6].

The Mathematical Components library provides many spec lemmas to be used this way. A paradigmatic one is `ifP`.

```
Section If.
Variables (A : Type) (vT vF : A) (b : bool).

Inductive if_spec : bool -> A -> Type :=
| IfSpecTrue  (p : b)         : if_spec true vT
| IfSpecFalse (p : b = false) : if_spec false vF.

Lemma ifP : if_spec b (if b then vT else vF).
```

Reasoning by cases on `ifP` has the following effects: 1) the goal is searched for an expression like `(if _ then _ else _)`; 2) two goals are generated, one in

which the condition of the if statement is replaced by `true` and the if-then-else expression by the value of the then branch, another one where the condition is replaced by `false` and the if-then-else by the value of the else branch; 3) the first goal gets an extra assumption `(b = true)`, while the second goal gets `(b = false)`. Note that "`case: ifP`" is very compact, much shorter than any if-then-else expression.

It is worth mentioning the convenience lemma `boolP` that takes a boolean formula and reasons by excluded middle providing some extra comfort like an additional hypothesis in each sub goal.

Another reflection view worth mentioning is `leqP` that replaces, in one shot, both `(_ <= _)` and the converse `(_ < _)` by opposite truth values. Sometime a proof works best by splitting into three branches, i.e., separating the equality case. The `ltngtP` lemma is designed for that.

In practice lines of reasoning consisting in a specific branching of a proof can often be modelled by an appropriate spec lemma.

Advice

The structure of the proof shall not be driven by the syntax of the definition/predicate under study but by the view/spec used to reason about it

4.3 Real proofs, finally!

So far we've only tackled simple lemmas; most of them did admit a one line proof. When proofs get longer *structure* is the best ally in making them readable and maintainable. Structuring proofs means identifying intermediate results, factor similar lines of reasoning (e.g., symmetries), signal crucial steps to the reader, and so on. In short, a proof written in Coq should not look too different from a proof written on paper.

The first subsection introduces the `have` tactic, that is the key to structure proofs into intermediate steps. The second subsection deals with the "problem" of symmetries. The third one uses most of the techniques and tactics seen so far to prove correct the Euclidean division algorithm. The three subsections contain material in increasing order of difficulty.

4.3.1 Primes, a never ending story

Saying that primes are infinite can be phrased as: for any natural number m, there exists a prime greater than m. The proof of this claim goes like that: every natural number greater than 1 has at least one prime divisor. If we take $m!+1$, then such prime divisor p can be shown to be greater than m as follows. By contraposition we assume $p \leq m$ and we show that p does not divide $m!+1$. Being smaller than m, p divides $m!$, hence to divide $m!+1$, p should divide 1; that is not possible since p is prime, hence greater than 1. □

We first show that any positive number smaller than n divides $n!$.

```
1  Lemma dvdn_fact m n : 0 < m <= n -> m %| n`!.
2  Proof.
3  case: m => //= m; elim: n => //= n IHn; rewrite ltnS leq_eqVlt.
4  by case/orP=> [/eqP-> | /IHn]; [apply: dvdn_mulr | apply: dvdn_mull].
5  Qed.
```

After the first line the proof state is the following one:

```
m, n : nat
IHn : m < n -> m.+1 %| n`!
========================
(m == n) || (m < n) -> m.+1 %| (n.+1)`!
```

The case analysis rules out `(m = 0)`, and simplifies the hypothesis to `(m <= n)`. Recall that `(x <= y <= z)` is a notation for `((x <= y) && (y <= z))`; hence when the first inequality evaluates to true (e.g. when `x` is 0) the conjunction simplifies to the second conjunct. The `leq_eqVlt` rewrite rule rephrases `<=` as a disjunction (the capital `V` letter is reminiscent of $\vee$).

When we reason by cases on the top assumption, line 4, we face two goals, both easy to solve if we look at the development of the factorial of `n.+1`, i.e., `(n.+1 * n`!)`. The former amounts to showing that `(n.+1 %| n.+1 * n`!)`, while the latter to showing that `(m.+1 %| n.+1 * n`!)` under the (inductive) hypothesis that `(m.+1 %| n`!)`.

What is paradigmatic in this little proof is the use of the *goal stack as a*

work space. In other words the proof script would be much more involved if we started by introducing in the proof context all assumptions.

We can now move to the proof of the main result. We state it using a "synonym" of the exists quantifier that is specialized to carry two properties. This way the statement is simpler to destruct: with just one case analysis we obtain the witness and the two properties. We also resort to the following lemmas.

Tools

```
Lemma fact_gt0 n : 0 < n`!.
Lemma pdivP n : 1 < n -> exists2 p, prime p & p %| n,
Lemma dvdn_addr m d n : d %| m -> (d %| m + n) = (d %| n).
Lemma gtnNdvd n d : 0 < n -> n < d -> (d %| n) = false.
```

The first step is to prove that $m! + 1$ is greater than 1, a triviality. Still it gives us the occasion to explain the `have` tactic, which lets us augment the proof context with a new fact, typically an intermediate step of our proof.

```
Lemma prime_above m : exists2 p, m < p & prime p.
Proof.
have m1_gt1: 1 < m`! + 1.
  by rewrite addn1 ltnS fact_gt0.
```

Its syntax is similar to the one of the `Lemma` command: it takes a name, a statement and starts a (sub) proof. Since the proof is so short, we will put it on the same line, and remove the full stop.

The next step is to use the `pdivP` lemma to gather a prime divisor of `m`!.+1`. We end up with the following, rather unsatisfactory, script.

```
Lemma prime_above m : exists2 p, m < p & prime p.
Proof.
have m1_gt1: 1 < m`! + 1 by rewrite addn1 ltnS fact_gt0.
case: (pdivP m1_gt1) => [p pr_p p_dv_m1].
```

It is unsatisfactory because in our paper proof what plays an interesting role is the `p` that we obtain in the second line, and not the `m1_gt1` fact we proved as an intermediate fact.

We can resort to the flexibility of `have` to obtain a more pertinent script: the first argument to `have`, here a name, can actually be any introduction pattern, i.e. what follows the `=>` operator, for example a view application. At the light of that, the script can be rearranged as follows.

```
Lemma prime_above m : exists2 p, m < p & prime p.
Proof.
have /pdivP[p pr_p p_dv_m1]: 1 < m`! + 1 by rewrite addn1 ltnS fact_gt0.
exists p => //; rewrite ltnNge; apply: contraL p_dv_m1 => p_le_m.
by rewrite dvdn_addr ?dvdn_fact ?prime_gt0 // gtnNdvd ?prime_gt1.
Qed.
```

Here the first line obtains a prime `p` as desired, the second one begins to show it fits by contrapositive reasoning, and the third one, already commented

in section 2.3.3, concludes.

As a general principle, in the proof script style we propose, a full line should represent a meaningful reasoning step (for a human being).

4.3.2 Order and max, a matter of symmetry

It is quite widespread in paper proofs to appeal to the reader's intelligence pointing out that a missing part of the proof can be obtained by symmetry. The worst thing one can do when formalizing such an argument on a computer is to use the worst invention of computer science: copy-paste. The language of Coq is sufficiently expressive to model symmetries, and the Ssreflect proof language provides facilities to write symmetric arguments.

We prove the following characterization of the max of two natural numbers:

$$\forall n_1, n_2, m, \quad m \leq \max(n_1, n_2) \Leftrightarrow m \leq n_1 \text{ or } m \leq n_2$$

The proof goes as follows: Without loss of generality we can assume that n_2 is greater or equal to n_1, hence n_2 is the maximum between n_1 and n_2. Under this assumption it is sufficient to check that $m \leq n_2$ holds iff either $m \leq n_2$ or $m \leq n_1$. The only non-trivial case is when we suppose $m \leq n_1$ and we need to prove $m \leq n_2$, which holds by transitivity. □

As usual we model double implication as an equality between two boolean expressions:

```
1  Lemma leq_max m n1 n2 : (m <= maxn n1 n2) = (m <= n1) || (m <= n2).
```

The proof uses the following lemmas. Pay attention to the premise of `orb_idr`, which is an implication.

Tools

```
1  Lemma orb_idr (a b : bool) : (b -> a) -> a || b = a.
2  Lemma maxn_idPl {m n} : reflect (maxn m n = m) (n <= m).
3  Lemma leq_total m n : (m <= n) || (n <= m).
```

Our first attempt takes no advantage of the symmetry argument: we reason by cases on the order relation, we name the resulting fact on the same line (it eases tracking where things come from) and we solve the two goals independently.

```
1  Proof.
2  case/orP: (leq_total n2 n1) => [le_n21|le_n12].
3    rewrite (maxn_idPl le_n21) orb_idr // => le_mn2.
4    by apply: leq_trans le_mn2 le_n21.
5  rewrite maxnC orbC.
6  rewrite (maxn_idPl le_n12) orb_idr // => le_mn1.
7  by apply: leq_trans le_mn1 le_n12.
8  Qed.
```

After line 2, the proof status is the following one:

Output line 2

```
2 subgoals
m, n1, n2 : nat
le_n21 : n2 <= n1
========================
(m <= maxn n1 n2) = (m <= n1) || (m <= n2)

subgoal 2 is:
(m <= maxn n1 n2) = (m <= n1) || (m <= n2)
```

The first goal is simplified by rewriting with `maxn_idPl` (as we did in section 4.1.4). Then `orb_idr` trivializes the main goal and generates a side condition with an extra hypothesis we name `le_mn2`.

Output before line 3

```
2 subgoals
m, n1, n2 : nat
le_n21 : n2 <= n1
========================
(m <= n1) = (m <= n1) || (m <= n2)

subgoal 2 is: ...
```

Output after 3

```
2 subgoals
m, n1, n2 : nat
le_n21 : n2 <= n1
le_mn2 : m <= n2
========================
m <= n1

subgoal 2 is: ...
```

Line 4 combines by transitivity the two hypotheses to conclude. Since it closes the proof branch we use the prefix `by` to asserts the goal is solved and visually signal the end of the paragraph. Line 5 commutes `max` and `||`. We can then conclude by copy-paste.

To avoid copy-pasting, shrink the proof script and finally make the symmetry step visible we can resort to the `have` tactic. In this case the statement is a variation of what we need to prove. Remark that as for `Lemma`, we can place parameters, `x` and `y` here, before the `:` symbol.

```
1 Lemma leq_max m n1 n2 : (m <= maxn n1 n2) = (m <= n1) || (m <= n2).
2 Proof.
3 have th_sym x y: y <= x -> (m <= maxn x y) = (m <= x) || (m <= y).
4   move=> le_yx; rewrite (maxn_idPl le_yx) orb_idr // => le_my.
5   by apply: leq_trans le_my le_yx.
6 by case/orP: (leq_total n2 n1) => /th_sym; last rewrite maxnC orbC.
7 Qed.
```

The proof for the `th_sym` sub proof is the text we copy-paste in the previous script, while here it is factored out. Once we have such extra fact in our context we reason by cases on the order relation and we conclude. Remark that the last line instantiates `th_sym` *in each branch* using the corresponding hypothesis on `n1` and `n2` generated by the case analysis. As expected the two instances are symmetric.

```
2 subgoals
m, n1, n2 : nat
th_sym : ∀ x y : nat,
         y <= x -> (m <= maxn x y) = (m <= x) || (m <= y)
========================
(m <= maxn n1 n2) = (m <= n1) || (m <= n2) ->
(m <= maxn n1 n2) = (m <= n1) || (m <= n2)

subgoal 2 is:
(m <= maxn n2 n1) = (m <= n2) || (m <= n1) ->
(m <= maxn n1 n2) = (m <= n1) || (m <= n2)
```

This is exactly what is needed in the first branch of the case analysis. The last subgoal just requires commuting `max` and `||`.

We can further improve the script. For example we could rephrase the proof putting in front the justification of the symmetry, and then prove one case when we pick x to be smaller than y.

```
1 Lemma leq_max m n1 n2 : (m <= maxn n1 n2) = (m <= n1) || (m <= n2).
2 Proof.
3 suff th_sym x y: y <= x -> (m <= maxn x y) = (m <= x) || (m <= y).
4   by case/orP: (leq_total n2 n1) => /th_sym; last rewrite maxnC orbC.
5 move=> le_yx; rewrite (maxn_idPl le_yx) orb_idr // => le_my.
6 by apply: leq_trans le_my le_yx.
7 Qed.
```

The `suff` tactic (or `suffices`) is like `have` but swaps the two goals.

Note that here the sub proof is now the shortest paragraph. This is another recurrent characteristic of the proof script style we adopt in the Mathematical Components library.

There is still a good amount of repetition in the current script. In particular the main conjecture has been almost copy-pasted in order to invoke `have` or `suff`. When this repetition is a severe problem, i.e. the statement to copy is large, one can resort to the `wlog` tactic (or `without loss`).

```
1 Lemma leq_max m n1 n2 : (m <= maxn n1 n2) = (m <= n1) || (m <= n2).
2 Proof.
3 wlog le_n21: n1 n2 / n2 <= n1  => [th_sym|].
4   by case/orP: (leq_total n2 n1) => /th_sym; last rewrite maxnC orbC.
5 rewrite (maxn_idPl le_n21) orb_idr // => le_mn2.
6 by apply: leq_trans le_mn2 le_n21.
7 Qed.
```

Remark how `wlog` only needs the statement of the extra assumption, and which portion of the context needs to be abstracted, here `n1` and `n2`. The sub goal to be proved is the following one.

```
2 subgoals
m, n1, n2 : nat
th_sym : ∀ n1 n2 : nat, n2 <= n1 ->
           (m <= maxn n1 n2) = (m <= n1) || (m <= n2)
========================
(m <= maxn n1 n2) = (m <= n1) || (m <= n2)


subgoal 2 is:
(m <= maxn n1 n2) = (m <= n1) || (m <= n2)
```

To keep the script similar to the previous one, we named explicitly `th_sym`, to better link the final script to the previous attempts. This is rarely the case in proof scripts of the library, since one typically uses the `/(_ ...)` intro pattern to specialize the top of the stack.

Shrinking proof scripts is a never ending game. The impatient reader can jump to the next section to see how intro patterns can be used to squeeze the last two lines into a single one. In the end, this proof script consists of three steps: the remark that we can assume `(n2 <= n1)` without losing generality; its justification in terms of totality of the order relation and commutativity of `max` and `||`; and the final proof, by transitivity, in the case when the `max` is `n1` due to the extra assumption.

Partially applied views

A less important, but very widespread, feature of the Ssreflect proof language can be used to shrink the proof even further. In this proof script, we have named the fact `le_mn2` only for the purpose of referring to this fact in the transitivity step.

```
1 Lemma leq_max m n1 n2 : (m <= maxn n1 n2) = (m <= n1) || (m <= n2).
2 Proof.
3 wlog le_n21: n1 n2 / n2 <= n1 => [th_sym|].
4   by case/orP: (leq_total n2 n1) => /th_sym; last rewrite maxnC orbC.
5 by rewrite (maxn_idPl le_n21) orb_idr // => /leq_trans->.
6 Qed.
```

Remember that the statement of `leq_trans` is `(∀ c a b, a <= c -> c <= b -> a <= b)` and we have used it to transform the top assumption `(m <= n2)`. Note that the `leq_trans` expects a second proof argument, and that its type would fix `b`, that is otherwise unspecified. If one puts a full stop just before the terminating `->`, to see the output of the view application, one sees the following stack:

```
1 (∀ n, n1 <= n -> m <= n) -> m <= n1.
```

Rewriting the top assumption fixes `n` to `n1`, trivializes the goal `(m <= n1)` to `true` and opens a trivial side condition `(n2 <= n1)`.

The very compact idiom `/leq_trans->` is quite frequent in the Mathematical Components library.

4.3.3 Euclidean division, simple and correct

Euclidean division is defined as one expects: iterating subtraction.

```
1  Definition edivn_rec d :=
2    fix loop m q := if m - d is m'.+1 then loop m' q.+1 else (q, m).
3
4  Definition edivn m d := if d > 0 then edivn_rec d.-1 m 0 else (0, m).
```

The `fix` keyword lets one write a recursive function locally, without providing a global name as `Fixpoint` does. This also means that `d` is a parameter of `edivn_rec` that does not change during recursion. The `edivn` program handles the case of a null divisor, producing the dummy pair `(0,m)` for the quotient and the reminder respectively.

We start by showing the following equation.

```
1  Lemma edivn_recE d m q :
2   edivn_rec d m q = if m - d is m'.+1 then edivn_rec d m' q.+1 else (q,m).
3  Proof. by elim: m. Qed.
```

It is often useful to state and prove unfolding equations like this one. When the simplification tactic `/=` unfolds too aggressively, rewriting with such equations gives better control on how many unfold steps one performs.

The statement of our theorem uses a let-in construct (remark the `:=` sign) to name an expression used multiple times, in this case the result of the division of `m` by `d`.

```
1  Lemma edivnP m d (ed := edivn m d) :
2    ((d > 0) ==> (ed.2 < d)) && (m == ed.1 * d + ed.2).
3  Proof.
```

As one expects, `edivn` being a recursive program, its specification needs to be proved by induction. Given that the recursive call is on the subtraction of the input, we need to perform a strong induction, as we did for the postage example in section 3.2.4.

Bet let's start by dealing with the trivial case of a null divisor.

```
1  case: d => [//=|d /=] in ed *.
2  rewrite -[edivn m d.+1]/(edivn_rec d m 0) in ed *.
3  rewrite -[m]/(0 * d.+1 + m).
4  elim: m {-2}m 0 (leqnn m) @ed => [[]//=|n IHn [//=|m]] q le_mn.
5  rewrite edivn_recE subn_if_gt; case: ifP => [le_dm|lt_md]; last first.
6    by rewrite /= ltnS ltnNge lt_md eqxx.
7  have /(IHn _ q.+1) : m - d <= n by rewrite (leq_trans (leq_subr d m)).
8  by rewrite /= mulSnr -addnA -subSS subnKC.
9  Qed.
```

Line 1 handles the case of `d` being zero. The "`in E...`" suffix can be appended to any tactic in order to push on the stack the specified hypotheses before running the tactic and pulling them back afterwards (see [13, section 6.5]). The `*` means that the goal is also affected by the tactic, and not just the hypotheses explicitly selected.

Lines 2 and 3 prepare the induction by unfolding the definition of `edivn` (to

expose the initial value of the accumulators of `edivn_rec`) and makes the invariant of the division loop explicit replacing `m` by `(0 * d.+1 + m)`. Recall the case `d` being `0` has already been handled.

Line 4 performs a strong induction, also generalizing the initial value of the accumulator `0`, leading to the following goal:

```
d, n : nat
IHn : ∀ m n0 : nat, m <= n ->
      let ed := edivn_rec d m n0 in
        (ed.2 < d.+1) && (n0 * d.+1 + m == ed.1 * d.+1 + ed.2)
m, q : nat
le_mn : m < n.+1
========================
let ed := edivn_rec d m.+1 q in
  (ed.2 < d.+1) && (q * d.+1 + m.+1 == ed.1 * d.+1 + ed.2)
```

Note that the induction touches variables used in `ed` that is hence pushed on the goal stack. The `@` modifier tells Ssreflect to keep the body of the let-in.

Line 5 unfolds the recursive function and uses the following lemma to push the subtraction into the branches of the if statement. Then it reasons by cases on the guard of the if-then-else statement.

```
1  Lemma subn_if_gt T m n F (E : T) :
2    (if m.+1 - n is m'.+1 then F m' else E) =
3      (if n <= m then F (m - n) else E).
```

The else branch corresponds to the non-recursive case of the division algorithm and is trivially solved in line 6. The recursive call is done on `(m-d)`, hence the need for a strong induction. In order to satisfy the premise of the induction hypothesis, line 7 shows that `(m - d <= n)`. Line 8 concludes.

4.4 Notational aspects of specifications

When a typing error arises, it always involves three objects: a term `t`, its type `ity` and the type expected by its context `ety`. Of course, for this situation to be an error, the two types `ity` and `ety` do not compare as equal. The simplest way one has to explain Coq how to fix `t`, is to provide a functional term `c` of type `(ity -> ety)` that is inserted around `t`. In other words, whenever the user writes `t` in a context that expects a term of type `ety`, the system instead of raising an error replaces `t` by `(c t)`.

A function automatically inserted by Coq to prevent a type error is called *coercion*. The most pervasive coercion in the Mathematical Components library is `is_true` one that lets one write statements using boolean predicates.

```
1  Lemma example : prime 17.
2
3
```

Goal fully printed

```
========================
is_true (prime 17)
```

When the statement of the example is processed by Coq and it is enforced to be a type, but `(prime 17)` is actually a term of type `bool`. Early in the library

the function `is_true` is declared as a coercion from `bool` to `Prop` and hence is it inserted by Coq automatically.

```
1  Definition is_true b := b = true.
2  Coercion is_true : bool >-> Sortclass. (* Prop *)
```

Another coercion that is widely used injects booleans into naturals. Two examples follow:

```
1  Fixpoint count (a : pred nat) (s : seq nat) :=
2    if s is x :: s' then a x + count a s' else 0.
3  Lemma count_uniq_mem (s : seq nat) x :
4    uniq s -> count (pred1 x) s = has (pred1 x) s.
```

where `pred T` is a notation for the type `T -> bool` of boolean predicates.

At line number 2 the term `(a x)` is a boolean. The `nat_of_bool` function is automatically inserted to turn `true` into `1` and `false` into `0`. This notational trick is reminiscent of Kronecker's δ notation. Similarly, in the last line the membership test is turned into a number, that is shown to be equivalent to the count of any element in a list that is duplicate free.

Coercions are composed transitively.

```
1  Definition zerolist n := mkseq (fun _ => 0) n.
2  Coercion zerolist : nat >-> seq.
3  Check 2 :: true == [:: 2; 0].
```

For the convenience of the reader we list here the most widely used coercions. There are also a bunch on Funclass not listed and elimT surely deserves some explanation.

Coercions

coercion	source	target
`Posz`	`nat`	`int`
`nat_of_bool`	`bool`	`nat`
`elimT`	`reflect`	`Funclass`
`isSome`	`option`	`bool`
`is_true`	`bool`	`Sortclass`

The reader already familiar with the concept of coercion may find the presentation of this chapter nonstandard. Coercions are usually presented as a device to model subtyping in a theory that, like the Calculus of Inductive Constructions, does not feature subtyping. As we will see in chapter 7 the role played by coercions in the modelling of the hierarchy of algebraic structures is minor. The job of coercions in that context is limited to forgeting some fields of a structure to obtain a simpler one, and that is easy. What is hard is to reconstruct the missing fields of a structure or compare two structures finding the minimum super structure. These tasks are mainly implemented with programming type inference.

4.5 Exercises

Exercise 12. *reflect*

Prove the following lemmas. In particular prove the first one with and without using `iffP`.

```
Lemma iffP_lr (P : Prop) (b : bool) :
  (P -> b) -> (b -> P) -> reflect P b.
Lemma iffP_rl (P : Prop) (b : bool) :
  reflect P b -> ((P -> b) /\ (b -> P)).
```

Exercise 13. *eqnP*

Finish this proof.

```
Lemma eqnP n m : reflect (n = m) (eqn n m).
Proof.
apply: (iffP idP).
```

Exercise 14. *Injectivity to nat*

Finish this proof.

```
Lemma nat_inj_eq T (f : T -> nat) x y :
  injective f -> reflect (x = y) (eqn (f x) (f y)).
Proof.
move=> f_inj; apply: (iffP eqnP).
```

Exercise 15. *Characterization of max*

Prove the following lemma.

```
Lemma maxn_idPl m n : reflect (maxn m n = m) (m >= n).
```

4.5.1 Solutions

Answer of Exercise 12

```
Lemma iffP_lr (P : Prop) (b : bool) :
  (P -> b) -> (b -> P) -> reflect P b.
Proof.
by case: b => [_ H|H _]; [left; apply H | right=> p; move: (H p)].
(* with iffP: by move=> *; apply: (iffP idP). *)
Qed.

Lemma iffP_rl (P : Prop) (b : bool) :
  reflect P b -> ((P -> b) /\ (b -> P)).
Proof. by case: b; case=> p; split. Qed.
```

Answer of Exercise 13

```
Lemma eqnP n m : reflect (n = m) (eqn n m).
Proof.
apply: (iffP idP) => [|->]; last by elim: m.
by elim: n m => [[]|n IH [//|m] /IH ->].
Qed.
```

Answer of Exercise 14

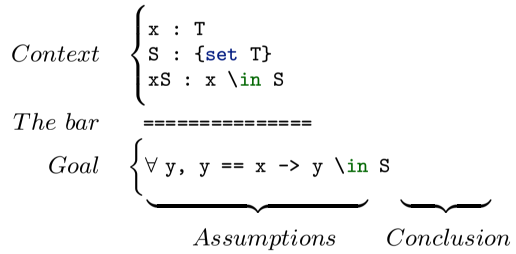
```
Lemma nat_inj_eq T (f : T -> nat) x y :
  injective f -> reflect (x = y) (eqn (f x) (f y)).
Proof. by move=> f_inj; apply: (iffP eqnP) => [/f_inj|->]. Qed.
```

Answer of Exercise 15

```
Lemma maxn_idPl m n : reflect (maxn m n = m) (m >= n).
Proof.
by rewrite -subn_eq0 -(eqn_add2l m) addn0 -maxnE; apply: eqP.
Qed.
```

Cheat Sheet

Terminology



Top is the first assumption, `y` here

Stack alternative name for the list of *Assumptions*

Popping from the stack

Note: in the following example we assume *cmd* does nothing, exactly like `move`, to focus on the effect of the intro pattern.

cmd`=> x px`

Run *cmd*, then pop Top, put it in the context naming it `x` then pop the new Top and names it `px` in the context

```
=========                  x : T
 ∀ x,                      px : P x
  P x -> Q x -> G  →      =========
                           Q x -> G
```

cmd`=> [|x xs] //`

Run *cmd*, then reason by cases on Top. In the first branch do nothing, in the second one pop two assumptions naming then `x` and `xs`. Then get rid of trivial goals. Note that, since only the first branch is trivial, one can write `=> [// | x xs]` too. caveat: Immediately after `case` and `elim` it does not perform any case analysis, but can still introduce different names in different branches

```
=========                  x : nat
 ∀ s : seq nat,            xs : seq nat
  0 < size s -> P  →       =========
    s                       0 < size (x ::
                              xs) -> P (x ::
                               xs)
```

cmd`=> /andP[pa pb]`

Run *cmd*, then apply the view `andP` to Top, then destruct the conjunction and introduce in the context the two parts naming the `pa` and `pb`

```
=========                  pa : A
 A && B -> C -> D  →       pb : B
                           =========
                           C -> D
```

cmd`=> /= {px}`

Run *cmd* then simplify the goal then discard `px` from then context

```
 x : nat
 px : P x                  x : nat
=========                 =========
 true && Q x -> R  →       Q x -> R x
    x
```

cmd`=> [y -> {x}]`

Run *cmd* then destruct the existential, then introduce y, then rewrite with Top left to right and discard the equation, then clear x

```
 x : nat
=========                  y : nat
 (exists2 y, x = y & Q y  →=========
  )                         Q y -> P y
   -> P x
```

cmd`=> /(_ x) h`

Introduce `h` specialized to `x`

```
 P : nat -> Prop
 x : nat                   P : nat -> Prop
                           x : nat
  ====================>   h : P x
                           =================
 (∀ n, P n) -> G           G
```

Pushing to the stack

Note: in the following *cmd* is not `apply` or `exact`. Moreover we display the goal just before *cmd* is run.

cmd`: (x) y`

Push `y` then push `x` on the stack. `y` is also cleared

```
 x, y : nat                 x : nat
 px : P x                   px : P x
=========                  =========
 Q x y             →        ∀ x0 y, Q x0 y
```

cmd`: {-2}x (erefl x)`

Push the type of `(erefl x)`, then push `x` on the stack binding all but the second occurrence

```
 x : nat                    x : nat
=========                  =========
 P x               →        ∀ x0,
                             x0 = x -> P x0
```

cmd`: _.+1 {px}`

Clear `px` and generalize the goal with respect to the first match of the pattern `_.+1`

```
 x : nat
 px : P x                   x : nat
=========                  =========
 x < x.+1          →        ∀ x0, x < x0
```

Proof commands

`rewrite Eab (Exc b).`

Rewrite with `Eab` left to right, then with `Exc` by instantiating the first argument with `b`

```
 Eab : a = b                Eab : a = b
 Exc : ∀ x, x = c           Exc : ∀ x, x = c
=========                  =========
 P a               →        P c
```

`rewrite -Eab {}Eac.`

Rewrite with `Eab` right to left then with `Eac` left to right, finally clear `Eac`

```
 Eab : a = b
 Eac : a = c                Eab : a = b
=========                  =========
 P b               →        P c
```

`rewrite /(_ && _).`

Unfold the definition of `&&`

```
 a : bool                    a : bool
=========                  =========
 a && a = a          →      if a then a else
                              false = a
```

`rewrite /= -[a]/(0+a)  -/c.`

Simplify the goal, then change `a` into `0+a`, finally fold back the local definition `c`

```
 a, b : nat                  a, b : nat
 c := b + 3 : nat            c := b + 3 : nat
=========                  =========
true && (a == b +      →    0 + a == c
    3)
```

`apply: H.`

Apply `H` to the current goal

```
 H : A -> B                  =========
=========                      A
 B                  →
```

`case: ab.`

Eliminate the conjunction or disjunction

```
 ab : A /\ B                 =========
=========                      A -> B -> G
 G                  →
```

`case: (leqP a b).`

Reason by cases using the `leqP` spec lemma

```
 a, b : nat      a, b : nat      a, b : nat
=========      =========      =========
 a <= b &&  →    true &&         false &&
   b > a          false           true
```

`elim: s.`

Perform an induction on `s`

```
 s : seq                     =========
   nat                       ∀ n s,
=========  →                  P s -> P (n ::
 P s              P [::]        s)
```

`elim/last_ind: s`

Start an induction on `s` using the induction principle `last_ind`

```
 s : seq                     =========
   nat                       ∀ s x,
=========  →      =========    P s ->
 P s              P [::]        P (rcons s
                                  x)
```

`have pa : P a.`

Open a new goal for `P a`. Once resolved introduce a new entry in the context for it named `pa`

```
 a : T           a : T           a : T
=========      =========       pa : P a
 G          →    P a           =========
                                 G
```

`by [].`

Prove the goal by trivial means, or fail

```
=========
 0 <= n             →
```

`exact: H.`

Apply `H` to the current goal and then assert all remaining goals, if any, are trivial. Equivalent to `by apply: H.`

```
 H : B
=========
 B                  →
```

Reflect and views

`reflect P b`

States that `P` is logically equivalent to `b`

`apply: (iffP V)`

Proves a reflection goal, applying the view lemma `V` to the propositional part of `reflect`. E.g. `apply: (iffP idP)`

```
 P : Prop          P : Prop        P : Prop
 b : bool          b : bool        b : bool
============ →   =========       =========
 reflect P b       b -> P          P -> b
```

`apply/V1/V2`

Prove boolean equalities by considering them as logical double implications. The term `V1` (resp. `V2`) is the view lemma applied to the left (resp. right) hand side. E.g. `apply/idP/negP`

```
 b1 : bool       b1 : bool       b1 : bool
 b2 : bool       b2 : bool       b2 : bool
=========  →    =========       =========
 b1 = ~~ b2      b1 -> ~ b2      ~ b2 -> b1
```

`rewrite: (eqP Eab)`

rewrite with the boolean equality `Eab`

```
 Eab : a == b      Eab : a == b
============  →   =========
 P a               P b
```

Idioms

`case: b => [h1| h2 h3]`

Push `b`, reason by cases, then pop `h1` in the first branch and `h2` and `h3` in the second

`have /andP[x /eqP->] : P a && b == c`

Open a subgoal for `P a && b == c`. When proved apply to it the `andP` view, destruct the conjunction, introduce `x`, apply the view `eqP` to turn `b == c` into `b = c`, then rewrite with it and discard the equation

`elim: n.+1 {-2}n (ltnSn n) => {n}// n`

General induction over `n`, note that the first goal has a false assumption `∀ n, n < 0 -> ...` and is thus solved by `//`

```
 n : nat                     n : nat
=========                  =========
 P n                →       (∀ m, m < n -> P m) ->
                             ∀ m, m < n.+1 -> P m
```

`rewrite lem1 ?lem2 //`

Use the equation with premises `lem1`, then get rid of the side conditions with `lem2`

Searching

`Search _ addn (_ * _)  "C"  in ssrnat`

Search for all theorems with no constraints on the main conclusion (conclusion head symbol is the wildcard `_`), that talk about the `addn` constant, matching anywhere the pattern `(_ * _)` and having a name containing the string `"C"` in the module `ssrnat`

Misc notations

```
"f1 \o f2" := (comp f1 f2)
"x \in A" := (in_mem x (mem A))
"x \notin A" := (~~ (x \in A))
"[ /\ P1 , P2 & P3 ]" := (and3 P1 P2 P3)
"[ \/ P1 , P2 | P3 ]" := (or3 P1 P2 P3)
"[ && b1 , b2 , .. , bn & c ]" :=
  (b1 && (b2 && .. (bn && c) .. ))
"[ || b1 , b2 , .. , bn | c ]" :=
  (b1 || (b2 || .. (bn || c) .. ))
"#| A |" := (card (mem A))
"n .-tuple" := (tuple_of n)
"'I_ n" := (ordinal n)
"f1 =1 f2" := (eqfun f1 f2)
"b1 (+) b2" := (addb b1 b2)
```

Notations for natural numbers: `nat`

```
"n .+1" := (succn n)
"n .-1" := (predn n)
"m + n" := (addn m n)
"m - n" := (subn m n)
"m <= n" := (leq m n)
"m < n"  := (m.+1 <= n)
"m <= n <= p" := ((m <= n) && (n <= p))
"m * n" := (muln m n)
"n .*2" := (double n)
"m ^ n" := (expn m n)
"n `!" := (factorial n)
"m %/ d" := (divn m d)
"m %% d" := (modn m d)
"m == n %[mod d ]" := (m %% d == n %% d)
"m %| d" := (dvdn m d)
"pi .-nat" := (pnat pi)
```

Notations for lists: `seq T`

```
"x :: s" := (cons _ x s)
"[ :: ]" := nil
"[ :: x1 ]" := (x1 :: [::])
"[ :: x & s ]" := (x :: s)
"[ :: x1 , x2 , .. , xn & s ]" :=
  (x1 :: x2 :: .. (xn :: s) ..)
"[ :: x1 ; x2 ; .. ; xn ]" :=
  (x1 :: x2 :: .. [:: xn] ..)
"s1 ++ s2" := (cat s1 s2)
```

Notations for iterated operations

```
"\big [ op / idx ]_ i F" :=
"\big [ op / idx ]_ ( i | P ) F" :=
"\big [ op / idx ]_ ( i <- r | P ) F" :=
"\big [ op / idx ]_ ( m <= i < n | P ) F" :=
"\big [ op / idx ]_ ( i < n | P ) F" :=
"\big [ op / idx ]_ ( i \in A | P ) F" :=
"\sum_ i F" :=
"\prod_ i F" :=
"\max_ i F" :=
"\bigcap_ i F" :=
"\bigcup_ i F" :=
```

caveat: in the general form, the iterated operation `op` is displayed in prefix form (not in infix form) caveat: the string `"big"` occurs in every lemma concerning iterated operations

Rewrite patterns

`rewrite [pat]lem [in pat2]lem2 [X in pat3]lem3`

Rewrite the subterms selected by the pattern `pat` with `lem`. Then in the subterms selected by the pattern `pat2` match the pattern inferred from the left hand side of `lem2` and rewrite the terms selected by it. Last, in the sub terms selected by `pat3` rewrite with `lem3` the sub terms identified by `X` exactly

`rewrite {3}[in X in pat1]lem1`

Like in `rewrite [X in pat1]lem1` but use the pattern inferred from `lem1` to identify the sub terms of `X` to be rewritten. Of these terms, rewrite only the third one. Example: `rewrite {3}[in X in f _ X]E`.

```
E : a = c
=========
a + f a (a + a) = f a (a + a)  →
    + a
```

```
  E : a = c
  =========
  a + f a (a + a) = f a (c + a)
+ a
```

`rewrite [e in X in pat1]lem1`

Like before, but override the pattern inferred from `lem1` with `e`

`rewrite [e as X in pat1]lem1`

Like `rewrite [X in pat1]lem1` but match `pat1[X := e]` instead of just `pat1`

`rewrite /def1 -[pat]/term /=`

Unfold all occurrences of `def1`. Then match the goal against `pat` and change all its occurrences into `term` (pure computation). Last simplify the goal

`rewrite 3?lem2 // {hyp} => x px`

Rewrite from 0 to 3 times with lem2, then try to solve with `by []` all the goals. Finally clear `hyp` and introduce `x` and `px`

Pattern matching detailed rules

pattern a term, possibly containing `_`

key The head symbol of a pattern

The sub terms selected by a pattern:

1. the goal is traversed outside in, left to right, looking for verbatim occurrences of the key
2. the sub terms whose key matches verbatim are higher order matched (i.e. up to definition unfolding and recursive function computation) against the pattern
3. if the matching fails, the next sub term whose key matches is tried
4. if the matching succeeds, the sub term is considered to be the (only) instance of the pattern
5. the sub terms selected by the pattern are then all the copies of the instance of the pattern
6. these copies are searched looking again at the key, and higher order comparing the arguments pairwise

Note: occurrence numbers can be combined with patterns. They refer to the list of of sub terms selected by the (last) pattern (i.e. they are processed at the very end).

`set n := {2 4}(_ + b)`

Put in the context a local definition named `n` for the second and fourth occurrences of the sub terms selected by the pattern `(_ + b)`

```
=========
a + c + (a + b) + (a + b)
   =
 a + (a + b) + (0 + a + b)    →
   + c
```

```
 n := a + b
=========
 a + c + (a + b) + n =
  a + (a + b) + n + c
```

Part II

Formalization Techniques

Chapter 5

Implicit Parameters

The rules of the Calculus of Inductive Constructions, as the ones sketched in 3, are expressed on the syntax of terms and are implemented by the kernel of Coq. Such software component performs *type checking*: given a term and type it checks if such term has the given type. To keep type checking simple and decidable the syntax of terms makes all information explicit. As a consequence the terms written in such verbose syntax are pretty large.

Luckily the user very rarely interacts directly with the kernel. Instead she almost always interacts with the refiner, a software component that is able to accept open terms. Open terms are in general way smaller than regular terms because some information can be left implicit [22]. In particular one can omit any subterm by writing "_" in place of it. Each missing piece of information is either reconstructed automatically by the *type inference* algorithm, or provided interactively by means of proof commands. In this chapter we focus on type inference.

Type inference is *ubiquitous*: whenever the user inputs a term (or a type) the system tries to infer a type for it. One can think of the work of the type inference algorithm as trying to give a meaning to the input of the user possibly completing and constraining it by inferring some information. If the algorithm succeeds, the term is accepted; otherwise an error is given.

What is crucial to the Mathematical Components library is that *the type inference algorithm is programmable*: one can extend the basic algorithm with small declarative programs[1] that have access to the library of already formalized facts. In this way one can make the type inference algorithm aware of the contents of the library and make Coq behave as a trained reader that is able to guess the intended meaning of a mathematical expressions from the context thanks to his background knowledge.

[1] A program is said to be declarative when it explains what it computes rather than how. The programs in question are strictly linked with the Prolog programming language, a technology that found applications in artificial intelligence and computational linguistic.

This chapter also introduces the key concepts of *interface* and *instance*. An interface is essentially the signature of an algebraic structure: operations, properties and notations letting one reason abstractly about a family of objects sharing the interface. An instance is an example of an algebraic structure, an object that fits an interface. For example `eqType` is the interface of data types that come equipped with a comparison function, and the type `nat` forms, together with the `eqn` function, an example of `eqType`.

The programs we will write to extend type inference play two roles. On one hand they link instances to interfaces, like `nat` to `eqType`. On the other hand they build *derived instances* out of basic ones. For example we teach type inference how to synthesize an instance of `eqType` for a type like `(A * B)` whenever `A` and `B` are instances of `eqType`.

The concepts of interface and instances are recurrent in both computer science and modern mathematics, but are not a primitive notion in Coq. Despite that, they can be encoded quite naturally, although not trivially, using inductive types and the dependent function space. This encoding is not completely orthogonal to the actual technology (the type inference and its extension mechanism). For this reason we shall need to dive, from time to time, into technical details, especially in sections labelled with two stars.

5.1 Type inference and Higher-Order unification

The type inference algorithm is quite similar to the type checking one: it recursively traverses a term checking that each subterm has a type compatible with the type expected by its context. During type checking types are compared taking computation into account. Types that compare as equal are said to be *convertible*. Termination of reduction and uniqueness of normal forms provide guidance for implementing the convertibility test, for which a complete and sound algorithm exists. Unfortunately type inference works on open terms, and this fact turns convertibility into a much harder problem called *higher-order unification*. The special placeholder "`_`", usually called *implicit argument*, may occur inside types and stands for one, and only one, term that is not explicitly given. Type inference does not check if two types are convertible; it checks if they unify. Unification is allowed to assign values to implicit arguments in order to make the resulting terms convertible. For example unification is expected to find an assignment that makes the type `(list _)` convertible to `(list nat)`. By picking the value `nat` for the placeholder the two types become syntactically equal and hence convertible.

Unfortunately it is not hard to come up with classes of examples where guessing appropriate values for implicit arguments is, in general, not possible. In fact such guessing has been shown to be as hard as proof search in the presence of higher-order constructs. For example to unify `(prime _)` with `true` one has to guess a prime number. Remember that `prime` is a boolean function

that fed with a natural number returns either `true` or `false`. While assigning `2` to the implicit argument would be a perfectly valid solution, it is clear that it is not the only one. Enumerating all possible values until one finds a valid one is not a good strategy either, since the good value may not exist. Just think at the problem `(prime (4 * _))` versus `true`. An even harder class of problems is the one of synthesizing programs. Take for example the unification problem `(_ 17)` versus `[:: 17]`. Is the function we are looking for the list constructor? Or maybe, is it a factorization algorithm?

Given that there is no silver bullet for higher-order unification Coq makes a sensible design choice: provide an (almost) heuristic-free algorithm and let the user extend it via an extension language. We refer to such language as the language of *canonical structures*. Despite being a very restrictive language, it is sufficient to program a wide panel of useful functionalities.

The concrete syntax for implicit arguments, an underscore character, does not let one name the missing piece of information. If an expression contains multiple occurrence of the placeholder "_", they are all considered as potentially different by the system, and hence hold (internally) unique names. For the sake of clarity we take the freedom to use the alternative syntax $?_x$ for implicit arguments (where x is a unique name).

5.2 Recap: type inference by examples

Let us start with the simplest example one could imagine: defining the polymorphic identity function and checking its application to `3`.

Polymorphic identity

```
1  Definition id (A : Type) (a : A) : A := a.
2  Check (id nat 3).
3  Check (id _ 3).
```

Response

```
id nat 3 : nat
id nat 3 : nat
```

In the expression `(id nat 3)` no subterm was omitted, therefore Coq accepted the term and printed its type. In the third line, even if the subterm `nat` was omitted, Coq accepted the term. Type inference found a value for the place holder for us by proceeding in the following way: it traversed the term recursively from left to right, ensuring that the type of each argument of the application had the type expected by the function. In particular `id` takes two arguments. The former argument is expected to have type `Type` and the user left such argument implicit (we name it $?_A$). Type inference imposes that $?_A$ has type `Type`, and this constraint is satisfiable. The algorithm continues checking the remaining argument. According to the definition of `id` the type of the second argument must be the value of the first argument. Hence type inference runs recursively on the argument `3` discovering it has type `nat` and imposes that it unifies with the value of the first argument (that is $?_A$). For this to be true $?_A$ has to be assigned the value `nat`. As a result the system prints the input term, where the place holder has been replaced by the value type inference assigned to it.

At the light of that, we observe that every time we apply the identity function to a term we can omit to specify its first argument, since Coq is able to infer it and complete the input term for us. This phenomenon is so frequent that one can ask the system to insert the right number of _ for him. For more details refer to [24, section 2.7]. Here we only provide a simple example.

Setting implicit arguments

```
1 Arguments id {A} a.
2 Check (id 3).
3 Check (@id nat 3).
```

Response

```
id 3 : nat
id 3 : nat
```

The `Arguments` directive "documents" the constant `id`. In this case it just marks the argument that has to be considered as implicit by surrounding it with curly braces. The declaration of implicit arguments can be locally disabled by prefixing the name of the constant with the `@` symbol.

Another piece of information that is often left implicit is the type of abstracted or quantified variables.

Omitting type annotations

```
1 Check (fun x => @id nat x).
2
3 Lemma prime_gt1 p : prime p -> 1 < p.
```

Response

```
fun x : nat => id x :
  nat -> nat
```

In the first line the syntax (`fun x => ...`) is sugar for (`fun x : _ => ...`) where we leave the type of `x` implicit. Type inference fixes it to `nat` when it reaches the last argument of the identity function. It unifies the type of `x` with the value of the first argument given to `id` that in this case is `nat`. This last example is emblematic: most of the times the type of abstracted variables can be inferred by looking at how they are used. This is very common in lemma statements. For example, the third line states a theorem on `p` without explicitly giving its type. Since the statement uses `p` as the argument of the `prime` predicate, it is automatically constrained to be of type `nat`.

The kind of information filled in by type inference can also be of another, more interesting, nature. So far all place holders were standing for types, but the user is also allowed to put _ in place of a term.

Inferring a term

```
1 Lemma example q : prime q -> 0 < q.
2 Proof.
3 move=> pr_q.
4 have q_gt1 := @prime_gt1 _ pr_q.
5 exact: ltnW q_gt1.
6 Qed.
```

Goal after line 3

```
1 subgoal

q : nat
pr_q : prime q
========================
0 < q
```

The proof begins by giving the name `pr_q` to the assumption (`prime q`). Then it builds a proof term by hand using the lemma stated in the previous example and names it `q_gt1`. In the expression (`prime_gt1 _ pr_q`), the place holder, that we name $?_p$, stands for a natural number. When type inference reaches $?_p$, it fixes its type to `nat`. What is more interesting is what happens when type inference reaches the `pr_q` term. Such term has its type fixed by the context:

`(prime q)`. The type of the second argument expected by `prime_gt1` is `(prime ?p)` (i.e., the type of `prime_gt1` where we substitute `?p` for `p`. Unifying `(prime ?p)` with `(prime q)` is possible by assigning `q` to `?p`. Hence the proof term just constructed is well typed, its type is `(1 < q)` and the place holder has been set to be `q`. As we did for the identity function, we can declare the `p` argument of `prime_gt1` as implicit. Choosing a good declaration of implicit arguments for lemmas is tricky and requires one to think ahead how the lemma is used.

So far we have been using only the simplest for of type inference in our interaction with the system. The unification problems we have encountered would have been solved by a first order unification algorithm and we did not need to compute or synthesize *functions*. In the next section we illustrate how the unification algorithm used in type inference can be extended in order to deal with higher-order problem. This extension is based on the use of declarative programs, and we present the encoding of the relations which describe these programs. As of today however there is no precise, published, documentation of the type inference and unification algorithms implemented in Coq. For a technical presentation of a type inference algorithm close enough to the one of Coq we suggest the interested reader to consult [1]. The reader interested in a technical presentation of a simplified version of the unification algorithm implemented in Coq can read [23, 14].

5.3 Records as relations

In computer science a record is a very common data structure. It is a compound data type, a container with named fields. Records are represented faithfully in the Calculus of Inductive Constructions as inductive data types with just one constructor, recall section 1.3.4. The peculiarity of the records we are going to use is that they are *dependently typed*: the type of each field is allowed to depend on the values of the fields that precede it.

Coq provides syntactic sugar for declaring record types.

```
1  Record eqType : Type := Pack {
2    sort : Type;
3    eq_op : sort -> sort -> bool
4  }.
```

The sentence above declares a new inductive type called `eqType` with one constructor named `Pack` with two arguments. The first one is named `sort` and holds a type; the second and last one is called `eq_op` and holds a comparison function on terms of type `sort`. We recall that what this special syntax does is declaring at once the following inductive type plus a named projection for each record field:

```
Inductive eqType : Type :=
  Pack sort of sort -> sort -> bool.
Definition sort (c : eqType) : Type :=
  let: Pack t _ := c in t.
Definition eq_op (c : eqType) : sort c -> sort c -> bool :=
  let: Pack _ f := c in f.
```

Note that the type dependency between the two fields requires the first projection to be used in order to define the type of the second projection.

We think of the `eqType` record type as a *relation* linking a data type with a comparison function on that data type. Before putting the `eqType` relation to good use we declare an inhabitant of such type, that we call an *instance*, and we examine a crucial property of the two projections just defined.

We relate the `eqn` comparison function with the `nat` data type.

```
Definition nat_eqType : eqType := @Pack nat eqn.
```

Projections, when applied to a record instance like `nat_eqType` compute and extract the desired component.

Computation of projections

```
Eval simpl in sort nat_eqType.
Eval simpl in @eq_op nat_eqType.
```

Response

```
= nat : Type
= eqn : sort nat_eqType ->
          sort nat_eqType -> bool
```

Given that `(sort nat_eqType)` and `nat` are convertible, equal up to computation, we can use the two terms interchangeably. The same holds for `(eq_op nat_eqType)` and `eqn`. Thanks to this fact Coq can type check the following term:

```
Check (@eq_op nat_eqType 3 4).
```

```
eq_op 3 4 : bool
```

This term is well typed, but checking it is not as simple as one may expect. The `eq_op` function is applied to three arguments. The first one is `nat_eqType` and its type, `eqType`, is trivially equal to the one expected by `eq_op`. The following two arguments are hence expected of to be of type `(sort nat_eqType)` but `3` and `4` are of type `nat`. Recall that unification takes computation into account exactly as the convertibility relation. In this case the unification algorithm unfolds the definition of `nat_eqType` obtaining `(Pack nat eqn)` and reduces the projection extracting `nat`. The obtained term literally matches the type of the last two arguments given to `eq_op`.

Now, why this complication? Why should one prefer `(eq_op nat_eqType 3 4)` to `(eqn 3 4)`? The answer is *overloading*. It is recurrent in mathematics and computer science to reuse a symbol, a notation, in two different contexts. A typical example coming from the mathematical practice is to use the same infix symbol $*$ to denote any ring multiplication. A typical computer science example is the use of the same infix `==` symbol to denote the comparison over any data type. Of course the underlying operation one intends to use depends on the

values it is applied to, or better their type [2]. Using records lets us model these practices. Note that, thanks to its higher-order nature, the term `eq_op` can always be the head symbol denoting a comparison. This makes it possible to recognize, hence print, comparisons in a uniform way as well as to input them. On the contrary, in the simpler expression `(eqn 3 4)`, the name of the head symbol is very specific to the type of the objects we are comparing.

In the rest of this chapter, we focus on the overloading of the `==` symbol and we start by defining another comparison function, this time for the `bool` data type.

```
1  Definition eqb (a b : bool) := if a then b else ~~ b.
2  Definition bool_eqType : eqType := @Pack bool eqb.
```

Now the idea is to define a notation that applies to any occurrence of the `eq_op` head constant and use such notation for both printing and parsing.

Overloaded notation

```
1  Notation "x == y" := (@eq_op _ x y).
2  Check (@eq_op bool_eqType true false).
3  Check (@eq_op nat_eqType 3 4).
```

Response

```
true == false : bool
3 == 4 : bool
```

As a printing rule, the place holder stands for a wild card: the notation is used no matter the value of the first argument of `eq_op`. As a result both occurrences of `eq_op`, line 2 and 3, are printed using the infix `==` syntax. Of course the two operations are different; they are specific to the type of the arguments and the typing discipline ensures the arguments match the type of the comparison function packaged in the record.

When the notation is used as a parsing rule, the place holder is interpreted as an implicit argument: type inference is expected to find a value for it. Unfortunately such notation does not work as a parsing rule yet.

```
1  Check (3 == 4).
2
```

```
Error: The term "3" has type "nat" while
it is expected to have type "sort ?e".
```

If we unravel the notation, the input term is really `(eq_op _ 3 4)`. We name the place holder $?_e$. If we replay the type inference steps seen before, the unification step is now failing. Instead of `(sort nat_eqType)` versus `nat`, now unification has to solve the problem `(sort` $?_e$`)` versus `nat`. This problem falls in one of the problematic classes we presented in section 5.1: the system has to synthesize a comparison function (or better a record instance containing a comparison function).

Coq gives up, leaving to the user the task of extending the unification algorithm with a declarative program that is able to solve unification problems of the form `(sort` $?_e$`)` versus `T` for any `T`. Given the current context, it seems reasonable to write an extension that picks `nat_eqType` when `T` is `nat` and `bool_eqType` when `T`

[2] The meaning of a symbol in mathematics is even deeper: by writing $a * b$ one may expect the reader to figure out which ring she talks about, recall its theory, and use this knowledge to justify some steps in a proof. By programming type inference appropriately, we model this practice in section 5.4.

is `bool`. In the language of `Canonical Structure`s, such a program is expressed as follows.

Declaring canonical structures

```
Canonical nat_eqType.
Canonical bool_eqType.
```

The keyword `Canonical` was chosen to stress that the program is deterministic: each type `T` is related to (at most) one *canonical* comparison function.

```
Check (3 == 4).
Check (true == false).
Eval compute in (3 == 4).
```

```
3 == 4 : bool
true == false : bool
= false : bool
```

The mechanics of the small program we wrote using the `Canonical` keyword can be explained using the global table of canonical solutions. Whenever a record instance is declared as canonical Coq adds to such table an entry for each field of the record type.

canonical structures Index

projection	value	solution
`sort`	`nat`	`nat_eqType`
`sort`	`bool`	`bool_eqType`

Whenever a unification problem with the following shape is encountered, the table of canonical solution is consulted.

$$\texttt{(projection ?}_S\texttt{)} \quad \text{versus} \quad \texttt{value}$$

The table is looked up using as keys the projection name and the value. The corresponding solution is assigned to the implicit argument $?_S$.

In the table we reported only the relevant entries. Entries corresponding to the `eq_op` projection plays no role in the Mathematical Components library. The name of such projections is usually omitted to signal that fact.

What makes this approach interesting for a large library is that record types can play the role of interfaces. Once a record type has been defined and some functionality associated to it, like a notation, one can easily hook a new concept up by defining a corresponding record instance and declaring it canonical. One gets immediately all the functionalities tied to such interface to work on the new concept. For example a user defining new data type with a comparison function can immediately take advantage of the overloaded `==` notation by packing the type and the comparison function in an `eqType` instance.

This pattern is so widespread and important that the Mathematical Components library consistently uses the synonym keyword `Structure` in place of `Record` in order to make record types playing the role of interfaces easily recognizable.

The computer-science inclined reader shall see records as first-class values in the Calculus of Inductive Constructions programming language. Otherwise said, the projections of a record are just ordinary functions, defined by pattern-matching on an inductive type, and which access the fields of the record. Exercise 1 proposed to implement the projections of a triple onto its components,

and these functions are the exact analogues of the projections of a record with three fields. In particular, the fields of two given instances of records can be combined and used to build a new instance of another record. Canonical structures provide a language to describe how new instances of records, also called structures in this case, can be built from existing ones, via a set of combinators defined by the user.

So far we have used the `==` symbol for terms whose type is atomic, like `nat` or `bool`. If we try for example to use it on terms whose type was built using a type constructor like the one of pairs we encounter an error.

Error

```
Check (3, true) == (4, false).

```

Response

```
Error: The term "(3, true)" has
     type "(nat * bool)%type"
     while it is expected to have
      type "sort ?e".
```

The term `(3,true)` has type `(nat * bool)` and, so far, we only taught Coq how to compare booleans and natural numbers, not how to compare pairs. Intuitively the way to compare pairs is to compare their components *using the appropriate comparison function*. Let's write a comparison function for pairs.

Comparing pairs

```
Definition prod_cmp eqA eqB x y :=
  @eq_op eqA x.1 y.1 && @eq_op eqB x.2 y.2.
```

What is interesting about this comparison function is that the pairs `x` and `y` are not allowed to have an arbitrary, product, type here. The typing constraints imposed by the two `eq_op` occurrences force the type of `x` and `y` to be `(sort eqA * sort eqB)`. This means that the records `eqA` and `eqB` hold a sensible comparison function for, respectively, terms of type `(sort eqA)` and `(sort eqB)`.

It is now sufficient to pack together the pair data type constructor and this comparison function in an `eqType` instance to extend the canonical structures inference machinery with a new combinator.

Recursive canonical structure

```
Definition prod_eqType (eqA eqB : eqType) : eqType :=
  @Pack (sort eqA * sort eqB) (@prod_cmp eqA eqB).
Canonical prod_eqType.
```

The global table of canonical solutions is extended as follows.

canonical structures Index

projection	value	solution	combines solutions for
`sort`	`nat`	`nat_eqType`	
`sort`	`bool`	`bool_eqType`	
`sort`	`T1 * T2`	`prod_eqType pA pB`	`pA` ⟵ (`sort`,`T1`), `pB` ⟵ (`sort`,`T2`)

The third column is empty for base instances while it contains the recursive calls for instance combinators. With the updated table, when the unification problem

`(sort ?`$_e$`)` versus `(T1 * T2)`

is encountered, a solution for `?`$_e$ is found by proceeding in the following way. Two new unification problems are generated: `(sort ?`$_{eqA}$`)` versus `T1` and `(sort ?`$_{eqB}$`)` versus `T2`. If both are successful and `v1` is the solution for `?`$_{eqA}$ and `v2` for `?`$_{eqB}$, the solution for `?`$_e$ is `(prod_eqType v1 v2)`.

After the table of canonical solutions has been extended, our example is accepted.

```
Check (3, true) == (4, false).
```

```
(3, true) == (4, false) : bool
```

The term synthesized by Coq is the following one:

```
@eq_op (prod_eqType nat_eqType bool_eqType) (3, true) (4, false).
```

5.4 Records as (first-class) interfaces

When we define an overloaded notation, we convey through it more than just the arity (or the type) of the associated operation. We associate to it a property, or a collection thereof. For example, in the context of group theory, the infix `+` symbol is typically preferred to `*` whenever the group law is commutative.

Going back to our running example, the actual definition of `eqType` used in the Mathematical Components library also contains a property which enforces the correctness and the completeness of the comparison test.

eqType

```
Module Equality.

Structure type : Type := Pack {
  sort : Type;
  op : sort -> sort -> bool;
  axiom : ∀ x y, reflect (x = y) (eq_op x y)
}.

End Equality.
```

The extra property turns the `eqType` relation into a proper *interface*, which fully specifies what `op` is.

The axiom says that the boolean comparison function is compatible with equality: two ground terms compare as equal if and only if they are syntactically equal. Note that this means that the comparison function is not allowed to quotient the type by identifying two syntactically different terms.

Advice

The infix notation `==` stands for a comparison function compatible with Leibniz equality (substitution in any context).

The `Equality` module enclosing the record acts as a name space: `type`, `sort`, `eq` and `axiom`, three very generic words, are here made local to the `Equality` name space becoming, respectively, `Equality.type`, `Equality.sort`, `Equality.op` and `Equality.axiom`.

As in section 5.3, the record plays the role of a relation and its `sort` component is again the only field that drives canonical structure inference. Following a terminology typical of object-oriented programming languages, the set of operations (and properties) that define an interface is called a *class*. In the next chapter, we are going to re-use already defined classes in order to build new ones by mixing-in additional properties (typically called axioms). Hence the definition of `eqType` in the Mathematical Components library is closer to the following one:

The real definition of eqType

```
Module Equality.

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

Record mixin_of T := Mixin {op : rel T; _ : axiom op}.
Notation class_of := mixin_of.

Structure type : Type := Pack {sort :> Type; class : class_of sort; }.

End Equality.

Notation eqType := Equality.type.
Definition eq_op T := Equality.op (Equality.class T).
Notation "x == y" := (@eq_op _ x y).
```

In this simple case, there is only one property, named `Equality.axiom`, and the class is exactly the mixin.

Said that, nothing really changes: the `eqType` structure relates a type with a signature.

Remark the use of `:>` instead of `:` to type the field called `sort`. This tells Coq to declare the `Equality.sort` projection as a coercion. This makes possible to write `(∀ T : eqType, ∀ x y : T, P)` even if `T` is not a type and only `(sort T)` is.

Warning

Since `Equality.sort` is a coercion, it is not displayed by Coq; hence error messages about a missing canonical instance declaration typically look very confusing, akin to: "... has type `nat` but should have type `?e`", instead of "... but should have type `(sort ?e)`".

Given the new definition of `eqType`, when we write `(a == b)`, type inference does not only infer a function to compare `a` with `b`, but also a proof that such

function is correct. Declaring the `eqType` instance for `nat` now requires some extra work, namely proving the correctness of the `eqn` function.

```
Lemma eqnP : Equality.axiom eqn.
Proof.
move=> n m; apply: (iffP idP) => [|<-]; last by elim n.
by elim: n m => [|n IHn] [|m] //= /IHn->.
Qed.
```

We now have all the pieces to declare `eqn` as canonical.

Making eqn canonical

```
Definition nat_eqMixin := Equality.Mixin eqnP.
Canonical nat_eqType := @Equality.Pack nat nat_eqMixin.
```

Note that the `Canonical` declaration is expanded (showing the otherwise implicit first argument of `Pack`) to document that we are relating the type `nat` with its comparison operation.

5.5 Using a generic theory

The whole point of defining interfaces is to share a theory among all examples of each interface. In other words a theory proved starting from the properties (axioms) of an interface applies to all its instances, transparently. Every lemma part of an abstract theory is *generic*: the very same name can be used for each and every instance of the interface, exactly as the `==` notation.

The simplest lemma part of the theory of `eqType` is the `eqP` generic lemma that can be used in conjunction with any occurrence of the `==` notation.

The eqP lemma

```
Lemma eqP (T : eqType) : Equality.axiom (@Equality.op T).
Proof. by case: T => ty [op prop]; exact: prop. Qed.
```

The proof is just unpacking the input `T`. We can use it on a concrete example of `eqType` like `nat`

```
Lemma test (x y : nat) : x == y -> x + y == y + y.
Proof. by move=> /eqP ->. Qed.
```

In short, `eqP` can be used to change view: turn any `==` into `=` and vice versa.

The `eqP` lemma also applies to abstract instances of `eqType`. When we rework the instance of the type `(T1 * T2)` we see that the proof, by means of the `eqP` lemma, uses the axiom of `T1` and `T2`:

The complete definition of prod_eqType

```
Section ProdEqType.
Variable T1 T2 : eqType.

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

Lemma pair_eqP : Equality.axiom pair_eq.
Proof.
move=> [x1 x2] [y1 y2] /=; apply: (iffP andP) => [[]|[<- <-]] //=.
by move/eqP->; move/eqP->.
Qed.

Definition prod_eqMixin := Equality.Mixin pair_eqP.
Canonical prod_eqType := @Equality.Pack (T1 * T2) prod_eqMixin.
End ProdEqType.
```

where notation `[rel x y : T | E]` defines a binary boolean relation on type `T`. Note that a similar notation `[pred x : T | E]` exists for unary boolean predicates.

The generic lemma `eqP` applies to any `eqType` instance, like `(bool * nat)`

```
Lemma test (x y : nat) (a b : bool) : (a,x) == (b,y) -> fst (a,x) == b.
Proof. by move=> /eqP ->. Qed.
```

The `(a,x) == (b,y)` assumption is reflected to `(a,x) = (b,y)` by using the `eqP` view specified by the user. Here we write `==` to have all the benefits of a computable function (simplification, reasoning by cases), but when we need the underlying logical property of substitutivity we access it via the view `eqP`.

```
Lemma test (x y : nat) : (true,x) == (false,y) -> false.
Proof. by []. Qed.
```

This statement is true (or better, the hypothesis is false) by computation. In this last example the use of `==` give us immediate access to reasoning by cases.

Why one should always use == (EM)

```
Lemma test_EM (x y : nat) : if x == y.+1 then x != 0 else true.
Proof. by case: ifP => // /eqP  ->. Qed.
```

Advice

Whenever we want to state equality between two expressions, if they live in an `eqType`, always use `==`.

5.6 The generic theory of sequences

Now that the `eqType` interface equips a type with a well specified comparison function we can use it to build abstract theories, for example the one of sequences.

It is worth to remark that the concept of interface is crucial to the development of such theory. If we try to develop the theory of the type `(seq T)` for an arbitrary `T`, we can't go much far. For example we can express what belonging to a sequence means, but not write a program that tests if a value is actually in the list. As a consequence we lose the former automation provided by computation and it also becomes harder to reason by cases on the membership predicate. On the contrary when we quantify a theory on the type `(seq T)` for a `T` that is an `eqType`, we recover all that. In other words, by better specifying the types parameters of a generic container, we define which operations are licit and which properties hold. So far the only interface we know is `eqType`, that is primordial to boolean reflection. In the next chapters more elaborate interfaces will enable us to organize knowledge in articulated ways.

Going back to the abstract theory of sequences over an `eqType`, we start by defining the membership operation.

Membership

```
Section SeqTheory.
Variable T : eqType.
Implicit Type s : seq T.

Fixpoint mem_seq s x :=
  if s is y :: s' then (y == x) || mem_seq s' x else false.
```

Like we did for the overloaded `==` notation, we can define the `\in` (and `\notin`) infix notation. We can then easily define what a duplicate-free sequence is, and how to enforce such property.

```
Fixpoint uniq s :=
  if s is x :: s' then (x \notin s') && uniq s' else true.
Fixpoint undup s :=
  if s is x :: s' then
    if x \in s' then undup s' else x :: undup s'
  else [::].
```

Proofs about such concepts can be made pretty much as if the type `T` was `nat` or `bool`, i.e. our predicates do compute.

undup is correct (step 1)

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

Lemma mem_undup s : undup s =i s.
Proof.
move=> x; elim: s => //= y s IHs.
case Hy: (y \in s); last by rewrite in_cons IHs.
by rewrite in_cons IHs; case: eqP => // ->.
Qed.
```

where `(A =i B)` is a synonym for `(∀ x, x \in A = x \in B)`,

The `in_cons` lemma is just a convenience rewrite rule, while `mem_undup` says that the `undup` function does not drop any non-duplicate element. Note that in the proof we use both the decidability of membership (`Hy`) and the decidability of equality (via `eqP`).

undup is correct (step 2)

```
Lemma undup_uniq s : uniq (undup s).
Proof.
by elim: s => //= x s IHs; case sx: (x \in s); rewrite //= mem_undup sx.
Qed.
```

The proof of `undup_uniq` requires no new ingredients and completes the specification of `undup`.

A last, very important step in the theory of sequences is to show that the container preserves the `eqType` interface: whenever we can compare the elements of a sequence, we can also compare sequences.

eqType for sequences

```
Fixpoint eqseq s1 s2 {struct s2} :=
  match s1, s2 with
  | [::], [::] => true
  | x1 :: s1', x2 :: s2' => (x1 == x2) && eqseq s1' s2'
  | _, _ => false
  end.

Lemma eqseqP : Equality.axiom eqseq.
Proof.
elim=> [|x1 s1 IHs] [|x2 s2] /=; do? [exact: ReflectT | exact: ReflectF].
case: (x1 =P x2) => [<-|neqx]; last by apply: ReflectF => -[eqx _].
by apply: (iffP (IHs s2)) => [<-|[]].
Qed.

Definition seq_eqMixin := Equality.Mixin eqseqP.
Canonical seq_eqType := @Equality.Pack (seq T) seq_eqMixin.
```

In this script, `(x1 =P x2)` is a notation for `(@eqP T x1 x2)`, a proof of the `reflect` inductive spec; a case analysis on the term `(x1 =P x2)` has thus two branches. Since the constructors `ReflectT` and `ReflectF` carry a proof, each branch of this analysis features an extra assumption; the branch corresponding to `ReflectT` has the hypothesis `x1 = x2` and the branch corrsponding to `ReflectF` has its negation

```
x1 != x2.
```

As an example we build a sequence of sequences, and we assert that we can use the `==` and `\in` notation on it, as well as apply the list operations and theorems on objects of type `(seq (seq T))` when `T` is an `eqType`.

```
Let s1 := [:: 1; 2 ].
Let s2 := [:: 3; 5; 7].
Let ss :  seq (seq nat) := [:: s1 ; s2 ].
Check (ss != [::]) && s1 \in ss && undup_uniq ss.
```

As we have anticipated in chapter 1, functional programming and lists can model definite, iterated operations like the "big" sum Σ. The next section describes how the generic theory of iterated operations can be built and made practical thanks again to programmable type inference.

5.7 The generic theory of "big" operators

The objective of the *bigop* library is to provide compact notations for definite iterated operations and a library of general results about them.

Let us take two examples of iterated operations:

$$\sum_{i=1}^{n} f(i) = f(1) + f(2) + \ldots + f(n) \qquad \bigcup_{a \in A} g(a) = g(a_1) \cup g(a_2) \cup \ldots \cup g(a_{|A|})$$

To share an infrastructure for this class of operators we have to identify a common pattern. First the big symbol in front specifies the operation being iterated and the neutral element for such operator. For example if A is empty, then the union of all $g(a)$ is $\emptyset$, while if $n = 0$, then the sum of all $f(i)$ is 0. Then the range is an expression identifying a finite set of elements, some times expressing an order (relevant when the iterated operation is not commutative). Finally a general term describing the items being combined by the iterated operation.

As already mentioned in section 1.6, the functional programming language provided by Coq can express in a very natural way iterations over a finite domain. In particular such finite domain can be represented as a list; the general term $f(i)$ by a function `(fun i => ...)`; the operation of evaluating a function on all the elements of a list and combining the results by the `foldr` iterator.

Functional programming can also be used to describe the finite domain. For example, the list of natural numbers $m, m+1, \ldots, m+(n-m)$ corresponding to the range $m \leq i < n$ can be built using the `iota` function as follows:

```
Definition index_iota m n := iota m (n - m).
```

The only component of typical notations for iterated operations we have not discussed yet is the filter, used to iterate the operation only on a subset of the domain. For example, to state that the sum of the first n odd numbers is n^2,

one could write:

$$\sum_{i<2n,\, 2\nmid i} i = n^2$$

An alternative writing for the same summation exploits the general terms to rule out even numbers:

$$\sum_{i<n} (i * 2 - 1) = n^2$$

While this latter writing is elegant, it is harder to support generically, since the filtering condition is not explicit. For example the following equation clearly holds for any filter, range and general term. It would be hard to express such a statement if the filter were mixed with the general term, and hence its negation were not obvious to formulate.

$$\sum_{i<2n,\, 2\mid i} i + \sum_{i<2n,\, 2\nmid i} i = \sum_{i<2n} i$$

Last, not all filtering conditions can be naturally expressed in the general term. An example is not being a prime number.

At the light of that, our formal statement concerning the sum of odd numbers is the following:

```
Lemma sum_odd n : \sum_(0 <= i < n.*2 | odd i) i = n^2.
```

where `.*2` is a postfix notation, similar to `.+1`, standing for doubling. Under the hood we expect to find the following expression:

```
Lemma sum_odd n :
  foldr (fun acc i => if odd i then i + acc else acc)
    0 (index_iota 0 n.*2)
  = n^2
```

The following section details how the generic notation for iterated operation is built and specialized to frequent operations like Σ. Section 5.7.2 focuses on the generic theory of iterated operations.

5.7.1 The generic notation for `foldr`

The generic notation for iterated operations has to be attached to something more specific than `foldr` in order to clearly identify all components

```
Definition bigop R I idx op r (P : pred I) (F : I -> R) : R :=
  foldr (fun i x => if P i then op (F i) x else x) idx r.
```

Using the `bigop` constant to express our statement leads to

```
Lemma sum_odd n :
  bigop 0 addn (index_iota 0 n.*2) odd (fun i => i) = n^2
```

Note that `odd` is already a predicate on `nat`, the general term is the identity function, the range `r` is `(index_iota 0 n.*2)`, the iterated operation `addn` and the initial value is `0`.

A generic notation can now be attached to `bigop`.

```
Notation "\big [ op / idx ]_ ( i <- r | P ) F" :=
  (bigop idx op r (fun i => P%B) (fun i => F)) : big_scope.
```

Here `op` is the iterated operation, `idx` the neutral element, `r`, the range, `P` the filter (hence the boolean scope) and `F` the general term. Using such notation the running example can be stated as follows.

```
Lemma sum_odd n : \big[addn/0]_(i <- index_iota 0 n.*2 | odd i) i = n^2.
```

To obtain a notation closer to the mathematical one, we can specialize at once the iterated operation and the neutral element as follows.

```
Local Notation "+%N" := addn (at level 0, only parsing).
Notation "\sum_ ( i <- r | P ) F" :=
  (\big[+%N/0%N]_(i <- r | P%B) F%N) : nat_scope.
```

Such a notation is placed in `nat_scope` as it is specialized to `addn` and `0`. The general term `F` is also placed in the scope of natural numbers. We can proceed even further and specialize the notation to a numerical range:

```
Notation "\big [ op / idx ]_ ( m <= i < n | P ) F" :=
  (bigop idx op (index_iota m n) (fun i : nat => P%B) (fun i => F))
     : big_scope.
Notation "\sum_ ( m <= i < n ) F" :=
  (\big[+%N/0%N]_(m <= i < n) F%N) : nat_scope.
```

We can now comfortably state the theorem about the sum of odd numbers inside `nat_scope`. The proof of this lemma is left as an exercise; we now focus on a simpler instance, for n equal to 3, to introduce the library that equips iterated operations.

```
Lemma sum_odd_3 : \sum_(0 <= i < 3.*2 | odd i) i = 3^2.
Proof.
rewrite unlock /=.
```

The `bigop` constant is "locked" to make the notation steady. To unravel its computational behavior one has to rewrite with the `unlock` lemma.

```
============================
1 + (3 + (5 + 0)) = 3^2
```

The computation behavior of `bigop` is generic; it does not depend on the iterated operation. By contrast, some results on iterated operations may depend on a particular property of the operation. For example, to pull out the last item from the summation, i.e., using the following lemma

$$\text{if } a \leq b \text{ then } \sum_{a \leq i < b+1} Fi = \sum_{a \leq i < b} Fi + Fb$$

to obtain

```
============================
1 + (3 + 0) + 5 = 3^2
```

one really needs the iterated operation, addition here, to be associative. Also note that, given the filter, what one really pulls out is `(if odd 5 then 5 else 0)`, so for the theorem to be true for any range, `0` must also be neutral.

The lemma to pull out the last item of an iterated operation is provided as the combination of two simpler lemmas called respectively `big_nat_recr` and `big_mkcond`.

The former states that one can pull out of an iterated operation on a numerical range the last element, proviso the range is non-empty.

```
Lemma big_nat_recr n m F : m <= n ->
  \big[*%M/1]_(m <= i < n.+1) F i = (\big[*%M/1]_(m <= i < n) F i) * F n.
```

Such lemma applies to any operation `*%M` and any neutral element `1` and any generic term `F`, while the filter `P` is fixed to `true` (i.e., no filter). The `big_mkcond` lemmas moves the filter into the generic term.

```
Lemma big_mkcond I r (P : pred I) F :
  \big[*%M/1]_(i <- r | P i) F i =
     \big[*%M/1]_(i <- r) (if P i then F i else 1).
```

If we chain the two lemmas we can pull out the last item.

```
Lemma sum_odd_3 :
  \sum_(1 <= i < 3.*2) i.*2 = 5 * 4
Proof.
rewrite big_mkcond big_nat_recr //=.
rewrite unlock /=.
```

```
============================
\sum_(0 <= i < 5) (if odd i then i else 0) + 5 = 3^2
```

When the last item is pulled out, we can unlock the computation and obtain the following goal:

```
============================
0 + (1 + (0 + (3 + (0 + 0)))) + 5 = 3^2
```

It is clear that for the two lemmas to be provable, one needs the associativity property of `addn` and also that `0` is neutral. In other words, the lemmas we used require the operation `*%M` to form a monoid together with the unit `1`.

We detail how this requirement is stated, and automatically satisfied by Coq in the case of `addn`, in the next section. We conclude this section by showing that the same lemmas also apply to an iterated product.

```
Lemma prod_fact_4 :
  \prod_(1 <= i < 5) i = 4`!.
Proof.
rewrite big_nat_recr //=.
```

Pulling out the last product

```
============================
\prod_(1 <= i < 4) i * 4 = 4`!
```

This is the reason why we can say that the bigop library is generic: it works

uniformly on any iterated operator, and, provided the operator has certain properties, it gives uniform access to a palette of lemmas.

5.7.2 Assumptions of a bigop lemma

As we anticipated, canonical structures can be indexed not only on types, but on any term. In particular we can index them on function symbols to relate, for example, `addn` and its monoid structure.

Here we only present the `Monoid` interface an operation has to satisfy in order to access a class of generic lemmas. Chapter 7 adds other interfaces to the picture and organizes the bigop library around them.

```
Module Monoid.
Section Definitions.
Variables (T : Type) (idm : T).

Structure law := Law {
  operator : T -> T -> T;
  _ : associative operator;
  _ : left_id idm operator;
  _ : right_id idm operator
}.
```

The `Monoid.law` structure relates the `operator` (the key used by canonical structures) to the three properties of monoids.

We can then us this interface as a parameter for a bunch of lemmas, describing the theory shared by its instances.

```
Coercion operator : law >-> Funclass.
Section MonoidProperties.
Variable R : Type.

Variable idx : R.
Local Notation "1" := idx.

Variable op : Monoid.law idx.
Local Notation "*%M" := op (at level 0).
Local Notation "x * y" := (op x y).
```

The lemma we used in the previous section, `big_nat_recr`, is stated as follows. Note that `op` is a record, and not a function, but since the `operator` projection is declared as a coercion we can use `op` as such. In particular under the hood of the expression `\big[*%M/1]` we find `\big[ operator op / idx ]`.

```
Lemma big_nat_recr n m F : m <= n ->
  \big[*%M/1]_(m <= i < n.+1) F i = (\big[*%M/1]_(m <= i < n) F i) * F n.
```

If we print such statement once the `Section MonoidProperties` is closed, we see the requirement affecting the operation `op` explicitly.

```
big_nat_recr :
  ∀ (R : Type) (idx : R) (op : Monoid.law idx) (n m : nat) (F : nat -> R),
  m <= n ->
    \big[op/idx]_(m <= i < n.+1) F i =
    op (\big[op/idx]_(m <= i < n) F i) (F n)
```

Note that wherever the operation `op` occurs, we also find the `Monoid.operator` projection.

To make this lemma available on the addition on natural numbers, we need to declare the canonical monoid structure on `addn`.

```
1  Canonical addn_monoid := Monoid.Law addnA add0n addn0.
```

This command adds the following rule to the canonical structures index:

canonical structures Index

projection	value	solution
`Monoid.operator`	`addn`	`addn_monoid`

Whenever the lemma is applied to an expression about natural numbers as

```
1  Lemma test : \sum_(0 <= i < 6) i =  \sum_(0 <= i < 5) i + 5.
2  Proof. by apply: big_nat_recr. Qed.
```

the following unification problem has to be solved: `addn` versus `(operator ?m)`. Inferring a value for `?m` mean inferring a proof that `addn` forms a monoid with `0`; this is a prerequisite for the `big_nat_recr` lemma we don't have to provide by hand.

5.7.3 Searching the bigop library

Searching the bigop library for a lemma is slightly harder than searching the other libraries as explained in section 2.5. In particular one can hardly search with patterns. For example the following search returns no results:

```
1  Search _ (\sum_(0 <= i < 0) _).
```

A lemma stating that an empty sum is zero is not part of the library. What is part of the library is a lemma that says that, if the list being folded is `nil`, then the result is the initial value. Such a lemma, called `big_nil`, thus mentions only `[::]` in its statement, and not the (logically) equivalent `(index_iota 0 0)`. Still the goal `(\sum_(0 <= i < 0) i = 0)` can be solved by `big_nil`. Finally, the pattern we provide specifies a trivial filter, while the lemma is true for any filtering predicate. Of course one can craft a pattern that finds such lemma, but it is very verbose and hence inconvenient.

```
1  Search _ (\big[_/_]_(i <- [::] | _) _).
```

The recommended way to search the library is by name, using the word "big". For example to find all lemmas allowing one to prove the equality of two iterated operators one can `Search "eq" "big"`. Similarly, induction lemmas can be found with `Search "ind" "big"`; index exchange lemmas with `Search "exchange" "big"`;

lemmas pulling out elements from the iteration with `Search "rec" "big"`; lemmas working on the filter condition with `Search "cond" "big"`, etc...

Finally the Mathematical Components user is advised to read the contents of the bigop file in order to get acquainted with the naming policy used in that library.

5.8 Stable notations for big operators (⋆⋆)

The `bigop` constant and the notations attached to it are defined in a more complex way in the Mathematical Components library. In particular, `bigop` is fragile because the predicate and the general term do not share the same binder. For example, if we write the following

```
Lemma sum_odd n :
  bigop 0 addn (index_iota 0 n.*2) (fun j => odd j) (fun i => i) = n^2
```

What should be printed by the system? It is an iterated sum on `j` or `i`? Similarly, if the index becomes unused during a proof, which name should be printed?

```
Lemma sum_0 n :
  bigop 0 addn (index_iota 0 n) (fun _ => true) (fun _ => 0) = 0
```

To solve these problems we craft a box, `BigBody`, with separate compartments for each sub component. Such box will be used under a single binder and will hold an occurrence of the bound variable even if it is unused in the predicate and in the general term.

```
Inductive bigbody R I := BigBody of I & (R -> R -> R) & bool & R.
```

The arguments of `BigBody` are respectively the index, the iterated operation, the filter and the generic expression. For our running example the `bigbody` component would be:

```
Definition sum_odd_def_body i := BigBody i addn (odd i) i.
```

It is then easy to turn such compound term into the function expected by `foldr`:

```
Definition applybig {R I} (body : bigbody R I) acc :=
  let: BigBody _ op b v := body in if b then op v acc else acc.
```

Finally the generic iterated operator can be defined as follows.

```
Definition bigop R I idx r (body : I -> bigbody R I) :=
  foldr (applybig \o body) idx r.
```

And a generic notation can be attached to it.

```
Notation "\big [ op / idx ]_ ( i <- r | P ) F" :=
  (bigop idx r (fun i => BigBody i op P%B F)) : big_scope.
```

5.9 Working with overloaded notations (⋆)

This little section deals with two "technological issues" the reader may need to know in order to define overloaded notations or work comfortably with them.

The first one is the necessity to tune the behavior of the simplification tactic (the `/=` switch) to avoid loosing the head constant to which the overloaded notation is attached. For example the following term:

```
(@eq_op bool_eqType true false).
```

can be simplified (reduced) to the following one

```
(@eqb true false).
```

While the two terms are logically equivalent (i.e., the logic cannot distinguish them), the pretty printer can. The overloaded `==` notation is attached to the `eq_op` constant, and if such constant fades away the notation follows it. Coq lets one declare constants that should not be automatically simplified away, unless they occur in a context that demands it.

```
Arguments eq_op {_} _ _ : simpl never.
Eval simpl in ∀ x y : bool, x == y.
Eval simpl in ∀ x y : bool, true == false || x == y.
```

The first call to `simpl` does not reduce away `eq_op` leaving the expression untouched. In the second example, it does reduce to `false` the test `(true == false)` in order to simplify the `||` connective.

The converse technological issue may arise when canonical structure inference "promotes" the operator name to a projection of the corresponding canonical monoid structure.

```
Implicit Type l : seq nat.
Lemma example F l1 l2 :
 \sum_(i <- l1 ++ l2) F i =
 \sum_(i <- l2 ++ l1) F i.
Proof.
rewrite big_cat.
rewrite /=.
by rewrite addnC -big_cat.
Qed.
```

Response after line 6

```
F : nat -> nat
l1, l2 : seq nat
============================
addn_monoid
 (\big[addn_monoid/0]_(i <- l1) F i)
 (\big[addn_monoid/0]_(i <- l2) F i) =
\sum_(i <- (l2 ++ l1)) F i
```

It is not uncommon to see `/=` switch in purely algebraic proofs (where no computation is really involved) just to clean up the display of the current conjecture.

5.10 Querying canonical structures (⋆)

It is possible to ask Coq if a certain term does validate an interface. For example, to check if `addn` forms a monoid one can `Check [law of addn]`. A notation

of this kind exists for any interface, for example `[eqType of nat]` is another valid query to check if `nat` is equipped with a canonical comparison function.

This mechanism can also be used to craft notations that assert if one of their arguments validates an interface. For example imagine one wants to define the concept of finite set as an alias of `(seq T)` but such that only values for `T` being `eqType`s are accepted.

The rest of this section introduces the general mechanism of phantom types used to trigger canonical structure resolution.

5.10.1 Phantom types (⋆⋆)

First of all, canonical structure resolution kicks in during unification that in turn is used to compare types. Types are compared whenever a function is applied to an argument, and in particular the type expected by the function and the one of the argument are unified. What we need to craft is a mechanism that takes any input term (proper terms like `addn` but also types as `nat`) and puts it into a type. We will then wire things up so that such type is unified with another one containing the application of a projection to an unknown canonical structure instance.

```
1  Inductive phantom (T : Type) (p : Type) := Phantom.
```

The `Phantom` constructor expects two arguments. If we apply it to `nat`, as in `(Phantom Type nat)`, we obtain a term of type `(phantom Type nat)`. If we apply it to `addn` as in `(Phantom (nat -> nat -> nat) addn)` we obtain a term of type `(phantom (nat -> nat -> nat) addn)`. In both cases the input term (`nat` and `addn` respectively) is now part of a type.

The following example defines a notation `{set T}` that fails if `T` is not an `eqType`: it is an alias of the type `(seq T)` that imposes extra requirements on the type argument.

```
1  Definition set_of (T : eqType) (_ : phantom Type (Equality.sort T)) := seq
       T.
2  Notation "{ 'set' T }" := (set_of _ (Phantom Type T))
3    (at level 0, format "{ 'set'  T }") : type_scope.
```

When type inference runs on `{set nat}` the underlying term being typed is `(set_of ?T (Phantom ?N nat))`. The unification problem arising for the last argument of `set_of` is `(phantom Type (Equality.sort ?T))` versus `(phantom Type nat)`, that in turn contains the sub problem we are interested in: `(Equality.sort ?T)` versus `nat`.

5.11 Exercises

Exercise 16. *Sum of $2n$ odd numbers*

Show the following lemma using the theory of big operators

```
Lemma sum_odd n : \sum_(0 <= i < n.*2 | odd i) i = n^2.
```

5.11.1 Solutions

Answer of Exercise 16

```
Lemma sum_odd n : \sum_(0 <= i < n.*2 | odd i) i = n^2.
Proof.
elim: n => [|n IHn]; first by rewrite unlock.
rewrite doubleS big_mkcond 2?big_nat_recr // -big_mkcond /=.
by rewrite {}IHn odd_double /= addn0 -addnn -!mulnn; ring.
Qed.
```

Chapter 6

Sub-Types

Inductive data types have been used to both code data, like lists, and logical connectives, like the existential quantifier. Properties were always expressed with boolean programs. The questions addressed in this chapter are the following ones. What status do we want to give to, say, lists of size 5, or integers smaller than 7? Which relation to put between integers and integers smaller than 7? How to benefit from extra properties integers smaller than 7 have, like being a finite collection?

In standard mathematics one would simply say that the integers are an infinite set (called `nat`), and that the integers smaller that 7 form a finite subset of the integers (called `'I_7`). Integers and integers smaller than 7 are interchangeable data: if `(n : nat)` and `(i : 'I_7)` one can clearly add `n` to `i`, and eventually show that the sum is still smaller than 7. Also, an informed reader knows which operations are compatible with a subset. E.g. `(i-1)` stays in `'I_7`, as well as `(i+n %| 7)`. So in a sense, subsets also provide a linguistic construct to ask the reader to track an easy invariant and relieving the proof text from boring details.

The closest notion provided by the Calculus of Inductive Constructions is the one of $\Sigma-$types, that we have already seen in the previous chapter in their general form of records. For example, one can define the type `'I_7` as $\Sigma_{(n:nat)} n < 7$. Since proofs are terms, one can pack together objects and proofs of some properties to represent the objects that have those properties. For example 3, when seen as an inhabitant of `'I_7`, is represented by a dependent pair `(3, p)` where `(p : 3 < 7)`. Note that, by forgetting the proof `p`, one recovers a `nat` that can be passed to, say, the program computing the addition of natural numbers, or to theorems quantified on any `nat`. Also, an inhabitant of `'I_7` can always be proved smaller than 7, since such evidence is part of the object itself. We call this construction a *sub-type*.

Such representation can be expensive in the sense that it imposes extra work (proofs!) to create a sub-type object, so it must be used with care. The Mathematical Components library provides several facilities that support the creation

of record-based sub-types, and of their inhabitants. We shall in particular see how both type inference and dynamic tests can be used to supply the property proofs, modelling once again the eye of a trained reader.

Finally, let us point out that we have already encountered proof-carrying records in the previous chapter, with `eqType`. The `eqType` record played the role of an interface, expressing a relation between a type and a function (the comparison operation), and giving access to a whole theory of results through type inference. Many such interfaces can be extended to sub-types, and we shall see that the Mathematical Components library provides facilities to automate this.

6.1 n-tuples, lists with an invariant on the length

We begin by defining the type of n-tuples: sequences of length n. In this section we focus on how tuples are defined, used as regular sequences and how to program type inference to track for us the invariant on tuples' length. Next section will complete the definition of the tuple sub-type by making the abstract theory attached to the `eqType` interface available on tuples whenever it is available on sequences.

A tuple is a sequence of values (of the same type) whose length is made explicit in the type.

Tuple sub-type of seq

```
Structure tuple_of n T :=  Tuple { tval :> seq T; _ : size tval == n }.
Notation "n .-tuple T" := (tuple_of n T).
```

The key property of this type is that it tells us the length of its elements when seen as sequences:

```
Lemma size_tuple T n (t : n.-tuple T) : size t = n.
Proof. by case: t => s /eqP. Qed.
```

In other words each inhabitant of the tuple type carries, in the form of an equality proof, its length. As test bench we pick this simple example: a tuple is processed using functions defined on sequences, namely `rev` and `map`. These operations do preserve the invariant of tuples, i.e., they don't alter the length of the subjacent list.

```
Example seq_on_tuple n (t : n.-tuple nat) :
  size (rev [seq 2 * x | x <- rev t]) = size t.
```

There are two ways to prove that lemma. The first one is to ignore the fact that `t` is a tuple; consider it as a regular sequence, and use only the theory of sequences.

```
Proof. by rewrite map_rev revK size_map. Qed.
```

Mapping a function over the reverse of a list is equivalent to first mapping the function over the list and then reversing the result (`map_rev`). Then, reversing

twice a list is a no-op, since `rev` is an involution (`revK`). Finally, mapping a function over a list does not change its size (`size_map`). The sequence of rewritings makes the left hand side of the conjecture identical to the right hand side, and we can conclude.

This simple example shows that the theory of sequences is usable on terms of type tuple. Still we didn't take any advantage from the fact that `t` is a tuple.

The second way to prove this theorem is to rely on the rich type of `t` to actually compute the length of the underlying sequence.

```
Example just_tuple_attempt n (t : n.-tuple nat) :
  size (rev [seq 2 * x | x <- rev t]) = size t.
Proof. rewrite size_tuple.
```

```
1 subgoal

 n : nat
 t : n .-tuple nat
 ============================
 size (rev [seq 2 * x | x <- rev t]) = n
```

The rewriting replaces the right hand side with `n` as expected, but we can't go any further: the lemma does not apply (yet) to the left hand side, even if we are working with a tuple `t`. Why is that? In the left hand side `t` is processed using functions on sequences. The type of `rev` for example is `(∀ T, seq T -> seq T)`. The coercion `tval` from `tuple_of` to `seq` makes the expression `(rev (tval t))` well typed, but the output is of type `(seq nat)`. We would like the functions on sequences to return data as rich as the one taken in input, i.e., preserve the invariant expressed by the tuple type. Or, in alternative, we would like the system to recover such information.

Let us examine what happens if we try to unify the left hand side of the `size_tuple` equation with the redex `(size (rev t))`, using the following toolkit:

Unification debugging toolkit

```
Notation "X (*...*)" :=
  (let x := X in let y := _ in x)   (at level 100, format "X  (*...*)").
Notation "[LHS 'of' equation ]" :=
  (let LHS := _ in
   let _infer_LHS := equation : LHS = _ in LHS)   (at level 4).
Notation "[unify X 'with' Y ]" :=
  (let unification := erefl _ : X = Y in True).
```

We can now simulate the unification problem encountered by `rewrite size_tuple`

```
Check ∀ T n (t : n.-tuple T),
 let LHS := [LHS of size_tuple _] in
 let RDX := size (rev t) in
 [unify LHS with RDX].
```

The corresponding error message is the following one:

Response

```
Error:
In environment
T : Type
n : nat
t : n .-tuple T
LHS := size (tval ?94 ?92 ?96) (*...*) : nat
RDX := size (rev (tval n T t))           : nat
The term "erefl ?95" has type "?95 = ?95" while
it is expected to have type "LHS = RDX".
```

Unifying `(size (tval ?`$_n$ `?`$_T$ `?`$_t$`))` with `(size (rev (tval n T t)))` is hard. Both term's head symbol is `size`, but then the projection `tval` applied to unification variables has to be unified with `(rev ...)`, and both terms are in normal form.

Such problem is nontrivial because to solve it one has to infer a record for `?`$_t$ that contains a proof: a tuple whose `tval` field is `rev t` (and whose other field contains a proof that such sequence has length `?`$_n$).

We have seen in the previous chapter that this is exactly the class of problems that is addressed by canonical structure instances. We can thus use `Canonical` declarations to teach Coq the effect of list operations on the length of their input.

```
1  Section CanonicalTuples.
2  Variables (n : nat) (A B : Type).
3
4  Lemma rev_tupleP (t : n.-tuple A) : size (rev t) == n.
5  Proof. by rewrite size_rev size_tuple. Qed.
6  Canonical rev_tuple (t : n.-tuple A) := Tuple (rev_tupleP t).
7
8  Lemma map_tupleP (f: A -> B) (t: n.-tuple A) : size (map f t) == n.
9  Proof. by rewrite size_map size_tuple. Qed.
10 Canonical map_tuple (f: A -> B) (t: n.-tuple A) := Tuple (map_tupleP f t)
       .
```

Even if it is not needed for the lemma we took as our test bench, we add another example where the length is not preserved, but rather modified in a statically known way.

```
1  Lemma cons_tupleP (t : n.-tuple A) x : size (x :: t) == n.+1.
2  Proof. by rewrite /= size_tuple. Qed.
3  Canonical cons_tuple x (t : n.-tuple A) : n.+1 .-tuple A :=
4    Tuple (cons_tupleP t x).
```

The global table of canonical solutions is extended as follows.

Canonical Structures Index

projection	value	solution	combines solutions for
`tval N A`	`rev A S`	`rev_tuple N A T`	`T` ← (`tval N A`, `S`)
`tval N B`	`map A B F S`	`map_tuple N A B F T`	`T` ← (`tval N A`, `S`)
`tval N.+1 A`	`X :: S`	`cons_tuple N A T X`	`T` ← (`tval N A`, `S`)

Thanks to the now extended capability of type inference, we can prove our

lemma by just reasoning about tuples.

```
Example just_tuple n (t : n.-tuple nat) :
  size (rev [seq 2 * x | x <- rev t]) = size t.
Proof. by rewrite !size_tuple. Qed.
```

The iterated rewriting acts now twice replacing both the left hand and the right hand side with `n`. It is worth observing that the size of this proof (two rewrite steps) does not depend on the complexity of the formula involved, while the one using only the theory of lists requires roughly one step per list-manipulating function. What depends on the size of the formula is the number of canonical structure resolution steps type inference performs. Another advantage of this last approach is that one is not required to know the names of the lemmas: it is the new concept of tuple that takes care of the size related reasoning steps.

6.2 n-tuples, a sub-type of sequences

We have seen that `(seq T)` is an `eqType` whenever `T` is. We now want to transport such `eqType` structure on tuples. We first do it manually, then we provide a toolkit to ease the declaration of sub-types.

The first step is to define a comparison function for tuples.

```
Definition tcmp n (T : eqType) (t1 t2 : n.-tuple T) := tval t1 == tval t2.
```

Here we simply reuse the one on sequences, and we ignore the proof part of tuples. What we need now to prove

```
Lemma eqtupleP n (T : eqType) : Equality.axiom (@tcmp n T).
Proof.
move=> x y; apply: (iffP eqP); last first.
  by move=> ->.
case: x; case: y => s1 p1 s2 p2 /= E.
rewrite E in p2 *.
by rewrite (eq_irrelevance p1 p2).
Qed.
```

The first direction is trivial by rewriting. The converse direction makes an essential use of the `eq_irrelevance` lemma, which is briefly discussed in section 6.2.2.

Response after line 3

```
2 subgoals

n : nat
T : eqType
x, y : n .-tuple T
============================
x = y -> tval x = tval y

subgoal 2 is:
  tval x = tval y -> x = y
```

Response after line 5

```
1 subgoal

n : nat
T : eqType
s1 : seq T
p1 : size s1 = n
s2 : seq T
p2 : size s2 = n
E : s2 = s1
============================
Tuple p2 = Tuple p1
```

We can then declare the canonical `eqType` instance for tuples.

```
1  Canonical tuple_eqType n T : eqType :=
2    Equality.Pack (Equality.Mixin (@eqtupleP n T)).
```

As a simple test we check that the notations and the theory that equips `eqType` is available on tuples.

```
1  Check ∀ t : 3.-tuple nat, [:: t] == [::].
2  Check ∀ t : 3.-tuple bool, uniq [:: t; t].
3  Check ∀ t : 3.-tuple (7.-tuple nat), undup_uniq [:: t; t].
```

Although all these proofs and definitions are specific to `tuple`, we are following a general schema here, involving three parameters: the original type `(seq T)`, the sub-type `(n.-tuple T)` and the projection `tval`. The Mathematical Components library provides a *sub-type kit* to let one write just the following text:

```
1  Canonical tuple_subType := Eval hnf in [subType for tval].
2  Definition tuple_eqMixin := Eval hnf in [eqMixin of n.-tuple T by <:].
3  Canonical tuple_eqType := Eval hnf in EqType (n.-tuple T) tuple_eqMixin.
```

Line 1 registers `tval` as a canonical projection to obtain a known type out of the newly defined type of tuples. Once the projection is registered the equality axiom can be proved automatically by `[eqMixin of n.-tuple T by <:]`, where `<:` is just a symbol that is reminiscent of sub-typing in functional languages like OCaml. The following section details the implementation of the sub-type kit.

6.2.1 The sub-type kit (⋆)

When one has a base type T and a sub-type ST defined as a boolean sigma type, the Mathematical Components library provides facilities to build all the applicable canonical instances from just the name of the projection going from ST to T.

To register `tval` as the projection from tuples to sequences one writes:

```
1  Canonical tuple_subType := Eval hnf in [subType for tval].
```

As we will see in the next section, the `subType` structure provides a generic notation `val` for the projector of a sub-type (i.e., `tval` for `tuple`), with an over-

loaded injectivity lemma `val_inj` saying that two objects equal in ST are also equal in T.

In addition to the generic projection, we get a generic static constructor `Sub`, which takes a value in the type `T` and a proof.

More interestingly, the sub-type kit provides the dynamic constructors `insub` and `insubd` that do not need a proof as they dynamically test the property, and offer an attractive encapsulation of the difficult *convoy pattern* [5, section 8.4]. The `insubd` constructor takes a default sub-type value which it returns if the tests fails, while `insub` takes only a base type value and returns an `option`; both are *locked* and will not evaluate the test, even for a ground base type value.[1]

Both `Sub` and `insub` expect the typing context to specify the sub-type.

Here is a few example uses, using `tuple`:

```
Variables (s : seq nat) (t : 3.-tuple nat).
Hypothesis size3s : size s == 3.
Let t1 : 3.-tuple nat := Sub s size3s.
Let s2 := if insub s is Some t then val (t : 3.-tuple nat) else nil.
Let t3 := insubd t s. (* : 3.-tuple nat *)
```

We put `insub` to good use in section 6.4 when an enumeration for sub-types is to be defined.

The `subType` structure describes a *boolean sigma-type* (a dependent pair whose second component is the proof of a boolean formula) in terms of its projector, constructor, and elimination rule:

```
Section SubTypeKit.
Variables (T : Type) (P : pred T).

Structure subType : Type := SubType {
  sub_sort :> Type;
  val : sub_sort -> T;
  Sub : ∀ x, P x -> sub_sort;
  (* elimination rule for sub_sort *)
  _ : ∀ K (_ : ∀ x Px, K (@Sub x Px)) u, K u;
  _ : ∀ x Px, val (@Sub x Px) = x
}.
```

Instances can provide unification hints for any of the three named fields, not just for `sub_sort`. Hence, `val` $?_u$ unifies with `tval t`, and `Sub` $?_x$ $?_{xP}$ unifies with `Tuple s sP`, including in `rewrite` patterns.

The `subType` constructor notation assumes the sub-type is isomorphic to a sigma-type, so that its elimination rule can be derived using Coq's generic destructing `let`, and the projector-constructor identity can be proved by reflexivity.

```
Notation "[ 'subType' 'for' v ]" := (SubType _ v _
  (fun K K_S u => let (x, Px) as u return K u := u in K_S x Px)
  (fun x px => erefl x)).
```

Note how the value of `Sub` is determined by unifying the type of the first function with the type expected by `SubType`, with the help of the "`as u return K u`"

[1] The equations describing the computation of `insub` are called `insubT` and `insubF`.

annotation, see [24, section 1.2.13].

A useful variant of `[subType for tval]` is `[newType for Sval]`. Such specialized constructors force the predicate defining the sub-type to be the trivial one: the sub-type ST adds no property to the type T, but the resulting type ST is different from T and inhabitants of ST cannot be mistaken for inhabitants of T. Of course all the theory that equips T is also available on ST. Aliasing a type is useful to attach to it different notations or coercions.

6.2.2 A note on boolean Σ-types

The `eq_irrelevance` theorem used to prove that tuples form an `eqType` is a delicate matter in the Calculus of Inductive Constructions. In particular it is not valid in general: two proofs of the same predicate may not be provably equal.

To the rescue comes the result of Hedberg [15] that proves such property for a wide class of predicates. In particular it shows that any type with decidable identity has unique identity proofs. This result can be proved in its full generality in the Mathematical Components library, using to the `eqType` interface.

Hedberg

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

If we pick the concrete example of `bool`, then all proofs that `(b = true)` for a fixed `b` are identical.

Here we can see another crucial advantage of boolean reflection. Forming sub-types poses no complication from a logic perspective since proofs of boolean identities are very simple, canonical, objects.

In the Mathematical Components library, where *all predicates that can* be expressed as a boolean function *are expressed as a boolean function*, forming sub-types is extremely easy.

Advice

It is convenient to define new types as sub-types of existing ones, since they inherit all the theory.

6.3 Finite types and their theory

Before describing other sub-types, we introduce the interface of types equipped with a finite enumeration.

Interface for finite types

```
Notation count_mem x := (count [pred y | y == x]).
Module finite.
Definition axiom (T : eqType) (e : seq T) :=
  ∀ x : T, count_mem x e = 1.

Record mixin_of (T : eqType) := Mixin {
  enum : seq T;
  _ : axiom T enum;
}
End finite.
```

The axiom asserts that any inhabitant of `T` occurs exactly once in the enumeration `e`. We omit here the full definition of the interface, as it will be discussed in detail in the next chapter. What is relevant for the current section is that `finType` is the structure of types equipped with such enumeration, that any `finType` is also an `eqType` (see the parameter of the mixin), and that, to declare a `finType` instance, one can write:

Declare a finType

```
Definition mytype_finMixin := Finite.Mixin mytype_enum mytype_enumP.
Canonical mytype_finType := @Finite.Pack mytype mytype_finMixin.
```

Given that the most recurrent way of showing that an enumeration validates `Finite.axiom` is by proving that it is both duplicate free and exhaustive, a convenience mixin constructor is provided.

Declare a finType

```
Lemma myenum_uniq : uniq myenum.
Lemma mem_myenum : ∀ x : T, x \in myenum.
Definition mytype_finMixin := Finite.UniqFinMixin myenum_uniq mem_myenum.
```

The interface of `finType` comes equipped with a theory that, among other things, provides a cardinality operator `#|T|` and bounded boolean quantifications like `[∀ x, P]`.

Some theory for finType

```
Lemma cardT (T : finType) : #|T| = size (enum T).
Lemma forallP (T : finType) (P : pred T) : reflect (∀ x, P x) [∀ x, P x].
```

Given that `[∀ x, P x]` is a boolean expression, it enables reasoning by excluded middle and also combines well with other boolean connectives.

What makes this formulation of finite types handy is the explicit enumeration. It is hence trivial to iterate over the inhabitants of the finite type. This makes finite type easy to integrate in the library of iterated operations. In particular notations like `(\sum_(i : T) F)` are used to express the iteration over the inhabitants of the finite type `T`.

6.4 The ordinal subtype

Apart from the aforementioned theory, finite types can serve as a powerful notational device for ranges. For example one may want to state that a matrix of size $m \times n$ is only accessed inside its bounds, i.e., that one cannot get the $m + 1$ row. The way this will be formulated in the Mathematical Components library is by saying that its row accessors accept only inhabitants of a finite type of size m. Accessing a matrix out of its bounds becomes a type error. Of course one wants to access a matrix using integer coordinates, but integers are infinite. Hence the first step is to define the sub-type of bounded integers:

Ordinals

```
Inductive ordinal (n : nat) : Type := Ordinal m of m < n.
Notation "'I_' n" := (ordinal n)

Coercion nat_of_ord i := let: @Ordinal m _ := i in m.

Canonical ordinal_subType := [subType for nat_of_ord].
Definition ordinal_eqMixin := Eval hnf in [eqMixin of ordinal by <:].
Canonical ordinal_eqType := Eval hnf in EqType ordinal ordinal_eqMixin.
```

The constructor `Ordinal` has two arguments: a natural number `m` and a proof that this number is smaller than the parameter `n`. We use the `of` notation for arguments of constructors that do not need to be named; thus `Ordinal m of m < n` stands for `Ordinal m (_ : m < n)`

We start by making ordinals a subtype of natural numbers, and hence inherit the theory of `eqType`. To show they form a `finType`, we need to provide a good enumeration.

```
Definition ord_enum n : seq (ordinal n) := pmap insub (iota 0 n).
```

The `iota` function produces the sequence `[:: 0, 1, ... n.-1]`. Such a sequence is mapped via `insub` that tests if an element `x` is smaller than `n`. If it is the case it produces `(Some x)`, where `x` is an ordinal, else `None`. `pmap` drops all `None` items, and removes the `Some` constructor from the others.

What `ord_enum` produces is hence a sequence of ordinals, i.e., a sequence of terms like `(@Ordinal m p)` where `m` is a natural number (as produced by `iota`) and `p` is a proof that `(m <= n)`. What we are left to show is that such an enumeration is complete and non-redundant.

```
Lemma val_ord_enum : map val ord_enum = iota 0 n.
Proof.
rewrite pmap_filter; last exact: insubK.
by apply/all_filterP; apply/allP=> i; rewrite mem_iota isSome_insub.
Qed.

Lemma ord_enum_uniq : uniq ord_enum.
Proof. by rewrite pmap_sub_uniq ?iota_uniq. Qed.

Lemma mem_ord_enum i : i \in ord_enum.
Proof. by rewrite -(mem_map ord_inj) val_ord_enum mem_iota ltn_ord. Qed.
```

It is worth pointing out how the `val_ord_enum` lemma shows that the ordinals in `ord_enum` are exactly the natural numbers generated by `(iota 0 n)`. In particular, the `insub` construction completely removes the need for complex dependent case analysis.

```
2 subgoals
n : nat
============================
[seq x <- iota 0 n | isSome (insub x)] = iota 0 n

subgoal 2 is:
 ocancel insub val
```

The view `all_filterP` shows that `reflect ([seq x <- s | a x] = s) (all a s)` for any sequence `s` and predicate `a`. After applying that view, one has to prove that if `(i \in iota 0 n)` then `(i < n)`, that is trivialized by `mem_iota`.

We can now declare the type of ordinals as a instance of `finType`.

```
Definition ordinal_finMixin n :=
  Eval hnf in UniqFinMixin (ord_enum_uniq n) (mem_ord_enum n).
Canonical ordinal_finType n :=
  Eval hnf in FinType (ordinal n) (ordinal_finMixin n).
```

An example of ordinals at work is the `tnth` function. It extracts the n-th element of a tuple exactly as `nth` for a sequence but without requiring a default element. As a matter of fact, one can use ordinals to type the index, making Coq statically checks that the index is smaller than the size of the tuple.

```
Lemma tnth_default T n (t : n.-tuple T) : 'I_n -> T.
Proof. by rewrite -(size_tuple t); case: (tval t) => [|//] []. Qed.

Definition tnth T n (t : n.-tuple T) (i : 'I_n) : T :=
  nth (tnth_default t i) t i.
```

Another use of ordinals is to express the position of an inhabitant of a `finType` in its enumeration.

```
Definition enum_rank (T : finType) : T -> 'I_#|T|.
```

6.5 Finite functions

In standard mathematics functions that are point wise equal are considered as equal. This principle, that we call *functional extensionality*, is compatible with the Calculus of Inductive Constructions but is not built-in. At the time of writing, only very recent variations of CIC, as Cubical Type Theory [7], include such principle.

Still, this principle is provable in Coq for functions with a finite domain, provided that they are described with a suitable representation. Indeed, the graph of a function with a finite domain is just a finite set of points, which can be represented by a finite sequence of values. The length of this sequence is the size of the domain. Pointwise equal finite functions have the same sequence of

values, hence their representations are equal. The actual definition of the type of finite functions in the Mathematical Components library uses this remark, plus the tricks explained in section 5.10:

```
Section FinFunDef.
Variables (aT : finType) (rT : Type).

Inductive finfun_type : Type := Finfun of #|aT|.-tuple rT.
Definition finfun_of of phant (aT -> rT) := finfun_type.
Definition fgraph f := let: Finfun t := f in t.

Canonical finfun_subType := Eval hnf in [newType for fgraph].

End FinFunDef.

Notation "{ 'ffun' fT }" := (finfun_of (Phant fT)).
```

As a result, the final notation provides a way to describe an instance of the type of finite function by giving the mere type of the domain and co-domain, without mentioning the name of the instance of `finType` for the domain. One can thus write the term `{ffun 'I_7 -> nat}` but the term `{ffun nat -> nat}` would raise an error message, and their cannot be a registered instance of finite type for `nat`.

Other utilities let one apply a finite function as a regular function or build a finite function from a regular function.

```
Definition fun_of_fin aT rT f x := tnth (@fgraph aT rT f) (enum_rank x).
Coercion fun_of_fin : finfun >-> FunClass.
Definition finfun aT rT f := @Finfun aT rT (codom_tuple f).
Notation "[ 'ffun' x : aT => F ]" := (finfun (fun x : aT => F))
```

What `codom_tuple` builds is a list of values `f` takes when applied to the values in the enumeration of its domain.

```
Check [ffun i : 'I_4 => i + 2].  (* : {ffun 'I_4 -> nat} *)
```

Finite functions inherit from tuples the `eqType` structure whenever the codomain is an `eqType`.

```
Definition finfun_eqMixin aT (rT : eqType) :=
  Eval hnf in [eqMixin of finfun aT rT by <:].
Canonical finfun_eqType :=
  Eval hnf in EqType (finfun aT rT) finfun_eqMixin.
```

When the codomain is finite, the type of finite functions is itself finite. This property is again inherited from tuples. Recall the `all_words` function, solution of exercise 7.

```
Definition tuple_enum (T : finType) n : seq (n.-tuple T) :=
  pmap insub (all_words n (enum T)).
Lemma enumP T n : Finite.axiom (tuple_enum T n).

Definition tuple_finMixin := Eval hnf in FinMixin (@FinTuple.enumP n T).
Canonical tuple_finType:= Eval hnf in FinType (n.-tuple T) tuple_finMixin.

Definition finfun_finMixin (aT rT : finType) :=
  [finMixin of (finfun aT rT) by <:].
Canonical finfun_finType  aT rT :=
  Eval hnf in FinType (finfun aT rT) (finfun_finMixin aT rT).
```

A relevant property of the `finType` of finite functions is its cardinality, being equal to `#|rT| ^ #|aT|`.

```
Lemma card_ffun (aT rT : finType) : #| {ffun aT -> rT} | = #|rT| ^ #|aT|.
```

Also, as expected, finite functions validate extensionality.

```
Definition eqfun (f g : B -> A) : Prop := ∀ x, f x = g x.
Notation "f1 =1 f2" := (eqfun f1 f2).

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

A first application of the type of finite functions is the following lemma.

```
Lemma bigA_distr_bigA (I J : finType) F :
  \big[*%M/1]_(i : I) \big[+%M/0]_(j : J) F i j
    = \big[+%M/0]_(f : {ffun I -> J}) \big[*%M/1]_i F i (f i).
```

Such lemma, rephrased in mathematical notation down below, states that the indices `i` and `j` are independently chosen.

$$\prod_{i \in I} \sum_{j \in J} Fij = \sum_{f \in I \to J} \prod_{i \in I} Fi(fi)$$

Remark how the finite type of functions from `I` to `J` is systematically formed in order to provide its enumeration as the range of the summation.

6.6 Finite sets

We have seen how sub-types let one easily define a new type by, typically, enriching an existing one with some properties. While this is very convenient for defining new types, it does not work well when the subject of study are sets and subsets of the type's inhabitants. In such case, it is rather inconvenient to define a new type for each subset, because one typically combines elements of two distinct subsets with homogeneous operations, like equality.

The Mathematical Components library provides an extensive library of finite sets and subsets that constitutes the pillar of finite groups.

```
Section finSetDef.
Variable T : finType.
Inductive set_type : Type := FinSet of {ffun pred T}.
Definition finfun_of_set A := let: FinSet f := A in f.
```

Recall that `(pred T)` is the type of functions from `T` to `bool`.

Using the sub-type kit we can easily transport the `eqType` and `finType` structure over finite sets.

```
Canonical set_subType := Eval hnf in [newType for finfun_of_set].
Definition set_eqMixin := Eval hnf in [eqMixin of set_type by <:].
Canonical set_eqType := Eval hnf in EqType set_type set_eqMixin.
Definition set_finMixin := [finMixin of set_type by <:].
Canonical set_finType := Eval hnf in FinType set_type set_finMixin.
End finSetDef.
Notation "{ 'set' T }" := (set_type T).
```

We omit again the trick to statically enforce that `T` is a finite type whenever we write `{set T}`, exactly as we did for finite functions. Finite sets do validate extensionality and are equipped with subset-wise and point-wise operations:

```
Lemma setP A B : A =i B <-> A = B.

Lemma example (T : finType) (x : T) (A : {set T}) :
  (A \subset x |: A) && (A :==: A :&: A) && (x \in [set y | y == x])
```

It is worth noticing that many "set" operations are actually defined on simpler structures we did not detail for conciseness. In particular membership and subset are also applicable to predicates, i.e. terms that can be seen as functions from a type to `bool`.

Since `T` is finite, values of type `{set T}` admit a complement and `{set T}` is closed under power-set construction.

```
Lemma setCP x A : reflect (~ x \in A) (x \in ~: A).
Lemma subsets_disjoint A B : (A \subset B) = [disjoint A & ~: B].
Definition powerset D : {set {set T}} := [set A : {set T} | A \subset D].
Lemma card_powerset (A : {set T}) : #|powerset A| = 2 ^ #|A|.
```

We have seen how tuples can be used to carry an invariant over sequences. In particular type inference was programmed to automatically infer the effect of sequence operations over the size of the input tuple. In a similar way finite groups can naturally be seen as sets and some set-wise operations, like intersection, do preserve the group structure and type inference can be programmed to infer so automatically.

6.7 Permutations

Another application of finite functions is the definition of the type of permutations.

```
Inductive perm_of (T : finType) : Type :=
  Perm (pval : {ffun T -> T}) & injectiveb pval.
Definition pval p := let: Perm f _ := p in f.
Notation "{ 'perm' T }" := (perm_of T).
```

This time we add the property of being injective, that in conjunction with finiteness characterizes permutations as bijections.

Similarly to finite functions we can declare a coercion to let one write `(s x)` for `(s : {perm T})` to denote the result of applying the permutation `s` to `x`.

Thanks to the sub-type kit it is easy to transport to the type `{perm T}` the `eqType` and `finType` structures of `ffun T -> T`.

```
Canonical perm_subType := Eval hnf in [subType for pval].
Definition perm_eqMixin := Eval hnf in [eqMixin of perm_type by <:].
Canonical perm_eqType := Eval hnf in EqType perm_type perm_eqMixin.
Definition perm_finMixin := [finMixin of perm_type by <:].
Canonical perm_finType := Eval hnf in FinType perm_type perm_finMixin.
```

A special class of permutations that comes in handy to express the calculation of a matrix determinant is the permutation of `'I_n`.

```
Notation "''S_' n" := {perm 'I_n}.
```

A relevant result of the theory of permutations is about counting their number. It is expressed on a subset `S` and counts only non-identity permutations.

```
Definition perm_on T (S : {set T}) : pred {perm T} :=
  fun s => [set x | s x != x] \subset S.
Lemma card_perm A : #|perm_on A| = #|A| `!.
```

6.8 Matrix

We finally have all the bricks to define the type of matrices and provide compact formulations for their most common operations.

```
Inductive matrix R m n : Type := Matrix of {ffun 'I_m * 'I_n -> R}.
Definition mx_val A := let: Matrix g := A in g.
Notation "''M[' R ]_ ( m , n )" := (matrix R m n).
Notation "''M_' ( m , n )" := (matrix _ m n).
Notation "''M[' R ]_ n" := (matrix R n n).
```

As for permutations and finite functions we declare a coercion to let one denote `(A i j)` the coefficient in column j of row i. Note that type inference will play an important role here. If `A` has type `'M[R]_(m,n)`, then Coq will infer that `(i : 'I_m)` and `(j : 'I_n)` from the expression `(A i j)`. In combination with the notations for iterated operations, this lets one define, for example, the trace of a square matrix as follows.

```
Definition mxtrace R n (A : 'M[R]_n) := \sum_i A i i.
Local Notation "'\tr' A" := (mxtrace R n A).
```

Note that, for the `\sum` notation to work, `R` needs to be a type equipped with an addition, for example a `ringType`. We will describe such type only in the next chapter. From now on the reader shall interpret the `+` and `*` symbols on the matrix coefficients as (overloaded) ring operations, exactly as `==` is the overloaded comparison operation of `eqType`.

Via the sub-type kit we can transport `eqType` and `finType` from `{ffun 'I_m * 'I_n -> R}` to `'M[R]_(m,n)`. We omit the Coq code for brevity.

A useful accessory is the notation to define matrices in their extension. We provide a variant in which the matrix size is given and one in which it has to be inferred from the context.

```
Definition matrix_of_fun R m n F :=
  Matrix [ffun ij : 'I_m * 'I_n => F ij.1 ij.2].
Notation "\matrix_ ( i < m , j < n ) E" :=
  (matrix_of_fun (fun (i : 'I_m) (j : 'I_n) => E))
Notation "\matrix_ ( i , j ) E" := (matrix_of_fun (fun i j => E)).

Example diagonal := \matrix_(i < 3, j < 7) if i == j then 1 else 0.
```

An interesting definition is the one of determinant. We base it on Leibniz's formula: $\sum_{\sigma \in S_n} \operatorname{sgn}(\sigma) \prod_{i=1}^{n} A_{i,\sigma(i)}$.

```
Definition determinant n (A : 'M_n) : R :=
  \sum_(s : 'S_n) (-1) ^+ s * \prod_i A i (s i).
```

The `(-1) ^+ s` denotes the signature of a permutation `s`: `s` can be used, thanks to a coercion, as a natural number that is `0` if `s` is an even permutation, `1` otherwise, and `^+` is ring exponentiation. In other words the `(-1)` factor is annihilated when `s` is even.

What makes this definition remarkable is the resemblance to the same formula typeset in LaTeX:

```
\sum_{\sigma \in S_n} \sgn(\sigma) \prod_{i = 1}^n A_{i, \sigma(i)}
```

Matrix multiplication deserves a few comments too.

```
Definition mulmx m n p (A : 'M_(m, n)) (B : 'M_(n, p)) : 'M[R]_(m, p) :=
  \matrix_(i, k) \sum_j (A i j * B j k).
Notation "A *m B" := (mulmx A B) : ring_scope.
```

First, the type of the inputs makes such operation total, i.e., Coq rejects terms which would represent the product of two matrices whose sizes are incompatible.

This has to be compared with what was done for integer division, that was made total by returning a default value, namely `0`, outside its domain. In the case of matrices a size annotation is enough to make the operation total, while for division a proof would be necessary. Working with rich types is not always easy, for example the type checker does not understand automatically that a square matrix of size `(m + n)` can be multiplied with a matrix of size `(n + m)`. In such case the user has to introduce explicit size casts, see section 6.8.3. At the same time type inference lets one omit size information most of the time, playing once again the role of a trained reader.

6.8.1 Example: matrix product commutes under trace

As an example let's take the following simple property of the trace. Note that we can omit the dimensions of `B` since it is multiplied by `A` to the left and to the right.

```
1  Lemma mxtrace_mulC m n (A : 'M[R]_(m, n)) B :
2     \tr (A *m B) = \tr (B *m A).
3  Proof.
4  have -> : \tr (A *m B) = \sum_i \sum_j A i j * B j i.
5    by apply: eq_bigr => i _; rewrite mxE.
```

The idea of the proof is to lift the commutativity property of the multiplication in the coefficient's ring. The first step is to prove an equation that expands the trace of matrix product. The plan is to expand it on both sides, then exchange the summations and compare the coefficients pairwise.

```
1 subgoal
R : comRingType
m, n : nat
A : 'M_(m, n)
B : 'M_(n, m)
============================
\sum_i \sum_j A i j * B j i = \tr (B *m A)
```

It is worth noticing that the equation we used to expand the left hand side and the one we need to expand the right hand side are very similar. Actually the sub proof following `have` can be generalized to any pair of matrices `A` and `B`. The Small Scale Reflection proof language provides the `gen` modifier in order to tell `have` to abstract the given formula over a list of context entries, here `m n A B`.

```
1  Lemma mxtrace_mulC m n (A : 'M[R]_(m, n)) B :
2     \tr (A *m B) = \tr (B *m A).
3  Proof.
4  gen have trE, trAB: m n A B / \tr (A *m B) = \sum_i \sum_j A i j * B j i.
5    by apply: eq_bigr => i _; rewrite mxE.
6  rewrite trAB trE.
```

The `gen have` step now generates two equations, a general one called `trE`, and its instance to `A` and `B` called `trAB`.

```
1 subgoal
R : comRingType
m, n : nat
A : 'M_(m, n)
B : 'M_(n, m)
trE : ∀ m n (A : 'M_(m, n)) B, \tr (A *m B) = \sum_i \sum_j A i j * B j i
trAB : \tr (A *m B) = \sum_i \sum_j A i j * B j i
============================
\sum_i \sum_j A i j * B j i = \sum_i \sum_j B i j * A j i
```

The proof is then concluded by exchanging the summations, i.e., summing on both sides first on `i` then on `j`, and then proving their equality by showing the identity on the summands.

```
rewrite exchange_big /=.
by do 2!apply: eq_bigr => ? _; apply: mulrC.
Qed.
```

Note that the final identity is true only if the multiplication of the matrix coefficients is commutative. Here `R` was assumed to be a `comRingType`, the structure of commutative rings and `mulrC` is the name of the commutative property (`C`) of ring (`r`) multiplication (`mul`). A more detailed description of the hierarchy of structures is the subject of the next chapter.

6.8.2 Block operations

The size information stocked in the type of a matrix is also used to drive the decomposition of a matrix into sub-matrices, called blocks. For example when the size expression of a square matrix is like `(n1 + n2)`, then the upper left block is a square matrix of size `n1`.

Block destructors

```
Definition lsubmx (A : 'M_(m, n1 + n2)) : 'M_(m, n1)
Definition usubmx (A : 'M_(m1 + m2, n)) : 'M_(m1, n)
Definition ulsubmx (A : 'M_(m1 + m2, n1 + n2)) : 'M_(m1, n1)
```

Conversely blocks can be glued together. This time, it is the size of the resulting matrix that shows a trace of the way it was built.

Block constructors

```
Definition row_mx (A1 : 'M_(m, n1)) (A2 : 'M_(m, n2)) : 'M_(m, n1 + n2)
Definition col_mx (A1 : 'M_(m1, n)) (A2 : 'M_(m2, n)) : 'M_(m1 + m2, n)
Definition block_mx Aul Aur Adl Adr : 'M_(m1 + m2, n1 + n2)
```

The interested reader can find in [10] a description of Cormen's LUP decomposition, an algorithm making use of these constructions. In particular, recursion on the size of a square matrix of size n naturally identifies an upper left square block of size 1, a row and a column of length $n-1$, and a square block of size $n-1$.

6.8.3 Size casts (⋆)

Types containing values are a double edged sword. While we have seen that they make the writing of matrix expressions extremely succinct, in some cases they require extra care. In particular the equality predicate accepts arguments of the very same type. Hence a statement like this one requires a size cast:

```
Section SizeCast.
Variables (n n1 n2 n3 m m1 m2 m3 : nat).

Lemma row_mxA (A1 : 'M_(m, n1)) (A2 : 'M_(m, n2)) (A3 : 'M_(m, n3)) :
  row_mx A1 (row_mx A2 A3) = row_mx (row_mx A1 A2) A3.
```

Observe that the left hand side has type `'M_(m, n1 + (n2 + n3))` while the right hand side has type `'M_(m, (n1 + n2) + n3)`. The `castmx` operator, and all its companion lemmas, let one deal with this inconvenience.

```
Lemma row_mxA (A1 : 'M_(m, n1)) (A2 : 'M_(m, n2)) (A3 : 'M_(m, n3)) :
  let cast := (erefl m, esym (addnA n1 n2 n3)) in
  row_mx A1 (row_mx A2 A3) = castmx cast (row_mx (row_mx A1 A2) A3).
```

The `cast` object provides the proof evidence that `(m = m)`, not strictly needed, and that `(n1 + (n2 + n3) = (n1 + n2) + n3)`.

Lemmas like the following two let one insert or remove additional casts.

```
Lemma castmxKV (eq_m : m1 = m2) (eq_n : n1 = n2) :
  cancel (castmx (esym eq_m, esym eq_n)) (castmx (eq_m, eq_n)).
Lemma castmx_id m n erefl_mn (A : 'M_(m, n)) : castmx erefl_mn A = A.
```

Remark that `erefl_mn` must have type `((m = m) * (n = n))`, i.e., it is a useless cast.

Another useful tool is `conform_mx` that takes a default matrix of the right dimension and a second one that is returned only if its dimensions match.

```
Definition conform_mx (B : 'M_(m1, n1)) (A : 'M_(m, n)) :=
  match m =P m1, n =P n1 with
  | ReflectT eq_m, ReflectT eq_n => castmx (eq_m, eq_n) A
  | _, _ => B
  end.
```

Remember that the notation `(m =P m1)` stands for `(@eqP nat_eqType m m1)`, a proof of the `reflect` inductive spec. Remember also that the `ReflectT` constructor carries a proof of the equality.

The following helper lemmas describe the behavior of `conform_mx` and how it interacts with casts.

```
Lemma conform_mx_id (B A : 'M_(m, n)) : conform_mx B A = A.
Lemma nonconform_mx (B : 'M_(m1, n1)) (A : 'M_(m, n)) :
  (m != m1) || (n != n1) -> conform_mx B A = B.

Lemma conform_castmx (e_mn : (m2 = m3) * (n2 = n3))
  (B : 'M_(m1, n1)) (A : 'M_(m2, n2)) :
    conform_mx B (castmx e_mn A) = conform_mx B A.
```

Sorts and `reflect` (⋆⋆)

The curious reader may have spotted that the declaration of the `reflect` inductive predicate of section 4.2.1 differs from the one part of the Mathematical Components library in a tiny detail. The real declaration indeed puts `reflect` in `Type` and not in `Prop`.

Recall that `reflect` is typically used to state properties about decidable predicates. It is quite frequent to reason on such a class of predicates by excluded middle in *both* proofs and programs. As soon as one needs proofs to cast terms, the proof evidence carried by the reflection lemma becomes doubly useful. Placing the declaration of `reflect` in `Type` is enough to make such a proof accessible

within programs.

However the precise difference between `Prop` and `Type` in the Calculus of Inductive Constructions is off topic for this text, so we will not detail further.

Chapter 7

Organizing Theories

We have seen in the last two chapters how inferred dependent records — *structures* — are an efficient means of endowing mathematical objects with their expected operations and properties. So far we have only seen single-purpose structures: `eqType` provides decidable equality, `subType` an embedding into a representation type, etc.

However, the more interesting mathematical objects have *many* operations and properties, most of which they share with other kinds of objects: for example, elements of a field have all the properties of those of a ring (and more), which themselves have all the properties of an additive group. By organizing the corresponding Calculus of Inductive Constructions structures in a hierarchy, we can materialize these inclusions in the Mathematical Components library, and share operations and properties between related structures. For example, we can use the same generic ring multiplication for rings, integral domains, fields, algebras, and so on.

Organizing structures in a hierarchy does not require any new logical feature beyond those we have already seen: type inference with dependent types, coercions and canonical instances of structures. It is only a "simple matter of programming", albeit one that involves some new formalisation idioms. This chapter describes the most important: telescopes, packed classes, and phantom parameters.

While some of these formalisation patterns are quite technical, casual users do not need to master them all: indeed the documented interface of structures suffices to use and declare instances of structures. We describe these interfaces first, so only those who wish to extend old or create new hierarchies need to read on.

7.1 Structure interface

Most of the documented interface to a structure concerns the operations and properties it provides. This will be obvious from the embedded documentation of the ssralg library, which provides structures for most of basic algebra (including rings, modules, fields). While these are of course important, they pertain to elements of the structure rather than the structure itself, and indeed are usually defined outside of the module introducing the structure.

The intrinsic interface of a structure is much smaller, and consists mostly of functions for creating instances to be typically declared `Canonical`. For structures like `eqType` that are packaged in a submodule (`Equality` for `eqType`), the interface coincides with the contents of the `Exports` submodule. For `eqType` the interface comprises:

- `eqType`: a short name for the structure type (here, `Equality.type`)
- `EqMixin`, `PcanEqMixin`, `[eqMixin of T by <:]`: mixin constructors that bundle the *new* operations and properties the structure provides.
- `EqType`: an instance constructor that creates an instance of the structure from a mixin.
- `[eqType of T]`, `[eqType of T for S]`: cloning constructors that specialize a canonical or given instance of the structure (to `T` here).
- canonical instances and coercions that link the structures to lower ones in the hierarchy or to its elements, e.g., `Equality.sort`.

Canonical instances and coercions are not mentioned directly in the documentation because they are only used indirectly, through type inference; a casual user of a structure only needs to be aware of which other structure it extends, in the hierarchy.

Let us see how the creation operations are used in practice, drawing examples from the zmodp library that puts a "mod p" algebraic structure on the type `ordinal p` of integers less than p. The `ordinal` type is defined in library fintype as follows:

```
1  Inductive ordinal n := Ordinal m of m < n.
2  Notation "'I_ n'" := (Ordinal n).
3  Coercion nat_of_ord n (i : 'I_n) := let: @Ordinal _ m _ := i in m.
```

Algebra only makes sense on non-empty types, so zmodp only defines arithmetic on `'I_p` when `p` is an explicit successor. This makes it easy to define `inZp`, a "mod p" right inverse to the `ordinal >-> nat` coercion, and a 0 value. With these the definition of arithmetic operations and the proof of the basic algebraic identities is straightforward.

```
1  Variable p' : nat.
2  Local Notation p := p'.+1.
3  Implicit Types x y : 'I_p.
4  Definition inZp i := Ordinal (ltn_pmod i (ltn0Sn p')).
```

The `inZp` construction injects any natural number `i` into `'I_p` by applying the modulus. Indeed the type of `ltn_pmod` is `(∀ m d : nat,0 < d -> m %% d < d)`, and `(ltn0Sn p')` is a proof that `(0 < p)`.

We can now build the $\mathbb{Z}$-module operations and properties:

```
1  Definition Zp0 : 'I_p := ord0.
2  Definition Zp1 := inZp 1.
3  Definition Zp_opp x := inZp (p - x).
4  Definition Zp_add x y := inZp (x + y).
5  Definition Zp_mul x y := inZp (x * y).
6
7  Lemma Zp_add0z : left_id Zp0 Zp_add.
8  Lemma Zp_mulC : commutative Zp_mul.
9  ...
```

Creating an instance of the lowest ssralg structure, the $\mathbb{Z}$-module (i.e., additive group), requires two lines:

```
1  Definition Zp_zmodMixin := ZmodMixin Zp_addA Zp_addC Zp_add0z Zp_addNz.
2  Canonical Zp_zmodType := ZmodType 'I_p Zp_zmodMixin.
```

Line 1 bundles the additive operations (0, +, −) and their properties in a *mixin*, which is then used in line 2 to create a canonical instance. After line 2 all the additive algebra provided in ssralg becomes applicable to `'I_p`; for example `0` denotes the zero element, and `i + 1` denotes the successor of `i` mod p.

The `ZmodMixin` constructor infers the operations `Zp_add`, etc., from the identities `Zp_add0z`, etc.. Providing an explicit definition for the mixin, rather than inlining it in line 2, is important as it speeds up type checking, which never need to open the mixin bundle.

The first argument to the instance constructor `ZmodType` is somewhat redundant, but documents precisely the type for which this instance will be used. This can be important as the value inferred by Coq could be "different". Indeed line 2 does perform some nontrivial inference, because, although `zmodType` is the first ssralg structure, it is not at the bottom of the hierarchy: in particular `zmodType` derives from `eqType`, so the `==` test can be used on all `zmodType` elements. The `ZmodType` constructor infers a parent structure instance from its first argument, then combines it with the mixin to create a full `zmodType` instance. This inference can have the side effect of unfolding constants occurring in the description of the type (though not in this case), that is the value on which the canonical solution is indexed. For this reason the type description, `'I_p` here, has to be provided explicitly. Section 7.4 gives the technical details of instance constructors.

Rings are the next step in the algebraic hierarchy. In order to simplify the formalization of the theory of polynomials, ssralg only provides structures for nontrivial rings, so we now need to restrict to `p` of the form `p'.+2`:

```
1  Variable p' : nat.
2  Local Notation p := p'.+2.
3  Lemma Zp_nontrivial : Zp1 != 0 :> 'I_p. Proof. by []. Qed.
4  Definition Zp_ringMixin :=
5    ComRingMixin (@Zp_mulA _) (@Zp_mulC _) (@Zp_mul1z _) (@Zp_mul_addl _)
       Zp_nontrivial.
6  Canonical Zp_ringType := RingType 'I_p Zp_ringMixin.
7  Canonical Zp_comRingType := ComRingType 'I_p (@Zp_mulC _).
```

Line 6 endows `'I_p` with a `ringType` structure, making it possible to multiply in `'I_p` or have polynomials over `'I_p`. Line 7 adds a `comRingType` commutative ring structure, which makes it possible to reorder products, or distribute evaluation over products of polynomials. Note that no mixin definition is needed for line 8 as only a single property is added.

Constraining the shape of the modulus p is a simple and robust way to enforce $p > 1$: it standardizes the proofs of $p > 0$ and $p > 1$, which avoid the unpleasantness of multiple interpretations of `0` stemming from different proofs of $p > 0$ — the latter tends to happen with ad hoc inference of such proofs using canonical structures or tactics. The shape constraint can however be inconvenient when the modulus is an abstract constant (say, `Variable p`), and zmodp provides some syntax to handle that case:

```
1  Definition Zp_trunc p := p.-2.
2  Notation "'Z_' p" := 'I_(Zp_trunc p).+2.
```

Although it is provably equal to `'I_p` when $p > 1$, `'Z_p` is the preferred way of referring to that type when using its ring structure. Note that the two types are identical when `p` is a `nat` literal such as 3 or 5.

Cloning constructors are mainly used to quickly create instances for defined types, such as

```
1  Definition Zmn := ('Z_m * 'Z_n)%type.
```

While ssralg and zmodp provide the instances type inference needs to synthesize a ring structure for `Zmn`, Coq has to expand the definition of `Zmn` to do so. Declaring `Zmn`-specific instances will avoid such spurious expansions, and is easy thanks to cloning constructors:

```
1  Canonical Zmn_eqType := [eqType of Zmn].
2  Canonical Zmn_zmodType := [zmodType of Zmn].
3  Canonical Zmn_ringType := [ringType of Zmn].
```

Cloning constructors are also useful to create on-the-fly instances that must be passed explicitly, e.g., when specializing lemmas:

```
1  have Zp_mulrAC := @mulrAC [ringType of 'Z_p].
```

Finally, instances of *join* structures that are just the union of two smaller ones are always created with cloning constructors. For example, `'I_p` is also a finite (explicitly enumerable) type, and the fintype library declares a corresponding `finType` structure instance. This means that `'I_p` should also have an

instance of the `finRingType` join structure (for `p` of the right shape). This is not automatic, but thanks to the cloning constructor requires only one line in zmodp.

```
Canonical Zp_finRingType := [finRingType of 'I_p].
```

7.2 Telescopes

While using and populating a structure hierarchy is fairly straightforward, creating a robust and efficient hierarchy can be more difficult. In this section we explain *telescopes*, one of the simpler ways of implementing a structure hierarchy. Telescopes suffice for most simple — tree-like and shallow — hierarchies, so new users do not necessarily need expertise with the more sophisticated *packed class* organization covered in the next section.

Because of their limitations (covered at the end of this section), telescopes were not suitable for the main type structure hierarchy of the Mathematical Components library, including eqtype, choice, fintype and ssralg. However, as we have seen in section 5.7.2, structures can be used to associate properties to any logical object, not just types, and the `Monoid.law` structure introduced in section 5.7.2 is part of a telescope hierarchy. Recall that `Monoid.law` associates an identity element and Monoid axioms to a binary operator:

```
Module Monoid.
Variables (T : Type) (idm : T).
Structure law := Law {
  operator : T -> T -> T;
  _ : associative operator;
  _ : left_id idm operator;
  _ : right_id idm operator
}.
Coercion operator : law >-> Funclass.
```

The `Coercion` declaration facilitates writing generic `bigop` simplification rules such as

```
  big1_eq R (idx : R) (op : Monoid.law idx) I r (P : pred I) :
    \big[op/idx]_(i <- r | P i) idx = idx
```

Because the `Monoid` hierarchy is small, there is no need to bundle the `Monoid.law` properties in a mixin. It thus takes only one line to declare an instance for the boolean "and" operator `andb`

```
Canonical andb_monoid := Law andbA andTb andbT.
```

This declaration makes it it possible to use `big1_eq` to simplify the "big and" expression

```
\big[and/true]_(i in A) true
```

to just `true`,[1] as it informs type inference how to solve the unification problem `(operator ?`$_L$`)` versus `andb`, by setting `?`$_L$ to `andb_monoid`.

Now many of the more interesting `bigop` properties permute the operands of the iterated operator; for example

```
pair_bigA: \big[op/idx]_i \big[op/idx]_j F i j = \big[op/idx]_p F p.1 p.2
```

Such properties only hold for commutative monoids, so, in order to state `pair_bigA` as above, we need a structure encapsulating commutative monoids — and one that builds on `Monoid.law` at that, to avoid mindless duplication of theories. The most naïve way of doing this, merely combining a `law` with a commutativity axiom, works remarkably well. After

```
Structure com_law := ComLaw {
   com_operator : law;
   _ : commutative com_operator
}.
Coercion com_operator : com_law >-> law.
```

At this stage, one can declare the canonical instance:

```
Canonical andb_comoid := ComLaw andbC.
```

and then use `big_pairA` to factor nested "big ands" such as

```
\big[andb/true]_i \big[andb/true]_j M i j.
```

However, things are not so simple on closer examination: the idiom only works because of the invisible coercions inserted during type inference.

The definition of `andb_comoid` infers the implicit `com_operator` argument `?`$_L$ of `ComLaw` by unifying the expected statement `commutative (operator ?`$_L$`)` of the commutativity property, with the actual statement of `andbC : commutative andb`. This finds `?`$_L$ to be `andb_monoid` as above.

More importantly, the `op` in the statement of `big_pairA` really stands for `operator (com_operator op)`. Thus applying `big_pairA` to the term above leads to the unification of `operator (com_operator ?`$_C$`)` with `andb`. This first resolves the outer `operator`, as above, reducing the problem to unifying `com_operator ?`$_C$ with `andb_monoid`, which is finally solved using `andb_comoid`. Hence, `andb_comoid` associates commutativity with the `law andb_monoid` rather than the operator `andb`, and the invisible chain of coercions guides the instance resolution.

The telescope idiom works recursively for arbitrarily deep hierarchies, though the `Monoid` one only has one more level for a distributivity property

```
Structure add_law (mul : T -> T -> T) := AddLaw {
  add_operator : com_law;
  _ : left_distributive mul add_operator;
  _ : right_distributive mul add_operator
}.
Coercion add_operator : add_law >-> com_law.
```

[1] There is no spurious `add_monoid` because the identity element is a manifest field stored in the structure type.

The instance declaration

```
Canonical andb_addoid := AddLaw orb_andl orb_andr.
```

then associates distributivity to the `andb_comoid` structure which is inferred by the two-stage resolution process above, and applying the iterated distributivity

```
bigA_distr_bigA :
  \big[times/one]_(i : I) \big[plus/zero]_(j : J) F i j
    = \big[plus/zero]_(f : {ffun I -> J}) \big[times/one]_i F i (f i).
```

to `\big[andb/true]_i /big[orb/false]_j M i j` involves three-stage resolution.

The coercion chains that support the ease of use of telescope hierarchies have unfortunately two major drawbacks: they limit the shape of the hierarchy to a tree (with linear ancestry) and trigger crippling inefficiencies in the type inference and type checking heuristics for deep hierarchies.

7.3 Packed classes (⋆)

The type structure hierarchy for the Mathematical Components library is both deep (up to 11 levels) and non-linear. It is not uncommon for an algebraic structure to combine the properties of two unrelated structures: for example an algebra is both a ring and a module, neither of which is an instance of the other. Thus the Mathematical Components type structures are organized along a different pattern, *packed classes*, which is more flexible and efficient than the telescope pattern, but requires more work to follow.

The packed class design calls for three layers of records for each structure: a *mixin* record holding the new operations and properties the structure adds to the structures it extends (as in section 7.1), a *class* record holding all the primitive operations and properties in the structure, including those in substructures, and finally a *packed class* record that associates the class to a type, and is used to define instances of the structure. Crucially, in this organization, the type "key" that directs inference is always a direct field of a structure's instance record, so all coercion chains have length one.

This arrangement was already hinted at in section 5.4 while commenting on the formalisation of the `eqType` structure, which we recall here:

```
Module Equality.

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

Record mixin_of T := Mixin {op : rel T; _ : axiom op}.
Notation class_of := mixin_of.

Structure type := Pack {sort :> Type; class : class_of sort}.
...
Module Exports.
Coercion sort : type >-> SortClass.
Notation eqType := type.
...
End Equality.
Export Equality.Exports.
```

The `Exports` submodule, which we had omitted in section 5.4, regroups all the declarations in `Equality` that should have global scope, such as the `Coercion` declaration for `Equality.sort`.

The roles of mixin and class are conflated for `eqType` because it sits at the bottom of the type structure hierarchy. To clarify the picture, we need to move one level up, to the `choiceType` structure that provides effective choice for decidable predicates:

```
Module Choice.

Record mixin_of T := Mixin {
  find : pred T -> nat -> option T;
  _ : ∀ P n x, find P n = Some x -> P x;
  _ : ∀ P : pred T, (exists x, P x) -> exists n, find P n;
  _ : ∀ P Q : pred T, P =1 Q -> find P =1 find Q
}.

Record class_of T :=
  Class {base : Equality.class_of T; mixin : mixin_of T}.

Structure type := Pack {sort; _ : class_of sort}.
```

The main operation provided by the `choiceType` structure `T` is a choice function for decidable predicates `(choose : pred T -> T -> T)` satisfying:

```
Lemma chooseP P x0 : P x0 -> P (choose P x0).
Lemma choose_id P x0 y0 : P x0 -> P y0 -> choose P x0 = choose P y0.
Lemma eq_choose P Q : P =1 Q -> choose P =1 choose Q.
```

The mixin actually specifies a specific, depth-based, strategy for searching and electing a witness: `choose` can be defined using `find` and the axiom of countable choice, which is derivable in Calculus of Inductive Constructions.

```
Lemma find_ex_minn (P : pred nat)  :
  (exists n, P n) -> {m | P m & ∀ n, P n -> n >= m}.
```

The stronger requirement makes it possible to compose `choiceTypes`, so that pairs or sequences of `choiceTypes` are also `choiceTypes`. This subtlety is detailed in the

comments of the choice file.

An important difference to telescopes is that the definition of `Choice.type` does not link it directly to `eqType`: a `choiceType` structure contains an `Equality.class_of` record, rather than an `eqType` structure. That link needs to be constructed explicitly in the code that follows the definition of `Choice.type`:

```
Coercion base : class_of >-> Equality.class_of.
Coercion mixin : class_of >-> mixin_of.
Coercion sort : type >-> Sortclass.
Variables (T : Type) (cT : type).
Definition class := let: @Pack _ c as cT' := cT return class_of cT' in c.
Definition eqType := Equality.Pack class.
```

Here `class` is just the explicit definition of the second component of the `Section` variable `cT : type`.

Thanks to the `Coercion` declarations, the `eqType` definition is indeed the `eqType` structure associated to `cT`, with `sort` equal to `cT` $\equiv$ `sort cT` and `class` equal to `base class`. The actual link between `choiceType` and `eqType` is established by the following two lines in `Choice.Exports`:

```
1  Coercion eqType : type >-> Equality.type.
2  Canonical eqType.
```

Line 1 merely lets us explicitly use a `choiceType` where an `eqType` is expected, which is rare as structures are almost alway implicit and inferred. It is line 2 that really lets `choiceType` extend `eqType`, because it makes it possible to use any *element* `(T : choiceType)` as an element of an `eqType`, namely `Choice.eqType T`: it tells type inference that `Choice.sort T` can be unified with `Equality.sort` $?_E$ by taking $?_E$ = `Choice.eqType T`.

The bottom structure in the Mathematical Components algebraic hierarchy introduced by the ssralg library is `zmodType` (`GRing.Zmodule.type`); it encapsulates additive groups, and directly extends the `choiceType` structure.

```
1  Module GRing.
2
3  Module Zmodule.
4
5  Record mixin_of (V : Type) : Type := Mixin {
6    zero : V; opp : V -> V; add : V -> V -> V;
7    _ : associative add;
8    ...}.
9
10 Record class_of T := Class { base: Choice.class_of T; mixin: mixin_of T }.
11 Structure type := Pack {sort; _ : class_of sort}.
```

Strictly speaking, the Mathematical Components algebraic structures don't really *have* to extend `choiceType`, but it is very convenient that they do. We can use `eqType` and `choiceType` operations to test for 0 in fields, or choose a basis of a subspace, for example. Furthermore, this is essentially a free assumption, because the Mathematical Components algebra mixins specify *strict* identities, such as `associative add` on line 7 above. In the pure Calculus of Inductive Constructions, these can only be realized for concrete data types with a binary

representation, which are both discrete and countable, hence are `choiceType`s. On the other hand, the "classical Calculus of Inductive Constructions" axioms needed to construct, e.g., real numbers, imply that all types are `choiceType`s.

Similarly to the definition of `eq_op` in eqtype, the operations afforded by `zmodType` are defined just after the `Zmodule` module.

```
Definition zero V := Zmodule.zero (Zmodule.class V).
Definition opp V := Zmodule.opp (Zmodule.class V).
Definition add V := Zmodule.add (Zmodule.class V).
Notation "+%R" := (@add V).
```

These are defined inside the `GRing` module that encloses most of ssralg, and also given the usual arithmetic syntax (`0`, `- x`, `x + y`) in the `%R` scope. Only the notations are exported from `GRing`, as these definitions are intended to remain private.

Warning

If you see an undocumented `GRing.something`, then you have broken an abstraction barrier

The next structure in the hierarchy encapsulates nontrivial rings. Imposing the non-triviality condition $1 \neq 0$ is a compromise: it greatly simplifies the theory of polynomials (ensuring for instance that X has degree 1), at the cost of ruling out possibly trivial matrix rings.

```
Module Ring.

Record mixin_of (R : zmodType) : Type := Mixin {
  one : R;
  mul : R -> R -> R;
  _ : associative mul;
  _ : left_id one mul;
  _ : right_id one mul;
  _ : left_distributive mul +%R;
  _ : right_distributive mul +%R;
  _ : one != 0
}.

Record class_of (R : Type) : Type := Class {
  base : Zmodule.class_of R;
  mixin : mixin_of (Zmodule.Pack base)
}.

Structure type := Pack {sort; _ : class_of sort}.
```

Unlike `choiceType` and `zmodType`, the definition of the `ringType` mixin depends on the `zmodType` structure it extends. Observe how the class definition instantiates the mixin's `zmodType` parameter with a record created on the fly by packing the representation type with the base class.

This additional complication does not affect the hierarchy declarations. These follow exactly the pattern we saw for `choiceType`, except that we have three definitions, one for each of the three structures `ringType` extends. They all look identical, thanks to the hidden `XXX.base` coercions.

```
Definition eqType := Equality.Pack class.
Definition choiceType := Choice.Pack class.
Definition zmodType := Zmodule.Pack class.
```

Two structures extend rings independently: `comRingType` provides multiplication commutativity, and `unitRingType` provides computable inverses for all units (i.e., invertible elements) along with a test of invertibility. These structures are incomparable, and there are reasonable instances of each: 2×2 matrices over $\mathbb{Q}$ have computable inverses but do not commute, while polynomials over $\mathbb{Z}_p$ commute but do not have easily computable inverses. The definition of `comRingType` and `unitRingType` follow exactly the pattern we have seen, except there is no need for a `ComRing.mixin_of` record.

Since there are also rings such as $\mathbb{Z}_p$ that commute *and* have computable inverses, and properties such as $(x/y)^n = x^n/y^n$ that hold only for such rings, ssralg provides a `comUnitRingType` structure for them. Although this structure simultaneously extends two unrelated structures, it is easy to define using the packed class pattern: we just reuse the `UnitRing` mixin.

```
Module ComUnitRing.

Record class_of (R : Type) : Type := Class {
  base : ComRing.class_of R;
  mixin : UnitRing.mixin_of (Ring.Pack base)
}.

Structure type := Pack {sort; _ : class_of sort}.
```

Since we construct explicitly the links between structures with the packed class pattern, the fact that the hierarchy is no longer a tree is not an issue.

```
1 Definition eqType := Equality.Pack class.
2 Definition choiceType := Choice.Pack class.
3 Definition zmodType := Zmodule.Pack class.
4 Definition ringType := Ring.Pack class.
5 Definition comRingType := ComRing.Pack class.
6 Definition unitRingType := UnitRing.Pack class.
7 Definition com_unitRingType := @UnitRing.Pack comRingType class.
```

In the above code, lines 1–6 relate the new structure to each of the six structures it extends, just as before. Line 7 is needed because `comUnitRingType` has several direct ancestors in the hierarchy. Making `ComUnitRing.com_unitRingType` a canonical `unitRingType` instance tells type inference that it can unify `UnitRing.sort` $?_U$ with `ComRing.sort` $?_C$, by unifying $?_U$ with `ComUnitRing.com_unitRingType` $?_R$ and $?_C$ with `ComUnitRing.comRingType` $?_R$, where $?_R$ `: comUnitRingType` is a fresh unification variable. In other words, the `ComUnitRing.com_unitRingType` instance

says that `comUnitRingType` is the join of `comRingType` and `unitRingType` in the structure hierarchy (see also [16, section 5]).

If a new structure S extends structures that are further apart in the hierarchy more than one such additional link may be needed: precisely one for each pair of structures whose join is S. For example, `unitRingAlgebraType` requires three such links, while `finFieldType` in library `finalg` requires 11. It is highly advisable to map out the hierarchy when simultaneously extending multiple structures.

Finally, the telescope and packed class design patterns are not at all incompatible: it is possible to extend a packed class hierarchy with telescopes (library `fingroup` does this), or to add explicit "join" links to a telescope hierarchy (ssralg does this for its algebraic predicate hierarchy).

7.4 Parameters and constructors $(\star\star)$

We have noted already that structure instances are often hard to provide explicitly because it is intended they always be inferred. For example the explicit `ringType` structure for `int * rat` is

```
pair_ringType int_ringType rat_ringType.
```

Inference usually happens when an element x of the structure is passed explicitly; unifying the actual type of x with its expected type — the sort of the unspecified structure — then triggers the search for a canonical instance. Unfortunately there are two common situations where a structure is required and no element is at hand:

- in a type parameter
- when constructing an instance explicitly.

The first case occurs in ssralg for structure types for modules and algebras, which depend on a ring of scalars: we would like to specify the type of scalars, and infer its `ringType`. We have seen in section 5.10 how to do this, using the `phantom` type

```
Inductive phantom T (p : T) := Phantom.
Arguments phantom : clear implicits.
```

Here we can use a simpler type, equivalent to `phantom Type`

```
Inductive phant (p : Type) := Phant.
```

In the definition of the structure `type` for left modules, which depends on `ringType` parameter `R`, we add a dummy `phant R` parameter `phR`.

```
Module Lmodule.

Variable R : ringType.

Record mixin_of (R : ringType) (V : zmodType) := Mixin {
  scale : R -> V -> V;
  ...}.

Record class_of V := Class {
  base : Zmodule.class_of V;
  mixin : mixin_of R (Zmodule.Pack base)
}.

Structure type (phR : phant R) := Pack {sort; _ : class_of sort}.
```

Then the `Phant` constructor readily yields a value for `phR`, from just the sort of `R`. Hiding the call to `Phant` in a `Notation`

```
Notation lmodType R := (type (Phant R)).
```

allows us to write `V : lmodType (int * rat)` and let type inference fill in the unsightly expression `pair_ringType int_ringType rat_ringType` for `R`.

Inference for constructors is more involved, because it has to produce bespoke classes and mixins subject to dependent typing constraints. While it is in principle possible to program this using dependent matching and transport, the complexity of doing so can be daunting.

Instead, we propose a simpler, static solution using a combination of phantom and function types:

```
Definition phant_id T1 T2 v1 v2 := phantom T1 v1 -> phantom T2 v2.
```

For example, each packed class contains exactly the same definition of the clone constructor, following the introduction of section variables `T` and `cT`, and the definition of `class`:

```
Definition clone c & phant_id class c := @Pack T c.
```

Recall that with the SSREFLECT extension to COQ, `& T` introduces an anonymous parameter of type `T`. As for `lmodType` above we use `Notation` to supply the identity function for this dummy functional parameter

```
Notation "[ 'choiceType' 'of' T ]" := (@clone T _ _ id).
```

In the context of `Definition NN := nat`, `[choiceType of NN]` will by construction return a `choiceType` instance with sort *exactly* `NN` — provided it is well typed.

Now type checking `(@clone NN _ _ id)` will try to give `id` $\equiv$ `(fun x => x)` the type `(phant_id (Choice.class` $?_{cT}$`) ` $?_c$`)`. It will assign `x` the type `(phantom (Choice.sort` $?_{cT}$`) (Choice.class` $?_{cT}$`))`, which it will then unify with `(phantom NN` $?_c$`)`. To do so `Choice.sort` $?_{cT}$ will first be unified with `NN`, by setting $?_{cT}$ to the canonical instance `nat_choiceType` found by unfolding the definition of `NN`, then setting $?_c$ to `Choice.class nat_choiceType`.

The code for the instance constructor for `choiceType` is almost identical, be-

cause it only extends `eqType` with a mixin that does not depend on `eqType`. Note that this definition allows Coq to infer `T` from `m`.

```
Definition pack T m :=
  fun bT b & phant_id (Equality.class bT) b => Pack (@Class T b m).
Notation ChoiceType T m := (@pack T m _ _ id).
```

For `ringType` we use a second `phant_id id` parameter to check the dependent type constraint on the mixin.

```
Definition pack T b0 (m0 : mixin_of (@Zmodule.Pack T b0)) :=
  fun bT b & phant_id (Zmodule.class bT) b =>
  fun    m & phant_id m0 m => Pack (@Class T b m).
Notation "[ 'ringType' 'of' T ]" := (@pack T _ m _ _ id _ _ id)
```

Type-checking the second `id` will set `m` to `m0` after checking that the inferred base class `b` ≡ `Zmodule.class bT` coincides with the actual base class `b0` in the structure parameter of the type of `m0`. Forcing the sort of that parameter to be equal to `T` allows Coq to infer `T` from `m`.

The instance constructor for the join structure `comUnitRingType` uses a similar projection-by-unification idiom to extract a mixin of the appropriate type from the inferred `unitRingType` of a given type `T`. This is the only constructor for `comUnitRingType`.

```
Definition pack T :=
  fun bT b & phant_id (ComRing.class bT) (b : ComRing.class_of T) =>
  fun mT m & phant_id (UnitRing.class mT) (@UnitRing.Class T b m) =>
  Pack (@Class T b m).
Notation "[ 'comUnitRingType' 'of' T ]" := (@pack T _ _ id _ _ id)
```

Finally, the instance constructor for the left algebra structure `lalgType`, a join structure with an additional axiom and a `ringType` parameter, uses all the patterns discussed in this section, using a `phant` and three `phant_id` arguments.

```
Definition pack T b0 mul0 (axT: @axiom R (@Lmodule.Pack R _ T b0 T) mul0):=
 fun bT b & phant_id (Ring.class bT) (b : Ring.class_of T) =>
 fun mT m & phant_id (@Lmodule.class R phR mT) (@Lmodule.Class R T b m) =>
 fun ax & phant_id axT ax =>
 Pack (Phant R) (@Class T b m ax) T.
...
Notation LalgType R T a := (@pack _ (Phant R) T _ _ a _ _ id _ _ id _ id).
```

The interested reader can also refer to [16, section 7] for a description of this technique.

7.5 Linking a custom data type to the library

The sub-type kit of chapter 6 is not the only way to easily add instances to the library. For example imagine we are interested to define the type of a wind rose and attach to it the theory of finite types.

```
Inductive windrose := N | S | E | W.
```

The most naive way to show that `windrose` is a `finType` is to provide a comparison function, then a choice function, ... finally an enumeration. Instead, it is much simpler to show one can punt `windrose` in bijection with a pre-existing finite type, like `'I_4`. Let us start by defining the obvious injections.

```
Definition w2o (w : windrose) : 'I_4 :=
  match w with
  | N => inord 0 | S => inord 1 | E => inord 2 | W => inord 3
  end.
```

Remark how the `inord` constructor lets us postpone the (trivial by computation) proofs that 0, 1, 2, 3 are smaller than 4.

The type of ordinals is larger; hence we provide only a partial function.

```
Definition o2w (o : 'I_4) : option windrose :=
  match val o with
  | 0 => Some N | 1 => Some S | 2 => Some E | 3 => Some W
  | _ => None
  end.
```

Then we can show that these two functions cancel out.

```
Lemma pcan_wo4 : pcancel w2o o2w.
Proof. by case; rewrite /o2w /= inordK. Qed.
```

Now, thanks to the `PcanXXXMixin` family of lemmas, one can inherit on `windrose` the structures of ordinals.

```
Definition windrose_eqMixin := PcanEqMixin pcan_wo4.
Canonical windrose_eqType := EqType windrose windrose_eqMixin.
Definition windrose_choiceMixin := PcanChoiceMixin pcan_wo4.
Canonical windrose_choiceType := ChoiceType windrose windrose_choiceMixin.
Definition windrose_countMixin := PcanCountMixin pcan_wo4.
Canonical windrose_countType := CountType windrose windrose_countMixin.
Definition windrose_finMixin := PcanFinMixin pcan_wo4.
Canonical windrose_finType := FinType windrose windrose_finMixin.
```

Only one tiny detail is left on the side. To use `windrose` in conjunction with `\in` and `#|...|`, the type declaration has to be tagged as a `predArgType` as follows.

```
Inductive windrose : predArgType := N | S | E | W.
```

After that, our new data type can be used exactly as the ordinal one can.

```
Check (N != S) && (N \in windrose) && (#| windrose | == 4).
```

More in general a data type can be equipped with a `eqType`, `choiceType`, and `countType` structure by providing a correspondence with the generic tree data type `(GenTree.tree T)`: an n-ary tree with nodes labelled with natural numbers and leafs carrying a value in `T`.

Part III

Indexes

Concepts

Ssreflect Tactics

Definitions and Notations

Coq Commands

Bibliography

[1] Andrea Asperti, Wilmer Ricciotti, Claudio Sacerdoti Coen, and Enrico Tassi. A bi-directional refinement algorithm for the Calculus of (Co)Inductive Constructions. *Logical Methods in Computer Science*, 8(1), 2012.

[2] H. Barendregt and H. Geuvers. Proof assistants using dependent type systems. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume 2, chapter 18, pages 1149 – 1238. Elsevier, 2001.

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

[4] Nicolas Bourbaki. *Theory of sets*. Elements of Mathematics (Berlin). Springer-Verlag, Berlin, 2004. Reprint of the 1968 English translation [Hermann, Paris; MR0237342].

[5] Adam Chlipala. *Certified Programming with Dependent Types*. The MIT Press, 2014.

[6] Cyril Cohen. Construction of real algebraic numbers in coq. In *Interactive Theorem Proving - Third International Conference, ITP 2012, Princeton, NJ, USA, August 13-15, 2012. Proceedings*, pages 67–82, 2012.

[7] Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. Cubical Type Theory: a constructive interpretation of the univalence axiom, 2015. Preprint.

[8] Thierry Coquand and Gérard Huet. The calculus of constructions. *Information and Computation*, 76(2-3):95–120, 1988.

[9] Thierry Coquand and Christine Paulin-Mohring. Inductively defined types. In *Colog'88*, volume 417 of *LNCS*. Springer-Verlag, 1990.

[10] François Garillot, Georges Gonthier, Assia Mahboubi, and Laurence Rideau. Packaging Mathematical Structures. In Tobias Nipkow and Christian Urban, editors, *Theorem Proving in Higher Order Logics*, volume 5674 of *Lecture Notes in Computer Science*, Munich, Germany, 2009. Springer.

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

[12] Georges Gonthier, Andrea Asperti, Jeremy Avigad, Yves Bertot, Cyril Cohen, François Garillot, Stéphane Le Roux, Assia Mahboubi, Russell O'Connor, Sidi Ould Biha, Ioana Pasca, Laurence Rideau, Alexey Solovyev, Enrico Tassi, and Laurent Théry. A Machine-Checked Proof of the Odd Order Theorem. In Sandrine Blazy, Christine Paulin, and David Pichardie, editors, *ITP 2013, 4th Conference on Interactive Theorem Proving*, volume 7998 of *LNCS*, pages 163–179, Rennes, France, July 2013. Springer.

[13] Georges Gonthier, Assia Mahboubi, and Enrico Tassi. A Small Scale Reflection Extension for the Coq system. Research Report RR-6455, Inria Saclay Ile de France, 2015.

[14] Georges Gonthier, Beta Ziliani, Aleksandar Nanevski, and Derek Dreyer. How to make ad hoc proof automation less ad hoc. *J. Funct. Program.*, 23(4):357–401, 2013.

[15] Michael Hedberg. A coherence theorem for Martin-Löf's Type Theory. *J. Funct. Program.*, 8(4):413–436, July 1998.

[16] Assia Mahboubi and Enrico Tassi. Canonical Structures for the working Coq user. In Sandrine Blazy, Christine Paulin, and David Pichardie, editors, *ITP 2013, 4th Conference on Interactive Theorem Proving*, volume 7998 of *LNCS*, pages 19–34, Rennes, France, July 2013. Springer.

[17] Rob Nederpelt and Herman Geuvers. *Type theory and formal proof*. Cambridge University Press, Cambridge, 2014. An introduction, With a foreword by Henk Barendregt.

[18] Rob Nederpelt, Herman Geuvers, and Roel de Vrijer, editors. *Selected Papers on Automath*, volume 133 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, 1994.

[19] 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 Lecture Notes in Computer Science, 1993. LIP research report 92-49.

[20] Christine Paulin-Mohring. Introduction to the Calculus of Inductive Constructions. In Bruno Woltzenlogel Paleo and David Delahaye, editors, *All about Proofs, Proofs for All*, volume 55 of *Studies in Logic (Mathematical logic and foundations)*. College Publications, January 2015.

[21] Benjamin C. Pierce, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjoberg, and Brent Yorgey. *Software Foundations*. Electronic textbook, 2015.

[22] Robert Pollack. Implicit syntax. In *Informal Proceedings of First Workshop on Logical Frameworks*, page pages, 1992.

[23] Matthieu Sozeau and Beta Ziliani. Towards a better behaved unification algorithm for Coq. In *Proceedings of The 28th International Workshop on Unification*, 2014.

[24] The Coq development team. *The Coq proof assistant reference manual*. Inria.

[25] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. https://homotopytypetheory.org/book, Institute for Advanced Study, 2013.