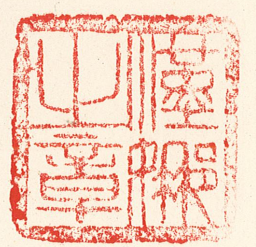


# Mathematical Components

Assia Mahboubi and Enrico Tassi

with contributions by Yves Bertot and Georges Gonthier



Copyright © 2020 Yves Bertot and Georges Gonthier and Assia Mahboubi and Enrico Tassi

[HTTPS://MATH-COMP.GITHUB.IO/MCB/](https://math-comp.github.io/mcb/)

Licensed under the Creative Commons Attribution-NonCommercial 3.0 Unported License (the “License”). You may not use this file except in compliance with the License. You may obtain a copy of the License at <http://creativecommons.org/licenses/by-nc/3.0>. Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an “AS IS” BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

*Draft version Thu, 14 Jan 2021 17:51:56 +0100 , zenodo-1.0.0-5-gc32cc2c*



## 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 [Gon08] and of the Odd Order Theorem [Gon+13a].

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 [BC04; Ch14; Pie+15] and the theory it is based on [Coq][Pau15; Uni13], 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 book targets two classes of public. On the 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 [Coq; GMT15]. 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.**

We thank Yves Bertot and Georges Gonthier, for contributing the first and the last chapter of this book. We wish to thank Reynald Affeldt, Guillaume Allais, Sophie Bernard, Simon Boulier, Cyril Cohen, Arthur Charguéraud, Alain Giorgetti, Darij Grinberg, Florent Hivert, Pierre Jouvelot, Marisa Kirisame, Guillaume Melquiond, Sebastian Miele, Yamamoto

Mitsuharu, Prashanth Mundkur, Michael Nahas, Julien Narboux, Laurence Rideau, Lionel Rieg, Damien Rouhling, Michael Soegtrop, Laurent Théry, and Anton Trunov for their careful proofreading and for their suggestions. Many thanks to Hanna for the illustrations.

## Structure of the book

The book has two parts which are meant to be read in order.

### Part 1: Languages for writing formal mathematics

This part introduces two languages and incidentally a formalization approach.

The first language is used to represent mathematics in the COQ proof assistant. It is called *Gallina* and, as the 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 language is used to write proofs and is called *SSReflect*, a shorthand for Small Scale Reflection. *SSReflect* is a language designed to ease the activity of writing and maintaining formal proofs. In particular the maintenance of large formal libraries requires a solid writing discipline and a language that supports it. *SSReflect* provides linguistic constructs well adapted to writing scripts that can be easily fixed in response to changes to the contents of the formal libraries.

Actually, Small Scale Reflection is firstly the name of a formalization methodology, sometimes also called *Boolean Reflection*. 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* proof language was named after this methodology because of the support it provides for its implementation.

### Part 2: Crafting a formal library

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 pieces of information, as the 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

Advice one should keep in mind are signaled as follows:

 Remember this advice.

Tricky details typically overlooked by beginners are signaled as follows:

 Mind this detail.

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

```
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 -divnMD1 //.
5 by rewrite mulnS muln1 -addnA -mulSn -mulnS.
6 Qed.
```

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

```
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 library 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/>.

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 Visual Studio Code programming environment (known as `vscoq`). 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:

```
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

I	<b>Languages for writing formal mathematics</b>	
<b>1</b>	<b>Functions and computation</b> .....	<b>13</b>
1.1	Functions	13
1.2	Data types, first examples	19
1.3	Containers	27
1.4	The Section mechanism	33
1.5	Symbolic computation	34
1.6	Iterators and mathematical notations	36
1.7	Notations, abbreviations	37
<b>2</b>	<b>First steps in formal proofs</b> .....	<b>41</b>
2.1	Formal statements	41
2.2	Formal proofs	45
2.3	Quantifiers	53
2.4	Rewrite, a Swiss army knife	62
2.5	Searching the library	66
<b>3</b>	<b>Dependent type theory</b> .....	<b>69</b>
3.1	Propositions as types, proofs as programs	69
3.2	Terms, types, sorts	71
3.3	Propositions, implication, universal quantification	74
3.4	Conversion	75

3.5	Inductive types	75
3.6	More connectives	77
3.7	Inductive reasoning	80
<b>4</b>	<b>A proof language for formal proofs</b> .....	<b>83</b>
4.1	Bookkeeping: goals as stacks	84
4.2	Structuring proofs, by examples	89
4.3	Proof maintenance: a matter of style	94
<b>5</b>	<b>Inductive specifications</b> .....	<b>99</b>
5.1	Reflection views	100
5.2	Advanced, practical, statements	106
5.3	Strong induction via inductive specs	108
5.4	Showcase: Euclidean division, simple and correct	110
5.5	Notational aspects of specifications	111

## II

## Crafting a formal library

<b>6</b>	<b>Implicit parameters</b> .....	<b>115</b>
6.1	Type inference and higher-order unification	116
6.2	Type inference by example	117
6.3	Records as relations	119
6.4	Records as (first-class) interfaces	123
6.5	Using a generic theory	125
6.6	The generic theory of sequences	126
6.7	The generic theory of “big” operators	128
6.8	Stable notations for big operators	134
6.9	Working with overloaded notations	135
6.10	Ad-hoc polymorphism	136
<b>7</b>	<b>Sub-types</b> .....	<b>137</b>
7.1	$n$ -tuples, lists with an invariant on the length	138
7.2	$n$ -tuples, a sub-type of sequences	141
7.3	Finite types and their theory	144
7.4	The ordinal subtype	145
7.5	Finite functions	146
7.6	Finite sets	148
7.7	Permutations	149
7.8	Matrix	150



<b>8</b>	<b>Organizing Theories</b> .....	<b>155</b>
<b>8.1</b>	<b>Structure interface</b>	<b>155</b>
<b>8.2</b>	<b>Telescopes</b>	<b>158</b>
<b>8.3</b>	<b>Packed classes</b>	<b>160</b>
<b>8.4</b>	<b>Parameters and constructors</b>	<b>165</b>
<b>8.5</b>	<b>Linking a custom data type to the library</b>	<b>167</b>

**III**

## **SSReflect Cheat Sheet**

**IV**

## **Indexes and Bibliography**

<b>Concepts</b> .....	<b>177</b>
<b>Ssreflect Tactics</b> .....	<b>179</b>
<b>Definitions and Notations</b> .....	<b>181</b>
<b>Coq Commands</b> .....	<b>183</b>
<b>Bibliography</b> .....	<b>185</b>





# Languages for writing formal mathematics

<b>1</b>	<b>Functions and computation</b> .....	<b>13</b>
1.1	Functions	
1.2	Data types, first examples	
1.3	Containers	
1.4	The Section mechanism	
1.5	Symbolic computation	
1.6	Iterators and mathematical notations	
1.7	Notations, abbreviations	
<b>2</b>	<b>First steps in formal proofs</b> .....	<b>41</b>
2.1	Formal statements	
2.2	Formal proofs	
2.3	Quantifiers	
2.4	Rewrite, a Swiss army knife	
2.5	Searching the library	
<b>3</b>	<b>Dependent type theory</b> .....	<b>69</b>
3.1	Propositions as types, proofs as programs	
3.2	Terms, types, sorts	
3.3	Propositions, implication, universal quantification	
3.4	Conversion	
3.5	Inductive types	
3.6	More connectives	
3.7	Inductive reasoning	
<b>4</b>	<b>A proof language for formal proofs</b> .....	<b>83</b>
4.1	Bookkeeping: goals as stacks	
4.2	Structuring proofs, by examples	
4.3	Proof maintenance: a matter of style	
<b>5</b>	<b>Inductive specifications</b> .....	<b>99</b>
5.1	Reflection views	
5.2	Advanced, practical, statements	
5.3	Strong induction via inductive specs	
5.4	Showcase: Euclidean division, simple and correct	
5.5	Notational aspects of specifications	





# 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. However, those functions are rather different in nature from the functions encountered in the mathematical tradition of set theory.

## 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. We start with a collection of examples involving natural numbers. For this purpose, we casually use the COQ term `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 also assume implicitly that the COQ symbols `0`, `1`, `2` represent natural numbers which represent the corresponding numerals `0`, `1`, `2`, `...`

### 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 `+` in the expression:

$$2 + 1.$$

This expression represents a natural number, obtained as the result of an operation applied to its arguments. More precisely, 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:

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

The textual transformation of expression  $(2 + 1)$  into an explicit application of this function to one argument can be described as follows: select the sub-expression 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 function to the argument 2, and this is written as:

```
1 (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 in the language of COQ, also called *terms*, are either variables, constants, functions of the form  $(\text{fun } x \Rightarrow e)$ , where  $e$  is itself an expression, or the application  $(e_1 e_2)$  of an expression  $e_1$  to another expression  $e_2$ . However, just like with pen and paper, the addition operation is denoted in the COQ language using an infix notation  $+$ . The transformation we just detailed, from expression  $(2 + 1)$  to expression  $(\text{fun } n \Rightarrow n + 1) 2$ , is called *abstracting 2 in  $(2 + 1)$* . As a contrast to traditional set-theoretic approaches, abstraction is essentially the only way to define a function in this language. A function is never described by providing extensively the graph as a subset of the cartesian product of the input and the output.

While expression  $(\text{fun } n \Rightarrow n + 1)$  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:

$$f: \begin{array}{l} \mathbb{N} \rightarrow \mathbb{N} \\ n \mapsto n + 1 \end{array} \tag{1.1}$$

In COQ, the user can also associate a name with the operation  $(\text{fun } n \Rightarrow n + 1)$ , in the following manner:

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

An alternative syntax for exactly the same definition is as follows and this alternative 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 enclosed between the `:=` symbol and the terminating dot.

The information on the domain and codomain of  $f$ , as in  $f : \mathbb{N} \rightarrow \mathbb{N}$ , 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 in fact 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  $(n : \text{nat})$ , 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 done so far. Here is the response to a query about  $f$ :

```
1 About f.
```

```
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. Note that 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. For example the function  $f$  we just defined can *only be applied to a natural number*, i.e., a term of type `nat`. COQ provides a command to check whether an expression is *well typed*: as 3 is a natural number, the following query succeeds:

```
1 Check f 3.
```

```
f 3 : nat
```

But  $f$  cannot be applied to an argument that is not a natural number, like for example a function:

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

```
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 sum of 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:

```
1 Compute f 3.
```

```
= 4 : nat
```

We can observe that in the course of the computation, argument 3 has been substituted for the argument variable in the definition of function `f`, and that the addition has been evaluated. This capability of COQ plays a crucial role in the Mathematical Components library, as we shall see in chapter 2. It is however too early to be more precise about this normalization strategy. This would indeed require describing in more details the formalism underlying the COQ system, which we do only in chapter 3. The interested reader should actually refer to [Coq, section 5.3.7, “Performing computations”] 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 command 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.
```

This 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`:

```
1 About g.
```

```
g : nat -> nat -> nat
```

The first two 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:



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

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

```
1 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.

```
1 Print g.                                     g = fun n m : nat => n + m * 2
2                                             : nat -> nat -> nat
3 Print h.                                     h = fun n m : nat => n + m * 2
4                                             : 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:

```
1 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:

```
1 Compute 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:

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

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

```
1 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.

```

1 Definition repeat_twice (g : nat -> nat) : nat -> nat :=
2   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:

```

1 Compute 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`:

```

1 Check (repeat_twice f).

```

`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.

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

```
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:

```
1 let x := B in
2   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:

```
1 Compute
2   let n := 33 : nat in
3   let e := n + n + n in
4     e + e + e.
```

```
= 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:

```
1 (33 + 33 + 33) + (33 + 33 + 33) + (33 + 33 + 33)
```

However, in practice the evaluation strategy used in a normalization command like `Compute` 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 a combination of variables and symbols of a given language that respects the prescribed arities of the operations. For instance, the expression:

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

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

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

is a well formed expression in the language  $\{0, S, +, \times\}$  of Peano arithmetic. These languages are usually equipped with behavior rules, expressed in the form of equality axioms, like

$$\top \vee b = \top \quad \text{or} \quad 0 + x = x$$

In practice, these axioms confer a distinctive status on the symbols  $\top$  and  $\perp$  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  $\perp$  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 what follows, we shall call the distinctive symbols *constructors*. So the constructors of boolean arithmetic are  $\top$  and  $\perp$  and the constructors of Peano arithmetic are  $0$  and  $S$ .

In COQ, such languages are represented using a *data type*, whose definition provides in a first single declaration the name of the type and the constructors, plus some rules on how to define maps on these data types or prove theorems about their elements. The other operations are then derived from these rules. The first single declarations of data types are introduced by the means of *inductive type definitions*. One can then 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 analysis or a recursive definition on the shape of this argument. This approach is used in a systematic way to define a variety of basic types, among which boolean values, natural numbers, pairs or sequences of values are among the most prominent examples.

In the following sections, we introduce basic types `bool` for boolean values and `nat` for natural numbers, taking the opportunity to describe various constructs of the COQ language as they become relevant. These sections serve simultaneously as an introduction to the data types `bool` and `nat` and to the COQ language constructs that are used to define new data types and describe operations on these data types.

### 1.2.1 Boolean values

The collection  $\mathbb{B} := \{\top, \perp\}$  of boolean values is formalized by a type called `bool`, with two inhabitants `true` and `false`, representing  $\top$  and  $\perp$  respectively. 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:

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

It is actually one of the simplest possible inductive definitions, with only base cases, and no inductive cases. 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`.

```
1 Check true.
```

```
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:

```
1 Compute twoVthree true.           = 2 : nat
2 Compute twoVthree false.         = 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 of the shape `if false then t1 else t2` into `t2`.

We can then use this basic operation to describe the other elements that we usually consider as parts of the boolean language, usually at least conjunction, written `&&` and disjunction, written `||`. We first define these as functions, 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.
```

and then the notations `&&` and `||` are attached to these two functions, respectively.

For any choice of values `b1`, `b2`, and `b3` it appends that the expressions `b1 && (b2 && b3)` and `(b1 && b2) && b3` always compute to the same result. In this sense, the conjunction operation is associative. We shall see in chapter 2 that this known property can be *stated* explicitly. To a practiced pair of eyes, this associativity property is implicitly used in the reading process, so that the two forms of three-argument conjunctions are identified. However, when manipulating boolean expressions of the COQ language, there is a clear distinction between these two forms of conjunctions of three values, and the notation conventions make sure that the two forms appear differently on the screen. We will come back to this question in section 1.7.

The Mathematical Components library provides a collection of boolean operations that model 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.

### 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 `0` (capital “o” letter) representing zero, or an application to an existing natural number of the function symbol `S` representing the successor:

```
1 Inductive nat := 0 | S (n : nat).
```

This inductive definition of the expressions of type `nat` has one base case, the constant `0`, and one inductive case, for the natural numbers obtained using the successor function at

least once: therefore `0` has type `nat`, `(S 0)` has type `nat`, so does `(S (S 0))`, and so on, and any natural number has this shape. The constant `0` and the function `S` are the constructors of type `nat`.

The decimal notations we have used so far are only a parsing and display facility provided to the user for readability: `0` is displayed `0`, `(S 0)` 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 `0` 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 stronger than function application, rendering some parentheses unnecessary: assuming that `f` is a function of type `nat -> nat` the expression `(f n.+1)` reads `(f (S n))`. Notations are also provided for `S (S n)`, written `n.+2`, and so on until `n.+4`.

```
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 again, 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 a toy example:

```
1 Definition non_zero n := if n is p.+1 then true else false.
```

The function `non_zero` returns the boolean value `false` if its argument is `0`, and `true` 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. This value is actually computed at the moment the argument `n` provided to the function is matched against the pattern. For instance:

```
1 Compute non_zero 5.
```

```
= true : bool
```

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, the value of `k.+1` is `5`, thus the value of `k` is `4` and `t1` is `true`.

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 reused in the result part, as in:

```
1 Definition pred n := if n is p.+1 then p else 0.
```

Observe that in the definitions of functions `predn` and `non_zero`, we did omit the type of the input `n`: 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 `true` for every input  $n$  larger than 4 and `false` otherwise, we can write the following definition:

```
1 Definition larger_than_4 n :=
2   if n is u.+1.+1.+1.+1.+1 then true else false.
```

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:

```
1 Definition three_patterns n :=
2   match n with
3     u.+1.+1.+1.+1.+1 => u
4   | v.+1 => v
5   | 0 => n
6   end.
```

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

```
1 Definition wrong (n : nat) :=
2   match n with 0 => true end.
3
```

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

We finish the section by showing a syntactic facility to scrutinize multiple values at the same time. However, this part is not specific to natural numbers, and we use boolean values to illustrate the facility.

```
1 Definition same_bool b1 b2 :=
2   match b1, b2 with
3     | true, true => true
4     | false, false => true
5     | _, _ => false
6   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:

```

1 Definition same_bool b1 b2 :=
2   match b1 with
3   | true => match b2 with true => true | _ => false end
4   | false => match b2 with true => false | _ => true end
5   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:

```

1 Fixpoint addn n m :=
2   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:

<code>(addn 2 3)</code>	use the “then” branch, <code>p = 1</code>
<code>(addn 1 3).+1</code>	use the “then” branch, <code>p = 0</code>
<code>(addn 0 3).+1.+1</code>	use the “else” branch
<code>3.+1.+1</code>	remember that <code>5 = 3.+1.+1</code>

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

If we reflect again on the discussion of Peano arithmetic, as in Section 1.2, we see that addition is provided by the definition of `addn`, and the usual axioms of Peano arithmetic, namely:

$$S x + y = S(x + y) \quad 0 + x = x$$

are actually provided by the computation behavior of the function, namely by the “then” branch and the “else” branch respectively. Therefore, Peano arithmetic is really provided by COQ in two steps, first by providing the type of natural numbers and its constructors thanks to an *inductive type definition*, and then by providing the operations as defined functions (usually recursive functions as in the case of addition). The axioms are not constructed explicitly, but they appear as part of the behavior of the addition function. In practice, it will be possible to create theorems whose statements are exactly the axioms of Peano arithmetic, using the tools provided in Chapter 2. The fact that the computation of



`addn 2 3` ends in an expression where the `addn` symbol does not appear is consistent with the fact that `0` (also noted `o`) and `s` (also noted `.+1`) are the constructors of natural numbers.

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:

```
1 Fixpoint addn n m :=
2   match n with
3   | 0 => m
4   | p.+1 => (addn p m).+1
5   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 3.7. 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 corresponding 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`. The way COQ verifies that recursive functions will terminate is explained in more detail in the reference manual [Coq] or in the Coq’Art book [BC04].

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

```
1 Fixpoint loop n :=
2   if n is 0 then loop n else 0.
3
```

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

We have seen that addition is implemented by repeating the operation of applying `.+1` to one of the arguments. Conversely, comparing two natural numbers is implemented by repeating the operation of fetching a subterm. Consider two terms `m` and `n` representing the natural numbers  $m$  and  $n$  respectively. Then  $m$  is larger than  $n$  if and only if `m` has  $k$  more constructors than `n` for some  $k \geq 0$ , which measures the distance between  $m$  and  $n$ . Comparison is thus implemented in terms of an auxiliary truncated subtraction, which is again easily expressed using pattern matching constructs:

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

Number  $m$  is less or equal to number  $n$  if and only if `(subn m n)` is zero. Here as well, this subtraction is already defined in the libraries, but we play the game of re-defining our own version. The second pattern matching rule indicates that when any argument of the subtraction is 0, then the result is the first argument. This rule thus 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. Another possible view on `subn` is to see it as a subtraction operation on natural numbers, made total by providing a default value in the “exceptional” cases.

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

```
1 Fixpoint eqn m n :=
2   match m, n with
3   | 0, 0 => true
4   | p.+1, q.+1 => eqn p q
5   | _, _ => false
6   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:

```
1 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.

```
1 Compute 0 == 0.
2 Compute 1 == S 0.
3 Compute 1 == 0.+1.
4 Compute 2 == S 0.
5 Compute 2 == 1.+1.
6 Compute 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 (apart from the comparisons) 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 (`predn`) and double (`double`) functions.

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

```
1 Definition leq m n := m - n == 0.
2 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 the source files<sup>1</sup> and not to limit herself to interactive queries to the system. The

<sup>1</sup><http://math-comp.github.io/math-comp/html/doc/libgraph.html>

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.

- Ⓡ 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:

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

The elements of this data type constitute a possible description 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:

```
1 Check consn 1 (consn 2 niln).
2 Check consn true (consn false niln).
3
4
```

```
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`.

```
1 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  $\alpha$  is a schematic variable:

```
1 Inductive list := nil | cons (hd :  $\alpha$ ) (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 somehow be added on top of the COQ language. This way, we will make possible the writing of formal sentences with arbitrary quantifications on this parameter  $\alpha$ .

### 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`.<sup>2</sup>

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:

```
1 ∀ A : Type, seq A
```

so as to *bind* the value `A` of the argument in the type of the output. Even if the  $\forall$  symbol is actually written and displayed as `forall` in Coq, in this book we typeset it usfollowing the standard mathematical notation for that quantifier. 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

<sup>2</sup>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.

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`:

```
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 said to be *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. In the end, this ensures that users can write the following expression.

```
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 _))`.

The reader should refer to the documentation of the `Arguments` command [Coq] to know how to modify the implicit status of an argument).

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

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

```
[:: 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 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.

The Mathematical Components library provides similar notations to iterate constants other than `cons`. For example, one can write the boolean conjunction of three terms as `[&& true, false & true]`, the boolean disjunction as `[|| b1, b2 | b3]` and the boolean

implication as `[==> b1, b2 => b3]`.

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:

```
1 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 stacks 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 again, 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 stacks of constructors.

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

```
1 Fixpoint size A (s : seq A) :=
2   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 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:

```
1 Fixpoint map A B (f : A -> B) s :=
2   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*.

```
1 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:

```
1 Compute
2   [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 either be empty, or contain a single value:

```
1 Inductive option A := None | Some (a : 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 even ones by the `None` default value:

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

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

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

See also the sub-type kit presented in chapter 7, 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:

```
1 Inductive pair (A B : Type) : Type := mk_pair (a : A) (b : B).
2 Notation "( a , b )" := (mk_pair a b).
3 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:

```

1 Definition fst A B (p : pair A B) :=
2   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.

```

1 Check (3, false).
2 Compute (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 it 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 component: it can thus be obtained as `(3,true,4).1.2`. This is a 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.

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.<sup>3</sup> 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 [Coq, “Interpretation scopes”]).

### 1.3.4 Aggregating data in record types

Inductive types with a single constructor, like the type `pair` in section 1.3.3, 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:

<sup>3</sup>In reality the data type of pairs, as it comes with COQ, is called `prod` and its constructor `pair`.



```

1 Compute x (Point 3 0 2).
2 Compute 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.

```

1 Section iterators.
2
3 Variables (T : Type) (A : Type).
4 Variables (f : T -> A -> A).
5
6 Implicit Type x : T.
7
8 Fixpoint iter n op x :=
9   if n is p.+1 then op (iter p op x) else x.
10
11 Fixpoint foldr a s :=
12   if s is y :: ys then f y (foldr a ys) else a.
13
14 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 [Coq, “Extensions of Gallina”] 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.

```
1 Fixpoint iter (T : Type) n op (x : T) :=
2   if n is p.+1 then op (iter p op x) else x.
3 Fixpoint foldr (T A : Type) (f : T -> A -> A) a s :=
4   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.

```
1 About foldr.
2
3
```

```
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]`:

```
1 Compute iter 5 predn 7.
2 Compute foldr addn 0 [:: 1; 2].
```

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

## 1.5 Symbolic computation

As we mentioned in section 1.1, the `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.

```
1 Section iterators.
2
3 Variables (T : Type) (A : Type).
4 Variables (f : T -> A -> A).
5
6 Fixpoint foldr a s :=
7   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).

```
1 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.

```

1 Variable init : A.
2 Variables x1 x2 : T.
3 Compute 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 an 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`:

```
1 Compute 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:

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

Now let us consider the following alternative definition:

```
1 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 `Eval 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 [Coq, section 5.3.7, “Performing computations”] for more details.

```

1 Variable n : nat.
2 Eval simpl in predn (add n.+1 7).
3 Eval simpl in predn (addn n.+1 7).

```

```
= predn (add n 8) : 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 successor symbols `s` given as first argument to its second argument before resorting to its base case, whereas in the `addn` variant, the resulting stack of successor symbols 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:

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

and, more generally, in order to make explicit the meaning of the capital notations like  $\bigcap_{i=1}^n A_i, \prod_{i=1}^n u_i, \dots$ , we need to describe an iteration procedure akin to the `foldr` program. We illustrate this on the example of the summation symbol (for a finite sum):

```
1 Fixpoint iota m n := if n is u.+1 then m :: iota m.+1 u else [::].
2 Notation "\sum_ ( m <= i < n ) F" :=
3   (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<sup>4</sup>.

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.

```
1 Compute
2   \sum_ ( 1 <= i < 5 ) (i * 2 - 1).
3 Compute
4   \sum_ ( 1 <= i < 5 ) i.
```

```
= 16 : nat
```

```
= 10 : nat
```

<sup>4</sup>Although this change results in two syntactically different expressions, they have the same meaning for COQ

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 6.7.

## 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 kinds of notations and to specify their 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 beyond the scope of this book; the interested reader can refer to [Coq, "Syntax extensions and interpretation scopes"].

A notation can denote an arbitrary expression, not just 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).
```



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:

```
1 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.

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

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

```
1 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.

```
1 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 `+.1` 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 L<sup>A</sup>T<sub>E</sub>X 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.







## 2. First steps in formal proofs

In this chapter, we explain how to use the COQ system to state and 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 5.

### 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:

```
1 Check 3 = 3.  
2 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`:

```
1 Locate "=" .
2
3 About eq.
4
```

```
"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 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:

```
1 Check 3 = [:: 3] .
2
```

```
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!

```
1 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.

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

```
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:

```
1 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:

```
1 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.

```

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

```

### 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`.

```

1 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 5.5, 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:

```

1 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 equality test `==` 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:

```

1 Lemma eqn_leq m n : (m == n) = (m <= n) && (n <= m).
2 Lemma neq_ltn m n : (m != n) = (m < n) || (n < m).
3 Lemma leqn0 n : (n <= 0) = (n == 0).
4 Lemma dvdn1 d : (d %| 1) = (d == 1).
5 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:

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

This arrow  $\rightarrow$  is the same as the one we 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 3.2. For now let us only stress that  $\rightarrow$  is right-associative, and therefore a succession of arrows expresses a conjunction of conditions:

```
1 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.

```
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  $3 = 3$ .

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

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  $2 + 1 = 3$ .

```
1 Lemma my_second_lemma : 2 + 1 = 3.
2 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:

```
1 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  $\neg\neg (\neg\neg b) = b$ . The “proof by computation” technique of section 2.2.1 fails in this case:

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

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

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

```
1 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:

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

```
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 only works when `t` belongs to an inductive type. This tactic then performs a case analysis on (the shape of) a term `t`. 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 for each subproof to be constructed, starting with the branch associated with the `true` value. In order to help the reader identify the two parts of the proof script, we indent the first one.

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:

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

```
1 subgoal
=====
  ~~ ~~ false = false
```

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

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

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

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

```
1 Lemma negbK b : ~~ (~~ b) = b.
2 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:

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

```
1 Inductive nat := 0 | S (n : nat).
```

with two constructors, `0` 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 `0` 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 appears in the second branch as the argument of the `S` constructor of type `nat`:

```
1 Lemma leqn0 n : (n <= 0) = (n == 0).
2 Proof.
3 case: n => [| k].
4
5
```

```
=====
(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 easily be solved by computation, as both sides of the equality evaluate to `true`.

```
1 Lemma leqn0 n : (n <= 0) = (n == 0).
2 Proof.
3 case: n => [| k].
4 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:

```
1 Lemma leqn0 n : (n <= 0) = (n == 0).
2 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:

```
1 Fixpoint muln (m n : nat) : nat :=
2   if m is p.+1 then n + muln p n else 0.
3
4 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.

```
1 Lemma muln_eq0 m n :
2   (m * n == 0) = (m == 0) || (n == 0).
3 Proof.
4 case: m => [|m].
5 by [].
```

```
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:

```
1 Lemma muln_eq0 m n :
2   (m * n == 0) = (m == 0) || (n == 0).
3 Proof.
4 case: m => [|m].
5 by [].
6 case: n => [|k]; last first.
7
8
```

```
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)
```

It is a good practice to get rid of the easy subgoal first: since we are going to indent the text of its subproof, it better be the shortest one. In this way the reader can easily skip over the simple case, that is likely to be less interesting than the hard one.

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  $\tau.+1 == 0$ , where  $\tau$  stands for  $(k + (m * k.+1))$ , and this reduces to `false`. In consequence, the successor branch of the second case analysis is trivial by computation.

```

1 Lemma muln_eq0 m n :
2   (m * n == 0) = (m == 0) || (n == 0).
3 Proof.
4 case: m => [|m|.
5   by [].
6 case: n => [|k|; last first.
7   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:

```

1 Lemma muln_eq0 m n :
2   (m * n == 0) = (m == 0) || (n == 0).
3 Proof.
4 case: m => [|m] //.

```

```

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`:

```

1 Lemma muln_eq0 m n :
2   (m * n == 0) = (m == 0) || (n == 0).
3 Proof.
4 case: m => [|m] //.
5 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 [GMT15, 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:

```
1 Lemma muln_eq0 m n :
2   (m * n == 0) = (m == 0) || (n == 0).
3 Proof.
4 case: m => [|m] //.
5 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:

```
1 (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:

```
1 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.

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

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

```
6 rewrite muln0.
```

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<sup>1</sup>:

<sup>1</sup>Passing a single tactic to `by` requires no brackets; i.e., we can write `by rewrite muln0` instead of `by [rewrite muln0]`.

```

1 Lemma muln_eq0 m n : (m * n == 0) = (m == 0) || (n == 0).
2 Proof.
3 case: m => [|m] //.
4 case: n => [|k] //.
5 by rewrite muln0.
6 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_mul21`, which provides a necessary and sufficient condition for the comparison ( $m * n1 \leq m * n2$ ) to hold:

```

1 Lemma leq_mul21 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 equivalent to one of the arguments of the product being zero. And  $(n1 - n2 == 0)$  means  $(n1 \leq n2)$ .

The first step is performed using the following equation:

```

1 Lemma leqE m n : (m <= n) = (m - n == 0).
2 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:

```

1 Lemma leq_mul21 m n1 n2 :
2   (m * n1 <= m * n2) = (m == 0) || ...
3 Proof.
4   rewrite leqE.

```

```

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 `!`:

```

1 Lemma leq_mul21 m n1 n2 :
2   (m * n1 <= m * n2) = (m == 0) || ...
3 Proof.
4   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:

```

1 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 - :

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

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

Consecutive rewrite steps can be chained as follows:

```
1 Lemma leq_mul2l m n1 n2 :
2   (m * n1 <= m * n2) = (m == 0) || ...
3 Proof.
4 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.

```
1 Lemma leq_mul2l m n1 n2 :
2   (m * n1 <= m * n2) = (m == 0) || ...
3 Proof.
4 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 [GMT15].

## 2.3 Quantifiers

### 2.3.1 Universal quantification, first examples

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

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

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

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

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

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

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

```
1 About leq.
```

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

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

```
1 About leqn0.
```

```
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:

```
1 About muln_eq0.
2
```

```
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. 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:

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

Quantifiers are also allowed to range over functions:

```
1 Lemma size_map (T1 T2 : Type) :
2   ∀ (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: “ $\forall (f : T1 \rightarrow T2) (s : seq T1), \dots$ ” is syntactic sugar for “ $\forall f : T1 \rightarrow T2, \forall s : seq T1, \dots$ ”. However, in this case of prenex quantification, we could just as well write:

```
1 Lemma size_map (T1 T2 : Type) (f : T1 -> T2) (s : seq T1) :
2   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:

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

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

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

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

```

1 Section StandardPredicates.
2 Variable T : Type.
3 Implicit Types (op add : T -> T -> T).
4 Definition associative op := ∀ x y z, op x (op y z) = op (op x y) z.
5 Definition left_distributive op add :=
6   ∀ x y z, op (add x y) z = add (op x z) (op y z).
7 Definition left_id e op := ∀ x, op e x = x.
8 End StandardPredicates.

```

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:

```

1 Section MoreStandardPredicates.
2 Variables rT aT : Type.
3 Implicit Types (f : aT -> rT).
4 Definition injective f := ∀ x1 x2, f x1 = f x2 -> x1 = x2.
5 Definition cancel f g := ∀ x, g (f x) = x.
6 Definition pcancel f g := ∀ x, g (f x) = Some x.
7 End MoreStandardPredicates.

```

The types of these predicates deserve a few comments:

```

1 About commutative.
2

```

```

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

```

The constant `commutative` has polymorphic parameters `S` and `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 having the same type. Just like the polymorphic binary predicate `eq`, the predicate `commutative` can be used to form propositions:

```

1 Check 3 = 3.
2 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:

```

1 Section Chinese.
2
3 Variables m1 m2 : nat.
4 Hypothesis co_m12 : coprime m1 m2.
5
6 Lemma chinese_remainder x y :
7   (x == y %[mod m1 * m2]) = (x == y %[mod m1]) && (x == y %[mod m2]).
8 Proof.
9 ...
10 End.
11
12 End Chinese.

```

The part of this excerpt up to the beginning of the lemma 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:

```

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

```

Note that the syntax to start a `Lemma` lets one name not only parameters such as `m1` and `m2`, but also assumptions such as `co_m12`.

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:

```

1 Lemma leqnn n : n <= n. Proof. Admitted.
2
3 Lemma example a b : a + b <= a + b.
4 Proof. by apply: leqnn. Qed.

```

The comparison performed by the `apply` tactic is up to computation:

```

1 Lemma example a b : a.+1 + b <= (a + b).+1.
2 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:

```

1 (* This line belongs to the file where leqnn is stated and proved. *)
2 Hint Resolve leqnn.
3 Lemma example a b : a + b <= a + b.

```



```
4 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 4.2.1. 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:

```
1 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:

```
1 Lemma example m p : prime p ->
2   p %| m `! + 1 -> m < p.
3 Proof.
4   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

```
1 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  $m$ :

```
1 Lemma example m p : prime p ->
2   p %| m `! + 1 -> m < p.
3 Proof.
4   move=> prime_p.
5   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  $(b \rightarrow c)$  of the statement of the lemma `contraLR`. The new statement of the goal is the corresponding instance of the premise  $(\sim\sim c \rightarrow \sim\sim b)$  of lemma `contraLR`.

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

```

1 Lemma example m p : prime p ->
2   p %| m `! + 1 -> m < p.
3 Proof.
4 move=> prime_p.
5 apply: contraLR.
6 rewrite -leqNgt.
7 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:

```

1 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  $\sim\sim (p \%| 1)$  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  $p \%| m`!$ .

```

1 Lemma example m p : prime p ->
2   p %| m `! + 1 -> m < p.
3 Proof.
4 move=> prime_p.
5 apply: contraLR.
6 rewrite -leqNgt.
7 move=> leq_p_m.
8 rewrite dvdn_addr.

```

```

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  $p \leq m$  and because we can combine the following lemmas:

```

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

```

Notice that the expression  $0 < m \leq n$  in `dvdn_fact` is really an abbreviation for  $(0 < m) \ \&\& \ (m \leq n)$ .

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

```

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

```

Finally, the resulting script would be:

```

1 Lemma example m p : prime p -> p %| m `! + 1 -> m < p.
2 Proof.
3 move=> prime_p.
4 apply: contraLR.
5 rewrite -leqNgt.
6 move=> leq_p_m.
7 rewrite dvdn_addr.
8   rewrite gtnNdvd.
9     by []. (* ~` false *)
10    by []. (* 0 < 1 *)

```

```

11   by apply: prime_gt1. (* 1 < p *)
12   apply: dvdn_fact.
13   rewrite prime_gt0. (* 0 < p <= n *)
14   by []. (* true && p <= m *)
15   by []. (* prime p *)
16   Qed.

```

For brevity, we record the goal solved by a tactic in a comment after this tactic. Before improving this script a comment is due: the goal after line 12,  $0 < p \leq m$ , is really an abbreviation for  $(0 < p) \ \&\& \ (p \leq m)$ . The subsequent `rewrite` command replaces  $(0 < p)$  with `true`: after all the conclusion of `prime_gt0` is an equation. We explain such tricks in detail later on.


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 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.<sup>2</sup> The proof script (up to the end of the proof of the first goal) then looks like this:

```

1 Lemma example m p : prime p -> p %| m `! + 1 -> m < p.
2 Proof.
3   move=> prime_p; apply: contraLR; rewrite -leqNgt; move=> leq_p_m.
4   rewrite dvdn_addr.
5     rewrite gtnNdvd //.
6   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.

-  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.,  $(\text{true} = \text{true})$ ), and the side condition  $(\text{prime } p)$ . Both are trivial, hence solved by prefixing the line with `by`.

```

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

```

The same considerations hold for the last goal.

```

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

```

<sup>2</sup>From this example, one might take away the wrong impression that a semicolon is synonymous to a dot. In general, it is not, since the tactic following it is applied to each goal resulting from the tactic preceding it. More details can be found in [Coq, “The tactic language”].

```

5   by rewrite gtnNdvd // prime_gt1.
6   by rewrite dvdn_fact // prime_gt0.
7   Qed.

```

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 4.1), the `move` tactic does nothing; it is the `=>` that is responsible for naming 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 for 0 and if, for each natural number  $n$ , the property  $\mathcal{P}$  holds for  $n+1$  as soon as it holds for  $n$ , 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:

```

1   About nat_ind.
2
3

```

```

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:

```

1   (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

```

1   (fun n => (odd n && prime n) = true) 0

```

```

1   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.

For 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:

```
1 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`.

```
1 Lemma addn0 m : m + 0 = m.
2 Proof.
3 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). This proof can be concluded 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: a variable whose value changes during recursive calls; hence the induction hypothesis must be general. We show this feature later on a concrete example (lemma `foldl_rev`).

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:

```
1 Lemma last_ind A (P : list A -> Prop) :
2   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:

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

```

3
4 Lemma foldl_rev T R f (z : R) (s : seq T) :
5   foldl f z (rev s) = foldr (fun x z => f z x) z s .

```

The proof uses the following lemmas:

```

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

```

The complete proof script follows:

```

1 Lemma foldl_rev T A f (z : A) (s : seq T) :
2   foldl f z (rev s) = foldr (fun x z => f z x) z s .
3 Proof.
4 elim/last_ind: s z => [|s x IHs] z //.
5 by rewrite -cats1 foldr_cat -IHs cats1 rev_rcons.
6 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`:

```

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

```

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 would have 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 ++ [z])) =
2   foldr (fun y w => f w y) (foldr (fun y w => f w y) z [z]) 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 [z])`. 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. 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 [GMT15].

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, recall our proof of `example m p`:

```

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

```

Here, our use of the `rewrite dvdn_addr` tactic forced us to prove the side condition  $p \%| m`!$ . Side conditions (by default) become the second, third (and potentially higher) sub-goals 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.

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 does not mention the factorial operator. The same holds for `prime_gt0`. The resulting proof script is:

```

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

```

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.

```

1 Lemma silly_example n : n + 0 = (n + 0) + 0.
2 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.

```

1 Lemma simplify_me : size [:: true] = 1.
2 Proof.
3 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 5.4 we propose a less risky alternative. The sequence `// /=` can be collapsed into `//=`.

Another form of simplification that is often needed is the unfolding of a definition. For example, the lemma `leqE` that we used in the proof of `leq_mu121` back in page 52 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`. Such operation can be performed by prefixing the name of the object with `/`, as in `rewrite /leq`. Unfolding a definition is not a deductive operation but an instance of computation, as made more precise in chapter 3.

### 2.4.1 Rewrite contextual patterns

The example `leq_mul21` 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.

```
1 Lemma leq_mul21 m n1 n2 :
2   (m * n1 <= m * n2) =
3   (m == 0) || (n1 <= n2).
4 Proof.
5 rewrite [n1 <= _]leqE.
```

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

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]`:

```
1 Lemma leq_mul21 m n1 n2 :
2   (m * n1 <= m * n2) =
3   (m == 0) || (n1 <= n2).
4 Proof.
5 rewrite [in RHS]leqE.
```

```
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 [GMT15, 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:



```

1 Lemma leq_mul2l m n1 n2 :
2   (m * n1 <= m * n2) =
3   (m == 0) || (n1 <= n2).
4 Proof.
5 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<sup>3</sup> of the pattern: in this case it hence looks for an occurrence of  $(\_ \_ - \_ \_)$ , which is not found. As a consequence, we need 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:

```

1 Lemma leq_mul2l m n1 n2 :
2   (m * n1 <= m * n2) =
3   (m == 0) || (n1 <= n2).
4 Proof.
5 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:

```

1 Lemma leq_mul2l m n1 n2 :
2   (m * n1 <= m * n2) =
3   (m == 0) || (n1 <= n2).
4 Proof.
5 rewrite [in LHS]/leq.

```

```

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.

```

1 Lemma leq_mul2l_rewritten m n1 n2 :
2   (m * n1 - m * n2 == 0) =
3   (m == 0) || (n1 <= n2).
4 Proof.
5 rewrite -(leq _ _).

```

```

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

```

Here, we have used the `-/` prefix for the `rewrite` tactic; it allows *folding* a definition, i.e., the reverse of unfolding.

More generally, the `rewrite` tactic can be used to replace a certain subterm of the goal

<sup>3</sup>The head symbol is the root of the syntax tree of an expression.

by another one, which is equal to the former modulo computation:

```

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

```

```

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:

```

1 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):

```

1 dvdn2  ∀ n : nat, (2 %| n) = ~~ odd n
2 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`, and `addr` denote nat, boolean, and ring addition, respectively.
  - 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.<sup>4</sup>: 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 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 `Cx` for trailing complement)
    - \* D : set difference, addition
    - \* E : definition elimination (often conversion lemmas)

---

<sup>4</sup>These examples are taken from libraries in the Mathematical Components distribution.

- \* 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



## 3. Dependent type theory

The formal language we use to write mathematical statements and proofs in the COQ proof assistant is called *Gallina*. This language is an evolution of the *Calculus of Inductive Constructions* (CIC) [CH88; CP90] implemented in the early versions of COQ in the 80's. In turn, CIC is one of the many descendants of the intuitionistic type theory [Mar80] Martin-Löf developed in the 70's. In order to avoid ambiguities with other type theories we refer to this family of formal systems using the term *dependent type theory*.

In this chapter, we provide some hints on the main features of this formalism. The interested reader shall refer to the reference manual of COQ [Coq] for a formal definition of Gallina.

### 3.1 Propositions as types, proofs as programs

Set theory is commonly invoked as the foundational language for mathematics [Bou04]. 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, for example the ones of Zermelo-Fraenkel set theory. By contrast, proof assistants based on a flavor of dependent type theory, like COQ, embrace an approach coined *propositions-as-types* [How80], and use the same language of types in a uniform way to describe mathematical objects, mathematical assertions, and their proofs. As we shall see the rules dictating which sentences and proofs are well-formed are phrased quite differently in the two formal languages.

In set theory the rules which govern the construction of well-formed, grammatically correct, statements are rather loose. For example  $x \in x$  is a valid sentence, where  $x$  plays the role of both an element and as set. Still, the goal of the game is to construct a proof for a given well-formed proposition, using first-order logic to combine the axioms of the theory, and nonsensical sentences supposedly have no proof.

The first two chapters of the present book illustrate how types can be used to help classify, and clarify expressions passed to the checker. In fact, the language of types available in the Calculus of Inductive Constructions, and thus in COQ, is so expressive that

*logical statements* are identified with some *types* and their *proofs* with *terms*, or programs, having this type. This way, proving a statement consists in fact in constructing a term of the corresponding type. Objects of the formalism are programs, and proofs are themselves objects of the formalism.

In this setting the analogue of “a proposition has a proof” is that “a term has a given type”. Such a statement, called a *typing judgment*, is a ternary relation written as follows:

$$\Gamma \vdash t : T$$

and reads: *in the context*  $\Gamma$ , *the term*  $t$  *has the type*  $T$ . A type is just a term, which is called a type when it occurs on the right hand-side of a column in a well-formed typing judgment, like  $T$  in our case. A context is a list of variables, each paired with a type, that can occur in  $t$  and  $T$ . This typing judgment expresses that under the typing assumptions listed in the context  $\Gamma$ ,  $t$  and  $T$  are well-formed, and moreover that term  $t$  has type  $T$ . Example:

$$x : \mathbb{N} \vdash x + x : \mathbb{N}$$

A typing judgment is valid if it is justified by combining the *typing rules* of the type theory, up to atomic ones like this one:

$$x : T \vdash x : T$$

which asserts that a context assigns a type to each variable it contains. A call to the `Check` command introduced in Chapter 1 verifies that a certain typing judgment holds in the current context. For instance, term `3` has type `nat` in an empty context:

```
1 Check 3 : nat.
```

```
3 : nat
```

The context of a typing judgment includes all the variables and hypothesis currently assumed. For instance, in a context containing a variable `n` with type `nat`, the term `n + n` has type `nat`:

```
1 Variable n : nat.
2 Check n + n : nat.
```

```
n + n : nat
```

COQ complains if we try to verify the typing judgment asserting the same term has type `bool`. In this case, it even computes and displays the correct type:

```
1 Fail Check n + n : bool.
2
```

```
The term "n + n" has type "nat" while it
is expected to have type "bool".
```

Our last example is a typing judgment  $\Gamma \vdash t : T$  where the term  $t$  is a proof. Indeed, in a context containing a variable `n` of type `nat`, the term `(muln0 n)` has type `n * 0 = 0` and is thus a proof of the corresponding equational assertion:

```
1 Variable n : nat.
2 Check muln0 n : n * 0 = 0.
```

```
muln0 n : n * 0 = 0
```

### 3.2 Terms, types, sorts

This section makes more precise what contexts, terms and types are. Note that, for the sake of the exposition, the description is restricted to a subset of Gallina. We refer again the reader looking for an exhaustive description to the corresponding chapter of Coq's reference manual [Coq].

As alluded to in the introduction, type theory avoids the distinction set theory makes between sets and propositions: type theory is based on a same and single collection of inductively defined terms. A judgment  $\Gamma \vdash t : T$  relates two *terms*,  $t$  and  $T$  (in the context  $\Gamma$ ), and  $T$  is called a *type* because it appears on the right hand side of the colon symbol in a typing judgment. Term  $T$  can be thought of as a label for a collection of terms, and the judgment  $\Gamma \vdash t : T$ , as the statement that (in context  $\Gamma$ ) “term  $t$  belongs to the collection  $T$ ”, or even, to some extent,  $t \in T$ . For instance, we have used in previous examples assumptions of the form  $n : \text{nat}$  to model the sentence “let  $n$  be a natural number”.

This set-theoretic analogy should be taken with a pinch of salt though. First, there is no way in type theory to introduce a term, even a variable, without simultaneously introducing its type. Things are different in set theory, where an object  $a$  can be constructed without necessarily being casted as being the element of a super-set  $A$ . Second, a judgment is not a proposition in the same sense as the set-theoretic sentence  $t \in T$  would be. In particular, one cannot reason (internally) by case analysis on the proof that  $t$  has type  $T$ , nor can we disprove (internally) that  $t$  has type  $T$  for some particular  $T$ . Finally, as we shall see later in this section, substitution of equals does not behave the same for terms at the left of a colon, and for those on the right — types.

We assume a collection of distinct names, used to denote atomic terms and called *sorts*. One of this sorts is called `Prop`, and it is the type of statements:

```
1 Check 7 = 7 : Prop.
```

```
7 = 7 : Prop
```

As noted already in Chapter 2, a well formed statement is not necessary a provable one:

```
1 Check 7 = 9 : Prop.
```

```
7 = 9 : Prop
```

Types used as data-structures, to represent mathematical objects (as opposed to mathematical assertions), live in a distinct sort named `Set`:

```
1 Check nat : Set.
```

```
nat : Set
```

Gallina moreover features a countable, cumulative hierarchy of sorts, displayed as `Type` and indexed by an integer variable which is by default hidden to the user. In the following typing judgment we tell Coq to print the index of `Type`:

```
1 Set Printing Universes.
2 Check Type : Type.
```

```
Type@{U1} : Type@{U2} (* U1 < U2 *)
```

the instance of `Type` on the right side of the colon has a greater index (named `U2` here) than the instance of `Type` on the left.

In an empty context, the term `Prop` has type `Type` (for any value of the index) :

```
1 Check Prop : Type.
```

```
Prop : Type@{U3} (* Prop < U3 *)
```

`Set` happens to be a name for the smallest element of the hierarchy of `Type`:

```
1 Check Set : Type.
```

```
Set : Type@{U4} (* Set < U3 *)
```

Terms `nat`, `bool`, `Prop` are simple, atomic types, which can be used to build non-atomic ones. For instance, in Chapter 2, we met the type of functions from natural numbers to boolean, which is `nat -> bool`. Similarly the type of addition over natural numbers is `nat -> nat -> nat`, and the type of boolean negation is `bool -> bool`.

In Chapter 1, we used a *polymorphic* type to represent sequences of elements:

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

Since `seq` builds a new type for each given instance of its parameter, it has the type of a function:

```
1 Check seq : Type -> Type.
```

Non-atomic terms are defined from a countable infinite set of symbols, called *variables*. It is thus always possible to exhibit a new variable distinct from a given arbitrary finite collection of variables. In COQ syntax, we can use letters, or more generally sequences of alpha-numeric characters, to represent variables. For instance, the following command declares two variables of type `nat`, i.e. two natural numbers and one of type `seq nat`, i.e. a list of numbers:

```
1 Variables (n m : nat) (l : seq nat).
```

Hypotheses are themselves variables, whose type represent the assumed statement. The following line assumes a hypothesis:

```
1 Hypothesis neq0 : n = 0.
```

But it is just a synonym, with a more suggestive name, of:

```
1 Variable neq0 : n = 0.
```

Of course, it is only possible to extend the current context with variables, or hypotheses, or definitions, with a well-formed type. As we have seen so far, well-formed types include atomic types, like `nat` or `Prop`, and function types like `nat -> bool`. In Gallina, it is also possible to define functions which build a *type* for any *value* of a given datatype:

```
1 Variable t : nat -> Type.
```

Term `t` is called a *dependent type*, as it constructs a family of types which depend on a parameter in a data type. For instance, `t` could be the type of lists of booleans with a prescribed length, `t n` being the type of lists of length `n`. Now such a dependent type can itself be used as the return type of a function, like in:



```
1 Variable g : ∀ n : nat, t n.
```

For instance, `g n` could build the list of length `n` containing only `true` elements. Thus in Gallina, the return type of a function may depend on the *value* of its argument: the prefix `∀ n`, part of the type of `g` is a binder, which allows to describe the dependency in the return type. A type of the form `∀ x, τ` is called a *product type*, or sometimes a  $\Pi$ -type. Note that a product type `∀ x, τ` is well-formed for any `τ` of type `Type`, `Set` or `Prop`, even when `τ` does not depend on `x`. In this case, when the return type is constant in the argument, the product type is displayed with an arrow, as in:

```
1 Check ∀ x : nat, bool : Type.
```

```
nat -> bool : Type
```

A term with a product (or arrow) type is a function, in the sense that it can be *applied* to an argument, provided that the type of this argument agrees with the source of the function type:

```
1 Variable f : nat -> bool.
2 Check f 3 : bool.
3 Fail Check f true : bool.
4
5 Variable g : ∀ n, t n.
6 Check g 3 : t 3.
```

The typing rule governing the application of functions to arguments is a emblematic example of the strict well-formedness conditions enforced by types on mathematical statements. This rule will rule out nonsensical assertions, like “ $\pi$  is equilateral” or “2 is a Banach space”. But the same rule shall also play a nastier role, for instance if the user wants to casually embed natural numbers into integers, or and integer into rational numbers: working with these obvious inclusions might require an explicit cast in a typed setting, if  $\mathbb{N}$ ,  $\mathbb{Z}$ ,  $\mathbb{Q}$  are represented by distinct types.

New functions are defined by *abstracting* a variable in a term, as discussed in Chapter 1. The resulting term has a product (or an arrow) type:

```
1 Definition bar (n : nat) : bool :=
2   n == 0.
3 Check bar : nat -> bool.
```

Term `bar` has type `nat -> bool` because in the current context, augmented with a variable declaration `n : nat`, corresponding to the name and type of the argument, the body `n == 0` of the definition has type `bool`.

As we have seen in Chapter 1, a function is defined by *binding* its argument in the body of its definition: in `bar`, the variable `n`, on the left of the `:=` delimiter, is bound in the body of the definition, the `n` on the right of the `:=` delimiter. When a defined function is applied to its argument, the resulting term is *computed* by substituting the bound variable for the value of the argument.

```
1 Definition bar (n : nat) : nat := n.
2 Print bar.
3 Compute bar 4.
```

```
bar = fun n => n : nat -> nat
= 4 : nat
```

### 3.3 Propositions, implication, universal quantification

The sort `Prop` is the type of propositions. If `A` and `B` are two types in sort `Prop`, then `A → B` is also a type, living as well in sort `Prop`. A term of type `A → B` is a function, that transforms any term of type `A` into a term of type `B`, and that can be applied to any term of type `A`. This is the central typing rule of our formal system. It is a deep remark that the typing rules for function formation and function application can be read as the introduction rule and elimination rule of logical implication, respectively. For instance, let us analyse a proof that modus ponens holds:

```
1 Lemma modus_ponens (A B : Prop) : (A → B) → A → B.
2 Proof.
3 move=> hAB hA.
4 apply: hAB.
5 exact: hA.
6 Qed.
```

This proof starts by introducing the two hypotheses, akin to the two arguments of a function. Then it builds a proof of `B` which is the result of the function `hAB` applied to the argument `hA`. A variant is:

```
1 Lemma modus_ponens (A B : Prop) : (A → B) → A → B.
2 Proof.
3 move=> hAB hA.
4 exact: (hAB hA).
5 Qed.
```

A last variant, providing directly the proof, without interactive commands:

```
1 Lemma modus_ponens (A B : Prop) : (A → B) → A → B.
2 Proof.
3 exact: (fun hAB hA => hAB hA).
4 Qed.
```

The remark extends to universal quantification, whose introduction and elimination rules also coincide with the formation and application rules of a product type. Just like the constructive proof of an implication is a function transforming an arbitrary proof of the premise into a proof of the conclusion, a constructive proof of a universal statement is a family of proofs, indexed by the inhabitants of the type over which quantification takes place. In this view, a proof of the Pythagorean theorem is a family of proofs indexed by the collection of rectangular triangles.

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  $(\forall y, 3 + y = y + 3)$ . 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. Functions whose codomain type depend on the value of their input have a type of the form  $\forall x : A, B$ , where  $x$  can occur in  $B$ . When  $B$  does not depend on  $x$  this type is written  $A \rightarrow B$ , avoiding the useless introduction of a name  $x$ . The former is sometimes called the *dependent function space* ( $\forall$ ) and the latter, the standard function space ( $\rightarrow$ ).

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

Yet providing all the arguments of a given lemma, so as to describe the precise instance useful in a proof, can be tedious. It is one of the duties of the language used in the interactive construction of proofs to leverage this bureaucracy. For instance, the tactics `apply` and `rewrite` introduced in Chapter 2 are designed to guess some of these arguments. This guess is based on matching and unification with the current goal, or with a pattern provided in argument.

We refer the reader to the reference manual of COQ [Coq] 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`.

### 3.4 Conversion

The *conversion (typing) rule* in Gallina describes the status of computation in this dependent type theory, and plays a fundamental role in the formalization choices adopted in the Mathematical Component libraries. Computation is modeled by rewrite rules explaining how to apply functions to their argument. For instance, the so-called  $\beta$ -reduction rule rewrites the application of a function to an argument `(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 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*, written  $t_1 \equiv t_2$ , and these terms are indistinguishable to the type system.

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)`<sup>1</sup> is a valid proof of `(0 < n.+1)`, as well as a proof of `(0 != p.+1)`.

It is out of scope for this book to detail other uses of conversion. We just mention that while the Mathematical Components library takes advantage of conversion to deal with “little” computations like the ones above, the Coq system efficiently supports “large” computations as well. One idiomatic example is found in the formal proof of the Four-Color theorem [Gon08], another one in the implementation of the proof command `ring` that decides equalities in the homonymous algebraic structure [GM05].

### 3.5 Inductive types

Stricto sensu, in the formalism described in section 3.2, types are either sorts, like `Prop` and `Type`, or functional types constructed from these atomic ones. However, in the previous chapters, we have casually used other types like `nat` or `bool` and terms of these types like `0 : nat` or `S : nat -> nat`. In fact, in the Calculus of Inductive Construction it is also possible to introduce new types – and in fact new terms – via *inductive definitions* [CP90;

<sup>1</sup>In other words `isT` is a proof of all statements that are trivial by computation. We invite the reader that finds the writing `(isT : true)` ill typed to peek ahead to section 5.5.

[Pau93]. We only provide a very brief overview of this subtle feature and again refer the reader to the reference manual for a precise and formal account.

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 : Set := 0 : nat | S (n : nat).
```

introduces a new type `nat : Set` and two new terms (`0 : nat`) and (`S : nat -> nat`). Constructors of a type  $\tau$  are function symbols, possibly of zero arguments like `0`, and the codomain of their type is always  $\tau$ . The inductive type  $\tau$  may occur in the type of certain arguments of its constructors, like `nat` in the type of `S`. The only way to construct an inhabitant of an inductive type is to apply a constructor to sufficiently many arguments:

```
1 Unset Printing Notations.
2 Variable n : nat.
3
4 Check 0.
5 Check S 0.
6 Check S (S (S 0)).
7 Check S (S (S n)).
```

Note that for the sake of clarity, we do not make use in this section of the postfix notations `n.+1` for term `(S n)`, `n.+2` for term `(S (S n))`, etc. so as to display constructors explicitly in examples.

Constructors are by definition injective functions: two terms featuring the same head constructor can only be equal if the arguments passed to this constructor are equal. E.g., in the following goal:

```
n, m : nat
eqSnSm : S n = S m
=====
G
```

where `G` is an arbitrary formula, it is possible to simplify hypothesis `eqSnSm`:

```
1 case: eqSnSm.
2
3
4
5
```

```
n, m : nat
=====
n = m -> G
```

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 non-zero test function by:

```
1 Definition non_zero n := if n is (S p) then true else false.
```

where we recall from Chapter 1 that `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:

```

1 Fixpoint addn n m :=
2   match n with
3     | 0 => m
4     | S p => S (addn p m)
5   end.

```

These fixpoints are said to be guarded because the corresponding function should always terminate: more precisely, the `Fixpoint` command expects termination to follow from a syntactic criterion. For instance, in the case of `addn`, the recursive call happens on a strict subterm of the argument. Allowing non-terminating computations for well-typed terms would actually interact badly with the conversion rule, and ultimately lead to proofs of absurdity (see section 3.7). When the termination argument of a function falls outside this syntactic guard condition, its definition usually involves an extra argument, witnessing the decreasing order relation. For a more detailed exposition of these techniques, see for instance the corresponding chapter in Bertot and Casteran’s book [BC04, Chapter 14].

### 3.6 More connectives

We have seen in section 3.3 that functions, i.e. terms with a product type, provide a datastructure for proofs of implication and universally quantified statements. Using inductive types, it is possible to describe more data structures than mere functions. These data structures and their typing rules are used to model every other logical connectives.

For instance, the introduction rule of the conjunction connective reads: to prove  $A \wedge B$  one needs to prove both  $A$  and  $B$ . Conversely, the elimination rule 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, which provides a type for pairs of proofs, of the parameter statements  $A$  and  $B$ :

```

1 Inductive and (A B : Prop) : Prop := conj (pa : A) (pb : B).
2 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 models faithfully the introduction rule of conjunction, i.e. the rule governing the construction of proofs of conjunctive statements.

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

```

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

```

Pattern matching provides the elimination rule for conjunction, i.e. the (two) rules

governing the construction of proofs using a conjunctive hypothesis. Here is left elimination rule:

```
1 Definition proj1 A B (p : A /\ B) : A :=
2   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:

```
1 About conj.
```

```
conj:  $\forall A B : \text{Prop}, A \rightarrow B \rightarrow A \wedge B$ 
```

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.

```
1 Inductive ex (A : Type) (P : A -> Prop) : Prop :=
2   ex_intro (x : A) (p : P x).
3 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`:

```
1 ex :  $\forall A : \text{Type}, (A \rightarrow \text{Prop}) \rightarrow \text{Prop}$ .
2 ex_intro :  $\forall A : \text{Type}, \forall P : A \rightarrow \text{Prop}, \forall a : A, P a \rightarrow \text{ex } A P$ .
```

Both `ex` and `ex_intro` are parameterized by `(A : Type)`: as we have seen in chapter 1, the `( $\forall A : \text{Type}$ )` quantification indicates that `ex` and `ex_intro` are polymorphic constants, that can be instantiated for any type. Both `ex` and `ex_intro` also have a parameter of type `(A -> Prop)`: since it does not appear in the rest of the type of `ex`, this parameter is not named and the type uses the arrow syntax instead of a more verbose  `$\forall A : \text{Type}, \forall P : A \rightarrow \text{Prop}, \text{Prop}$` . This parameter is indicated by a so-called *higher-order* quantification, because the parameter has an arrow type. Constants `ex` and `ex_intro` can thus be specialized to any predicate `P`, so that the `ex` inductive declaration can be used on any formula. Finally, the `ex_intro` constant features a last, inner-most  `$\forall a : A$`  quantifier, which binds a term variable `a` representing the witness of the existential statement for `P`.

We now look at the inductive definition of the disjunction `or` and its two constructors `or_intro1` and `or_intror`.

```

1 Inductive or (A B : Prop) : Prop := or_introl (a : A) | or_intror (b : B).
2 Notation "A \\/ B" := (or A B).

```

The elimination rule can again be expressed by pattern matching:

```

1 Definition or_ind (A B P : Prop)
2   (aob : A \\/ B) (pa : A -> P) (pb : B -> P) : P :=
3   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  $\top$ ,  $\perp$  and  $\neg$  can be defined as follows.

```

1 Inductive True : Prop := I.
2 Inductive False : Prop := .
3 Definition not (A : Prop) := A -> False.
4 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:

```

1 Definition exfalse (P : Prop) (f : False) : P :=
2   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.

```

1 Inductive eq (A:Type) (x:A) : A -> Prop := erefl : eq A x x.
2 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 [Uni13].

### 3.7 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 propositions-as-types 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 through the looking glasses of the propositions-as-types correspondence.

```

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  $(\forall n : \text{nat}, P\ n \rightarrow P\ n.+1)$ . Let us write such a program by hand.

```

1 Fixpoint nat_ind (P : nat -> Prop)
2   (p0 : P 0) (pS : ∀ n : nat, P n -> P n.+1) n : P n :=
3   if n is m.+1 then
4     let pm (* : P m *) := nat_ind P p0 pS m in
5     pS m pm (* : P m.+1 *)
6   else p0.

```

The COQ system generates this program automatically, as soon as the `nat` data type is defined.

It is worth mentioning that this principle is general enough to prove the “stronger” induction principle that give access, in the inductive step, to the property  $P$  on all numbers small than  $n.+1$  and not just its predecessor.

```

1 Lemma strong_nat_ind (P : nat -> Prop)
2   (base : P 0)
3   (step : ∀ n, (∀ m, m <= n -> P m) -> P n.+1) x : P x.

```

In order to prove this statement it is sufficient to craft the right “P” for  $n$ .

```

1 Check (nat_ind (fun n => ∀ m, m <= n -> P m)).

```



```
( $\forall m, m \leq 0 \rightarrow P m$ )  $\rightarrow$ 
( $\forall n, (\forall m, m \leq n \rightarrow P m) \rightarrow \forall m, m \leq n.+1 \rightarrow P m$ )  $\rightarrow$ 
 $\forall n m, m \leq n \rightarrow P m$ 
```

If we fulfill the last premise with `(leqnn x : x <= x)` we obtain the following proof goals:

```
1 Proof.
2 apply: (nat_ind (fun n =>  $\forall m, m \leq n \rightarrow P m$ ) _ _ x x (leqnn x)).
```

```
P : nat -> Prop
base : P 0
step :  $\forall n : nat, (\forall m : nat, m \leq n \rightarrow P m) \rightarrow P n.+1$ 
x : nat
=====
 $\forall m : nat, m \leq 0 \rightarrow P m$ 

subgoal 2 is:
 $\forall n : nat,$ 
( $\forall m : nat, m \leq n \rightarrow P m$ )  $\rightarrow \forall m : nat, m \leq n.+1 \rightarrow P m$ 
```

The two goals follow from `base` and `step` respectively. Induction principles like this one can be used by giving their name to `elim` as in `elim/strong_nat_ind`. Still, strong induction is such a frequent proof step that the Mathematical Components library provides dedicated idioms that we will detail in Section 5.3.

Finally, 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.

```
1 Fixpoint oops (n : nat) : False := oops n.
2 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.

```
1 Inductive hidden := Hide (f : hidden -> False).
2 Definition oops (hf : hidden) : False := let: Hide f := hf in f hf.
3 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 is called *positivity condition* and, 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 [Coq].





## 4. A proof language for formal proofs

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 [NGV94].

Still, the use of a dedicated proof language enables a higher level description of the formal proof being constructed. The Small Scale Reflection proof language give us tools to structure proofs and to tame bookkeeping, a form of bureaucracy typical of formal proofs.

Structure is given by splitting large proofs in blocks with a specific, declared, purpose and by factoring repetition such as symmetric or less general cases. The main tool logic gives us is the cut rule where the author of a proof identifies an intermediate fact, or a generalization of the goal. The `have` and `without loss` tactics precisely cover this need.

Bookkeeping is the ubiquitous activity of context management, that is naming or discarding assumptions in order to track their lifetime to tide the context up, as well as massaging assumptions and goals as to please the formality requirements of Coq. A paradigmatic bookkeeping example is destructing assumption of  $(A \ \&\& \ B)$  in order to give distinct names to  $A$  and  $B$ : unlikely to be a pregnant step of the proof. The language of intro-patterns together with then goal-as-stack model provides a very flexible tool to succinctly deal with bookkeeping.

Structure and manageable size are key to the maintenance of proofs by a team of people. Being small in size helps maintenance since when computer code does not fit the screen one has hard times understanding it [Wei85] and hence repairing it. Moreover, structure confines errors arising after a change to smaller text blocks, and declares what the original author of the block being repaired was doing effectively implementing a form of check pointing and documentation. All that works best if the tactics used to fill in the blocks have a predictable behavior: fail early and locally when a breaking change takes place.

This chapter covers the tools to structure and tie proofs up and discusses some of the good practices that made the development of the Mathematical Components library possible.

## 4.1 Bookkeeping: goals as stacks

The presentation we gave so far of proof commands like `case: n => [!m]` is oversimplified. 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, breaking a pair into two pieces is usually not a significant, meaningful step in a proof: hence the possibility to use a lightweight and compact syntax for this bookkeeping action, instead of an explicit mention of the `case` tactic.

### 4.1.1 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 -> ...`, or on `x` if the goal has the form `∀ 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:

```
1 move=> xy pr_x odd_y.
2
3
4
5
```

```
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.

```
1 move=> [x y] pr_x odd_y.
2
3
4
5
```

```
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 tell COQ to reduce the formulas on the stack, before introducing them in the context, and obtain:

```

1 move=> [x y] /= pr_x odd_y.
2
3
4
5

```

```

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*. The `prime_gt1` lemma states that  $(\text{prime } p \rightarrow 1 < p)$  for any  $p$ , and we can use it as a function to obtain a proof of  $(1 < x)$  from a proof of  $(\text{prime } x)$ .

```

1 move=> [x y] /= /prime_gt1-x_gt1 odd_y.
2
3
4
5

```

```

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.

```

1 move=> [x [//|z]] /= /prime_gt1-x_gt1.
2
3
4
5

```

```

x, z : nat
x_gt1 : 1 < x
=====
~~ odd z -> 2 < z.+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 `z` the new variable (the predecessor of what used to be called `y`). Since `y` is destroyed we could have reused that name for its predecessor, as it is often done in the Mathematical Components library. In fact, the boolean predicate `odd` is defined by case analysis, and induction, on its argument of type `nat`:

```

1 Fixpoint odd n := if n is n'.+1 then ~~ odd n' else false.

```

Therefore, when applied to `z.+1`, it simplifies to `~~ odd z`.

Now, the fact that `z` is even is not needed to conclude, so we can discard it by giving it the `_` dummy name.<sup>1</sup>

```

1 by move=> [x [//|z]] /= /prime_gt1-x_gt1 _; apply: ltn_add1 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_add1` looks as follows:

<sup>1</sup>If an assumption has already a name, it can be discarded by writing its name in curly braces: e.g. `move=> {x_gt1}`.

```
1 About ltn_add1.
2
3
```

```
ltn_add1 : ∀ m n p : nat,
  m < n -> m < p + n
Arguments m, n are implicit
```

`apply`: automatically fills in the blanks between the function `ltn_add1` (the lemma name) and the given argument `x_gt1`. Since we are passing `x_gt1`, the variable `m` takes the value `1`. The conclusion of `ltn_add1` hence unifies with  $(2 < z.+1 + x)$  because both `+` and `<` are defined as programs that compute: Namely, addition exposes a `.+1` by reducing to  $2 < (z + x).+1$ ; then `<` (or, better, the underlying `<=`) eats a successor from both sides, leading to  $1 < z + x$ , 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. As a rule of thumb: follow the granularity of how you would naturally read your script to someone else.

! 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.

### 4.1.2 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.
2
3
```

```
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.
2
3
```

```
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).
2
```

```
=====
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,  $\text{top}$ ), by writing  $/v$ , we are forming the term  $(v \text{ top})$ , whereas when we specialize the top stack item  $\text{top}$  to an object  $x$ , by writing  $/(_ x)$ , we are forming the term  $(\text{top } x)$ . The application of a view written  $/v$  is short for  $/(v \_)$ .

When the top stack item is an equation, one can substitute it into the rest of the goal, using  $<-$  and  $>$  for right-to-left and left-to-right respectively.

```
1  move=> /(_ 3) <-.
2
```

```
=====
6 = 3 * 2
```

In other words, the arrows are just a compact syntax for rewriting, as in the `rewrite` tactic, with the top assumption.

### 4.1.3 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.

! The colon symbol in `apply:` is not the discharging operator. It is just a marker to distinguish the `apply:` tactic of Small Scale Reflection from the `apply` tactic of COQ. Indeed the two tactics, while playing similar roles, behave differently [GMT15, §5.2 and §5.3].

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, `:` 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
```

Note that listing context entry names inside curly braces purges them from the context. For instance the tactic `case: y {odd_y}` clears the `odd_y` fact. But this would lead to a dead end in the present proof, so we don't use it here.

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
3
4
5
6
7
```

```
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: (leqmn 7)`” pushes on the stack the additional assumption  $(7 \leq 7)$ .

## 4.2 Structuring proofs, by examples

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, factoring similar lines of reasoning (e.g., symmetries), signaling crucial steps to the reader, and so on. In short, a proof written in COQ should share its structure and main steps with the same 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.

### 4.2.1 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.  $\square$

We state our theorem 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.

```
1 Inductive ex2 A P Q : Prop := ex_intro2 x of P x & Q x.
2 Notation "exists2 x , p & q" := (ex2 (fun x => p) (fun x => q)).
```

We also resort to the following notations and lemmas.

```
1 Notation "n ^!" := (factorial n).
2 Lemma fact_gt0 n : 0 < n^!.
3 Lemma dvdn_fact m n : 0 < m <= n -> m %| n^!.
4 Lemma pdivP n : 1 < n -> exists2 p, prime p & p %| n,
5 Lemma dvdn_addr m d n : d %| m -> (d %| m + n) = (d %| n).
6 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.

```
1 Lemma prime_above m : exists2 p, m < p & prime p.
2 Proof.
3 have m1_gt1: 1 < m^! + 1.
4 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.

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.

```

1 Lemma prime_above m : exists2 p, m < p & prime p.
2 Proof.
3 have m1_gt1: 1 < m`! + 1.
4   by rewrite addn1 ltnS fact_gt0.
5 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. In the light of that, the script can be rearranged as follows.

```

1 Lemma prime_above m : exists2 p, m < p & prime p.
2 Proof.
3 have /pdivP[p pr_p p_dv_m1]: 1 < m`! + 1.
4   by rewrite addn1 ltnS fact_gt0.
5 exists p => //; rewrite ltnNge; apply: contraL p_dv_m1 => p_le_m.
6 by rewrite dvdn_addr ?dvdn_fact ?prime_gt0 // gtnNdvd ?prime_gt1.
7 Qed.

```

Here the first line obtains a prime `p` as desired, the third one begins to show it fits by contrapositive reasoning, and the last 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).

### 4.2.2 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 (i.e. duplication). The language of COQ is sufficiently expressive to model symmetries, and the Small Scale Reflection 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.  $\square$

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.

```

1 Lemma orP {a b : bool} : a || b -> a ∨ b.
2 Lemma orb_idr (a b : bool) : (b -> a) -> (a || b) = a.
3 Lemma orbC a b : a || b = b || a.
4 Lemma maxn_idP1 {m n} : n <= m -> maxn m n = m.
5 Lemma maxnC m n : maxn m n = maxn n m.
6 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_idP1 le_n21) orb_idr // => le_mn2.
4   by apply: leq_trans le_mn2 le_n21.
5   rewrite maxnC orbC.
6   rewrite (maxn_idP1 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:

```

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_idP1`. Then `orb_idr` trivializes the main goal and generates a side condition with an extra hypothesis we name `le_mn2`.

```

2 subgoals
m, n1, n2 : nat
le_n21 : n2 <= n1
=====
(m <= n1) = (m <= n1) || (m <= n2)

subgoal 2 is: ...

```

```

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 `th_sym` is the text we were copy-pasting 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. If we stop just before `; last rewrite ...` these are the (symmetric) goals:

```

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)

```

The instance of `th_sym` is exactly what is needed in the first branch, while the last goal requires the commutativity of `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 undesirable, 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.

```

Thanks to `wlog` one only needs to write the statement of the extra assumption, here  $n2 \leq n1$ , and which portion of the context needs to be abstracted, here  $n1$  and  $n2$ . The two goals to be proved are the following ones:

```

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)

```

```

m, n1, n2 : nat
le_n21 : n2 <= n1
=====
(m <= maxn n1 n2) = (m <= n1) || (m <= n2)

```

To keep the script similar to the previous one, we named explicitly `th_sym`, but one typically leaves the assumption on the stack and omit the trailing intro pattern `=> [th_sym|]`.

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  $(n2 \leq n1)$  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 Small Scale Reflection proof language can be used to shrink the proof even further. In the previous proof script at line 5 we had to name `le_mn2` the new assumption only for the purpose of referring to it in the immediately following transitivity step. We can avoid this by using `leq_trans` as a view.

```

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.

```

The statement of `leq_trans` is  $(\forall c a b, a \leq c \rightarrow c \leq b \rightarrow a \leq b)$  and we use it to transform the top assumption  $(m \leq n2)$  that fits in the place of the first proof argument  $(a \leq c)$ . Note that the `leq_trans` expects a second proof argument, and that its type would fix  $b$ , that is otherwise unspecified. As a result  $b$  stays quantified. We can put a full stop just before the terminating `->` in order to observe the shape of the top assumption before rewriting it:

```
1 (∀ b, n2 <= b -> m <= b) -> m <= n1.
```

Rewriting the top assumption left to right fixes  $b$  to  $n1$ , trivializes the goal  $(m <= n1)$  to `true` and opens the side condition  $(n2 <= n1)$  which is trivial.

### 4.3 Proof maintenance: a matter of style

The Mathematical Components library started from a few files from the Four Color Theorem in 2006 and it was developed over 6 years by 10 people to support the formal proof of the Odd Order Theorem, completed in 2012. At the time of writing it has reached the size of nearly 100 files and 100 KLOC, and has been maintained for over 12 years.

During this maintenance period breakage did happen, and breakage will happen again in the future. Things break for many reasons, some of which are even not under the control of the author of a formal proof. First of all COQ is an actively developed research software and changes in behavior, for the better, are hard to avoid and do happen in practice. Second nobody gets his definitions perfect at the first attempt in a formalization. Definitions and lemmas get improved over time. As a consequence having to repair and adapt proof scripts is an activity one can hardly avoid. For example around 2009 the way algebraic structures are represented changed substantially with a major impact on most of the files.

The maintenance cost of the Mathematical Components library and the proof of the Odd Order Theorem were kept under control thanks to the disciplined way in which the formal proofs are written. In this section we cover some of the aspects that we believe do contribute in a substantial way to write maintainable formal proofs. Coming up with a list of unobjectionable golden rules would be quite pretentious, and probably impossible to do, but we will anyway try to give some advices out of our experience.

The first and most important one is that formal proofs are just computer code about mathematical proofs, hence *good practices in both domains* are likely to apply to formal proofs too.

For example one can claim that a well written pen and paper proof makes evident and clear what is interesting and hides via suitable abstractions uninteresting details. Finding the right abstraction is crucial in formal proofs too, and the related concept of triviality is key to keep the desired level of abstraction. More specifically, each mathematical object comes with properties that are pervasively and silently used in each and every proof. In order to make a formal proof look like pen and paper one, this conciseness has to be retained, or the noise of formality risks to hide the salient part of the proof.

A striking similarity lies in the usability of lemmas compared to the one of computer code procedures (functions, or methods). It is hard to invoke a procedure that requires passing too many arguments to it. Even worse if order (or name) of these arguments is hard to remember. In order to ease invoking procedures related data is often grouped in packages that are passed all together, and the order of arguments is standardized across the whole codebase. Similarly the statement of a lemma can be designed so that it minimizes the information one has to explicitly provide in order to use it in conjunction with a specific tactic.

Finally, it is common sense to organize computer code in components with separate concerns and well defined interfaces: Unstructured code is well known to be hard to read, understand and locally modify. Also, mathematical theories are often organized and structured, for example by algebraic means. The axioms characterizing structures are not

unlike interfaces for computer code. Identifying the right components and their interfaces is key in a library of formal proofs too and is the subject of Chapter 8.

### 4.3.1 On the readability of formal proofs

A desirable characteristic of formal proofs is readability, there is no questioning that: How can a proof be maintained if it cannot even be read?

It is not uncommon for novice users to try to achieve readability by writing a single instruction per line, and relying on the capability of user interfaces to step through each and every step in order to understand what is going on. We call these proofs *vertical proofs*: they are lengthy and their structure, when present, is hard to find because, at a glance, one can hardly look at the entire text. It is a bit like typesetting a book by breaking the line after each and every word: in order to find the action the subject is performing one would need to turn page. Lengthy proofs without an apparent structure are hard to maintain: even manipulating their text is cumbersome. The last, and most important observation drawn from vertical proofs, is that it is not the “atomicity” of steps that makes the proof readable, but rather the fact that goals, that are intermediate statements, are more readable than proof steps.

Conversely, an expert Small Scale Reflection user is typically capable of squeezing into a single line so many steps that nobody could understand the resulting *horizontal proof*, even if it requires no page flipping. The game of “compressing proofs” is a typical exercise to learn the very articulate Small Scale Reflection proof language, and it is often mistaken by students as the recommended way of writing proofs in the large. Horizontal proofs are indeed short, hence handy to manipulate, but their poor readability is not always helping maintenance.

The funny fact is that the tension between horizontal and vertical proofs is totally artificial: there is no need to pick one direction and only one in a proof. Indeed, in order to obtain a readable and maintainable proof one can mix both styles! Another way to put it is that in a proof one can write readable-but-lengthy proof script for interesting steps and obscure-but-compact scripts to deal with the bureaucracy of side conditions.

An undoubtedly *readable proof step* is a declarative step asserting, in the proof text, a statement that holds. For example by using the `have` tactic. This step is as readable as the statements of theorems is: It goes without saying that curating the way statements can be written plays a main role in readability. Moreover, given that the statement is part of the proof text, one does not need the assistance of COQ in order to see it.

A very related source of unreadability is the choice of meaningless names for theorems, variables or hypotheses, e.g. `H42`. By asking COQ to display the goal one can see what the 42nd hypothesis is about, but without that one can hardly imagine. A name like `1eq_nm` is a more suggestive name for an assumption stating that `n` is smaller than `m`. Similarly `n` is a good name for a number, `A` for a set or a matrix, `s` for a sequence, and `G` for a group while `x7'` is not going to help the reader much.

*Obscure proof steps* are not declarative: They act on a specific proof state and perform a precise manipulation. For example “apply commutativity to the first addition on the right hand side of the goal”. Without seeing the actual goal, one cannot understand what the precise action is actually achieving. Another characteristic of these steps is the use of symbolic notations for some operations, such as getting rid of trivial goals via `//`, that help squeezing even more actions in a single line. One can only follow and understand a few obscure steps without the help of COQ.

Still a few tricks to help reading these steps can be given. Symbols can be grayed out, the whole point of having symbolic notation is indeed to have a less intrusive syntax for

an uninteresting manipulation. In addition to that in the Mathematical Components library short names are usually reserved for uninteresting theorems, e.g. `addmCA` to shuffle summands around. What is left is the salient contents of the obscure step.

To sum up our view: Readable proofs are structured by forward steps, and their length is kept under control by proving these steps in a compact way.

Is this way of writing proofs tied to the technology we have today?

While one can imagine a future where only declarative, readable, steps are part of a formal proof, and all their justifications can be silently omitted thanks to automation, we are clearly not there today. When a gap cannot be filled in automatically having the ability to do it by hand, rather than by declaring unnatural intermediate steps just to drive the system, proved to be quite effective since the way the machine is guided is more direct and precise. Also, and more importantly in the long term, explicit proof steps provide a solid ground for repair, while their absence may turn out to be an impeding factor for maintenance.

An advice one can hardly imagine to become irrelevant in the future is the following one. Structure formal proofs as they are structured on paper. Then, when filling the blocks, use one line of formal text to perform one meaningful proof step for a human reader, and use the best tools you have to achieve that: At worst, if the step is too obscure, the reader will ignore the text, and just see what happens by executing it. In other words, the flow of a proof script should be homogeneous and distil human-readable information at a suitable, regular pace.

### 4.3.2 Fail early and locally to ease repair

There is a trade off between formal proofs that are robust in face of changes and formal proofs that are easy to repair. With robustness we mean that the very same formal proof text works even if the statements of the invoked lemmas change. This is often achieved by making proof commands perform some automatic search that can compensate for the changes. The concept of a robust proof is surely tantalizing but we believe it is also a double-edge sword: The more sophisticated robustness is, the harder it can get to identify the root cause of the problem when things break (and sometimes they do, no matter how robust things are). On the contrary the failure of a predictable, dumb, tool is easy to understand, if only because it is simpler and less sophisticated.

The tenet of the Small Scale Reflection proof language is that predictable failure leads to easy to repair proofs.

As a consequence most Small Scale Reflection tactics fail early and locally: If they cannot operate as expected, they fail rather than silently perform a no operation or try to be smart and guess what one may have wanted to do. This helps errors to be discovered early in the proof script and hence localize precisely where things actually broke. It is a matter of style and discipline to preserve this property when composing Small Scale Reflection tactics. Indeed it is sometimes desirable to relax this behavior, but it has to be done with care.

For example by applying the `?` modifier in rewrite rules one can make a rewrite step never fail. While this is very useful in practice, abusing this feature typically results in scripts that are hard to repair. The recurrent idiom is `rewrite lem1 ?lem2 // lem3`. Here `lem2` is used to trivialize a side condition of `lem1` that is then closed by `//`. The `lem3` applies to the main goal after the rewriting of `lem1`, and the next line is going to take on from that point. Now imagine what happens if the side condition of `lem1` changes and `lem2` is not helping anymore in trivializing it and hence `//` cannot solve it. In this scenario `lem3` is likely to fail, since it is meant to apply to the main goal, and not the side condition.



Writing “`rewrite lem1 ?lem2 ?lem3 //`” can have the same meaning but is harder to repair. In particular one would discover the breakage later, in the next line, since after `lem1` all actions never fail. Also, the next line is likely is going to be run on the side condition, and not the main goal. The person repairing the proof is hence put on the wrong track: the line that fails was not even written to operate on the goal she sees.

### 4.3.3 Checkpointing

Forward steps in large proofs play the role of checkpoints: when a proof breaks one can start replaying it from the closest forward step since it is granted that the goal is the very same the original author was seeing when he wrote the proof. If there are no checkpoints one may be forced to replay the proof from the very beginning.

A recurrent idiom in the Mathematical Components library is the one of refining an assumption by replacing it with a stronger one. For example to refine `H` one can write `have {H}H : ty`, or even `have {H} : ty` in recent versions of the Small Scale Reflection language.

A related practice is the one of explicitly clearing hypothesis when they are not needed anymore. The first beneficial effect is that the context as displayed by COQ becomes smaller, easier to read. Also, by making the lifetime of context entries explicit, one can more easily read the dependencies among blocks, and that is a useful piece of information when one refactors proofs. To help keeping the context tidy, most tactics clear their arguments automatically, unless told otherwise. For example `apply`: `H` also removes `H` from the context; similarly `move`: `H` or `case`: `H`.

### 4.3.4 Large, monolithic, proofs: a fact of life

Somehow counter-intuitively, large proofs cannot always be broken down into smaller lemmas. More precisely, chopping long proofs in an artificial manner is often counterproductive.

In particular, as contexts grow larger, gathering numerous facts and local definitions, it becomes harder to extract an intermediate result and state it outside the proof mode of the main result. Indeed, the statement of such an auxiliary lemma would probably need to include a large chunk of the big context, in order to remain provable. Besides the trouble of copy-pasting this portion of the context, a lemma with too many hypotheses soon becomes very hard to use, as instantiating the latter becomes a technical issue. Last, it is often quite difficult to find an appropriate name for such artificial lemmas.

The `pose` and `set` tactics provide facilities for stating local definitions in proof mode. In particular, the `set` tactic eases the definition of an abbreviation for a given subterm of the current goal, by allowing to describe the latter via a pattern. The `have` tactic, and its variants `suffices` and `wlog`, provide syntactic facilities to state and use local definitions and lemmas to a main proofs, with similar features as the `Lemma` vernacular command (and its synonyms), like the automated introduction of named parameters. The `wlog` tactic goes one step further, and allows to describe the verbose cut statements hidden behind a *without loss of generality* proof pattern by providing only the names of the hypotheses in the context which should be included in the generalization. This feature saves the user from an painful manual writing of the explicit cut statement.

### 4.3.5 On the usability of lemmas

The usability of lemmas contributes to maintenance only indirectly, that is by making proofs shorter. Still it plays a crucial role in a library so it is worth spending a few words on it.

Most of the lemmas in the Mathematical Components libraries are equalities, meant to be used with the `rewrite` tactic. Let's take for example  $(\text{mul0n} : \forall n, 0 * n = 0)$ . Two equivalent lemmas, mathematically speaking, are  $(\text{mulnm\_n0} : \forall n m, n = 0 \rightarrow n * m = 0)$  and  $(\text{mul0\_0n} : \forall n, 0 = 0 * n)$ .

Let's see why `mul0n`, the one present in the library, is easier to use. The `rewrite` tactic uses the left hand side of the conclusion as a pattern and looks for sub terms of the goal that match. The pattern provided by the `mul0n` is  $(0 * \_)$  and is quite precise, only the right hand side of the multiplication is unknown. Moreover the presence of `0` is very related to the meaning of the lemma, that shows it is the annihilator of multiplication.

The pattern for `mulnm_n0` is, on the contrary, the very imprecise  $(\_ * \_)$ . This pattern is matched by any multiplication in the goal, including the ones where the left hand side is not (provably) zero. As a consequence one would need to drive `rewrite` manually, either by providing a more specific pattern or by specializing the lemma passing to it some arguments.

The pattern for `mul0_0n` is `0`. While it is very precise `rewrite` does not know what to put in the resulting goal, that is it cannot find out what `n` is by just matching the pattern. As a consequence this argument has to be passed explicitly. Of course `mul0_0n` can be used right-to-left, but it requires the `-` switch to `rewrite`. In some sense its "default" is wrong.

By stating equalities with the most informative term on the left one obtains lemmas that easier to rewrite with, since their arguments and the sub term of the goal to be replaced are going to be found, automatically, by pattern matching.

A similar line of reasoning can be applied to other tactics as well. For example `apply: lem` infers some of the arguments to `lem` by unifying the conclusion of its statement with the goal. The position of the arguments that are not inferable this way can be chosen so that it is easy to pass them explicitly. For example  $(\text{leq\_trans} : \forall y x z, x \leq y \rightarrow y \leq z \rightarrow x \leq z)$  quantifies first the variable that cannot be inferred by unifying the conclusion with the goal (since it does not occur in the conclusion). It is hence sufficient to write  $(\text{leq\_trans } m)$  to specify the only missing piece of information.

#### 4.3.6 Qed is not the shift bell

When a proof is finally accepted by COQ it just means it is correct: the absence of mistakes mistake correct for readable or maintainable.

As we argued so far that proof is going to eventually break, and repairing a tidy proof is much simpler than repairing a messy one. The advantage we have compared to, say, cleaning up regular computer code, is that once a proof is complete and correct we can iterate small cleanup steps much faster. Indeed we can have COQ check the proof after each and every step: If it checks, we did introduce no error and we can move on with the next cleanup iteration.

Unfortunately little support is given to the user to tidy proofs up by today's user interfaces, even for very recurrent steps. Among all, one that is very frequent is the removal of duplication within the current proof or across the entire library. It is quite frequent to end up re-proving, inline, an existing lemma when one does not know the lemma exists (or does not find it).



## 5. Inductive specifications

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:

```
1 Fixpoint has T (a : T -> bool) (s : seq T) : bool :=  
2   if s is x :: s' then a x || has a s' else false.
```

or we can use an alternate formula, like for instance:

```
1 Definition has_prop T (a : T -> bool) (s : seq T) :=  
2   exists x0 : T, exists i, i < size s /\ a (nth x0 s i)
```

or the equivalent variant:

```
1 Definition has_prop T (a : T -> bool) (x0 : T) (s : seq T) :=  
2   exists i, i < size s /\ a (nth x0 s i)
```

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:

```
1 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 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:

```
1 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.<sup>1</sup> Specifications in the `Prop` sort are structured logical statements, that can be “deconstructed” 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.

## 5.1 Reflection views

### 5.1.1 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:

```
1 Definition bool_Prop_equiv (P : Prop) (b : bool) := b = true <-> P.
```

where  $(A \leftrightarrow B)$  is defined as  $((A \rightarrow B) \wedge (B \rightarrow A))$ .

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:

<sup>1</sup>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 7.

```

1 Lemma test_bool_Prop_equiv b P : bool_Prop_equiv P b -> P ∨ ~ P.
2 Proof.
3   case: b; case => hlr hrl.
4   by left; apply: hlr.
5   by right => hP; move: (hrl hP).
6 Qed.

```

The last goal, just before line 5 is executed, is the following one:

```

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  $(b = \text{true} \wedge P) \vee (b = \text{false} \wedge \sim P)$ , but again the bureaucracy would be non-negligible.

A first improvement on this naive approach consists in using 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:

```

1 Inductive reflect (P : Prop) (b : bool) : Prop :=
2 | ReflectT (p : P) (e : b = true)
3 | 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`.

Let us illustrate the benefits of this alternate specialized double implication:

```

1 Lemma test_reflect b P :
2   reflect P b -> P ∨ ~ P.
3 Proof.
4 case.
5
6
7

```

```

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 5.2.1.

We start our collection of links between boolean and `Prop` statements with the lemmas relating boolean connectives with their `Prop` versions:

```

1 Lemma andP (b1 b2 : bool) : reflect (b1 /\ b2) (b1 && b2).
2 Proof. by case: b1; case: b2; [ left | right => // = [[1 r]] ..]. Qed.
3
4 Lemma orP (b1 b2 : bool) : reflect (b1 ∨ b2) (b1 || b2).
5 Proof.

```

```

6 case: b1; case: b2; [ left; by [ move | left | right ] .. ].
7 by right=> // [[l|r]].
8 Qed.
9
10 Lemma implyP (b1 b2 : bool) : reflect (b1 -> b2) (b1 ==> b2).
11 Proof.
12 by case: b1; case: b2; [ left | right | left .. ] => // = /(_ isT).
13 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 “;[  $\tau_1$  |  $\tau_2$  .. |  $\tau_n$  ]” syntax indeed corresponds to the application of the tactic  $\tau_1$  to the first subgoal generated by what comes before ;, and the application of the tactic  $\tau_n$  to the last subgoal, and the application of the tactic  $\tau_2$  to all the branches in between. See [Coq, “The tactic language”] 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.

**R** 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.

### 5.1.2 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 3.6) 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:

```

1 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.
2
3

```

```

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).
5
6
7
```

```
n : nat
m : nat
=====
m = n -> eqn m n

subgoal 2 is:
eqn m n -> m = n
```

In both cases the proof is now an easy induction on `m`.

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).
7
8
9
```

```
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.

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.

### 5.1.3 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 4.1. 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:

```

1 Lemma example n m k : k <= n ->
2   (n <= m) && (m <= k) -> n = k.
3 Proof.
4 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))`.

```

1 Lemma elimTF (P : Prop) (b c : bool) :
2   reflect P b -> b = c -> if c then P else ~ P.

```

Term `(elimTF andP top)` hence has type

```

1 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:

```

1 Lemma example n m k : k <= n ->
2   (n <= m) && (m <= k) -> n = k.
3 Proof.
4 move=> lekn /andP[lekm lemk].
5
6

```

```

n, m, k : nat
lekn : k <= n
lekm : n <= m
lekm : 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 `lekm` hypothesis:

```

1 Lemma example n m k : k <= n ->
2   (n <= m) && (m <= k) -> n = k.
3 Proof.
4 move=> lekn /andP [/eqnP lekm lemk].
5
6

```

```

n, m, k : nat
lekn : k <= n
lekm : n - m = 0
lekm : m <= k
=====
n = k

```

**R** 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.



### 5.1.4 Using views with tactics

We dissect the proof that  $\leq$  is a total relation. As usual the statement is expressed as a boolean formula:

```
1 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:

```
1 Lemma leq_total m n :
2   (m <= n) || (m >= n).
3 Proof.
4 rewrite -implyNb.
```

```
m, n : nat
=====
~~ (m <= n) ==> (n <= m)
```

This premise can be seen as  $n < m$ :

```
1 Lemma leq_total m n :
2   (m <= n) || (m >= n).
3 Proof.
4 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:

```
1 Lemma leq_total m n :
2   (m <= n) || (m >= n).
3 Proof.
4 rewrite -implyNb -ltnNge; apply/implyP.
```

```
m, n : nat
=====
(n < m) -> (n <= m)
```

We can now conclude the proof:

```
1 Lemma leq_total m n : (m <= n) || (m >= n).
2 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:  $n1 \leq n2$  and  $n2 \leq n1$ .

```
1 Lemma leq_max m n1 n2 :
2   (m <= maxn n1 n2) = (m <= n1) || (m <= n2).
3 Proof.
4 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  $(n1 \leq n2)$  in its context.

Finally, the `rewrite` tactic also handles views that relate an equation in `Prop` with a boolean formula.

```

1 Lemma maxn_idP1 {m n} : reflect (maxn m n = m) (m >= n).
2
3 Lemma leq_max m n1 n2 :
4   (m <= maxn n1 n2) = (m <= n1) || (m <= n2).
5 Proof.
6 case/orP: (leq_total n2 n1) => [le_n21 | le_n12].
7   rewrite (maxn_idP1 le_n21).

```

The tactic sees `(maxn_idP1 le_n21)` as the equation corresponding to the boolean formula `le_n21`, namely  $(\text{maxn } n1 \ n2 = n1)$ , 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` has been discussed in detail in section 4.2.2.

## 5.2 Advanced, practical, statements

In this section, we will further explore the possibility to improve the flow of formal proofs by refining the shape of the definitions specifying a given concept.

### 5.2.1 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 3.6. In particular their elimination rule encapsulates the notion of substitution, and that operation is performed automatically by the logic.

For instance, let us recall the definition of the `reflect` predicate proposed in Section 5.1.1.

```

1 Inductive reflect (P : Prop) (b : bool) : Prop :=
2   | ReflectT (p : P) (e : b = true)
3   | ReflectF (np : ~ P) (e : b = false)

```

When making use of this predicate in a proof, it is often the case that substituting `b` for its value in each branch of the case is needed right after the case analysis.

The following 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  $(\forall a b, \text{reflect } (a \wedge b) (a \ \&\& \ b))$ . The definition of the `andP` view is accompanied by the following configuration of its arguments:

```
1 Arguments andP {a b}.
```

We recall that this makes `a` and `b` implicit, hence writing `andP` is equivalent to `(@andP _ _)`, a term whose type is `(reflect (_ \wedge _) (_ \&\& _))`. The value of the index of the inductive family, to be replaced by `true` and `false` in the respective branches of a case analysis, is thus a pattern `(_ \&\& _)`. Therefore, the goal is searched for an instance of such pattern. The matching algorithm used to find an instance of this pattern is the same as one used by the `rewrite` tactic, for the instantiation of rewrite rules.

The following proof script illustrates how to use deal with a boolean conjunction `a \&\& b`, which appears ad the premise of a boolean implication, almost as conveniently as if it were a conjunction in `Prop`.

```
1 Lemma example a b :
2   a \&\& b ==> (a == b).
3 Proof.
4 case: andP => [ab|nab].
5
6
```

```
a, b : bool
ab : a \wedge b
=====
true ==> (a == b)

subgoal 2 is:
false ==> (a == b)
```

Note that we have not specified a value for the variables quantified in the statement of `andP`: the tuning of implicit arguments is key to making lemmas easy to use, even more so in the case of the “spec” ones. Thanks to the index of the `reflect` inductive, 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 following proof script:

```
1 Lemma example a b : a \&\& b ==> (a == b).
2 Proof. by case: andP => [[-> ->] |]. Qed.
```

Note that if one needs to override the pattern inferred for the index of `andP`, one can provide it by hand as follows:

```

1 Lemma example a b : (a || ~~ a) && (a && b ==> (a == b)).
2 Proof. by case: (a && _) / andP => [[-> ->] |] //; rewrite orbN. Qed.

```

A more detailed explanation of this syntax can be found in [GMT15, section 5.6].

The Mathematical Components library provides many spec lemmas to be used this way. A paradigmatic one is `ifP`.

```

1 Section If.
2 Variables (A : Type) (vT vF : A) (b : bool).
3
4 Inductive if_spec : bool -> A -> Type :=
5 | IfSpecTrue (p : b)          : if_spec true vT
6 | IfSpecFalse (p : b = false) : if_spec false vF.
7
8 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.

**R** 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.

### 5.3 Strong induction via inductive specs

As we have seen in Section 3.7 the strong induction principle can be obtained from the regular one by choosing an appropriate value for the predicate. Recall that the `elim` tactic infers that predicate from the goal, so another way to specify the predicate is to *load* the goal before performing the induction. For example if the goal is `(G n)` and one loads it to `( $\forall m, m <= n \rightarrow G m$ )` then he will have access to the induction hypothesis on all numbers smaller than `n` exactly as if he was using the strong induction principle.

It is possible to do so very concisely using the discharging operator “:”. In particular

```

1 move: {-2}n (leqnn n)

```

does exactly what we need for a goal like  $(G\ n)$ : it first pushes to the stack  $(n \leq n)$  and then pushes to the stack  $n$  capturing all occurrences but for the second one. The Mathematical Components library provides a few tools to load the goal like this without recurring to occurrence selection, which can be tricky and fragile.

The following inductive specification is crafted so that its only constructor holds as arguments exactly the items we pushed on the stack with the discharge operator.

```
1 Inductive ubn_geq_spec m : nat -> Type :=
2   UbnGeq n of n <= m : ubn_geq_spec m n.
3 Lemma ubnPgeq m : ubn_geq_spec m m.
4 Proof. by []. Qed.
```

As a consequence by performing a case analysis on  $(\text{ubnPgeq } n)$  one obtains the same effect of `move: -2n (leqnn n)`.

```
1 Lemma test_ubnP (G : nat -> Prop) n : G n.
2 Proof.
3 case: (ubnPgeq m).
```

```
G : nat -> Prop
n : nat
=====
∀ m, m <= n -> G m
```

By varying the arguments of the inductive specification constructor one can tune the way the goal is loaded. The Mathematical Components library provides a few inductive specifications like the one explained in this section, and their names contain the string “ubn” for “upper bound”.

## 5.4 Showcase: Euclidean division, simple and correct

In this section we use most of the techniques and tactics seen so far to prove correct the Euclidean division algorithm.

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 remainder 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 case: 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.

But let's start by dealing with the trivial case of a null divisor.

```

1 rewrite -[m]/(0 * d + m).
2 case: d => [//= | d /=] in ed *.
3 rewrite -[edivn m d.+1]/(edivn_rec d m 0) in ed *.
4 case: (ubnPgeq m) @ed; elim: m 0 => [|m IHm] q [/=|n] leq_nm //.
5 rewrite edivn_recE subn_if_gt; case: ifP => [le_dm ed|lt_md]; last first.
6   by rewrite /= ltnS ltnNge lt_md eqxx.
7 rewrite -ltnS in le_dm; rewrite -(subnKC le_dm) addnA -mulSnr subSS.
8 by apply: IHm q.+1 (n-d) _; apply: leq_trans (leq_subr d n) leq_nm.
```

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 [GMT15, 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  $o$ , leading to the following goal:

```
d, m : nat
IHm : ∀ q n, n <= m ->
  let ed := edivn_rec d n q in
  (ed.2 < d.+1) && (q * d.+1 + n == ed.1 * d.+1 + ed.2)
q, n : nat
leq_nm : n < m.+1
=====
let ed := edivn_rec d n.+1 q in
(ed.2 < d.+1) && (q * d.+1 + n.+1 == ed.1 * d.+1 + ed.2)
```

Note that the case analysis on  $(\text{ubnPgeq } m)$  needs to grab also the occurrence of  $m$  in the body of  $\text{ed}$ . To do so we push  $\text{ed}$  on the goal stack prior the case analysis. The  $\text{e}$  modifier tells Small Scale Reflection to keep the body of the let-in, which would be otherwise deleted.

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. We are left with this goal to prove:

```
d, m : nat
IHm : ∀ q n, n <= m ->
  let ed := edivn_rec d n q in
  (ed.2 < d.+1) && (q * d.+1 + n == ed.1 * d.+1 + ed.2)
q, n : nat
leq_nm : n < m.+1
le_dm : d <= n
ed := edivn_rec d (n - d) q.+1 : nat * nat
=====
(ed.2 < d.+1) && (q * d.+1 + n.+1 == ed.1 * d.+1 + ed.2)
```

Line 7 prepares the goal to accommodate the application of the induction hypothesis, instantiated at  $q.+1$  and  $(n - d)$ .

```
(ed.2 < d.+1) && (q.+1 * d.+1 + (n - d) == ed.1 * d.+1 + ed.2)
```

Finally, line 8 uses the induction hypothesis and proves its premise  $(m - d <= n)$ .

## 5.5 Notational aspects of specifications

When a typing error arises, it always involves three objects: a term  $\tau$ , its type  $\text{ity}$  and the type expected by its context  $\text{ety}$ . Of course, for this situation to be an error, the two types  $\text{ity}$  and  $\text{ety}$  do not compare as equal. The simplest way one has to explain COQ how to fix  $\tau$ , is to provide a functional term  $c$  of type  $(\text{ity} \rightarrow \text{ety})$  that is inserted around  $\tau$ . In other words, whenever the user writes  $\tau$  in a context that expects a term of type  $\text{ety}$ , the system instead of raising an error replaces  $\tau$  by  $(c \tau)$ .

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
```

```
=====
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 it is 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  $\delta$  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: in the following example, `true` is used as the tail of a `seq`. Even though there is no direct coercion from `bool` to `seq`, Coq can go from one to the other via `nat`. Thus `true` is first coerced to 1 of type `nat` by `nat_of_bool`, which is then turned into a `seq` (of length 1 containing 0s) by `zerolist`.

```
1 Definition zerolist n := mkseq (fun _ => 0) n.
2 Coercion zerolist : nat -> seq.
3 Check 2 :: true == [:: 2; 0].
```

The reader already familiar with the concept of coercion may find the presentation of this chapter slightly nonstandard. Coercions are usually presented as a device to model a flavor of subtyping in a type theory that, like the Calculus of Inductive Constructions, does not enjoy such a feature. But as will become clear in Chapter 8, the role played by coercions in the modeling of the hierarchy of algebraic structures is actually minor. In fact, it is limited to the easy task of forgetting some fields of a structure to obtain a simpler one. The trickiest part of setting up a hierarchy is actually the reconstruction of the missing fields of a richer structure from a weaker one, and the comparison of two structures in order to find the minimum super structure. These tasks are mainly implemented by programming type inference, which is the main topic of the next part of the book.





# Crafting a formal library

<b>6</b>	<b>Implicit parameters</b> .....	<b>115</b>
6.1	Type inference and higher-order unification	
6.2	Type inference by example	
6.3	Records as relations	
6.4	Records as (first-class) interfaces	
6.5	Using a generic theory	
6.6	The generic theory of sequences	
6.7	The generic theory of “big” operators	
6.8	Stable notations for big operators	
6.9	Working with overloaded notations	
6.10	Ad-hoc polymorphism	
<b>7</b>	<b>Sub-types</b> .....	<b>137</b>
7.1	$n$ -tuples, lists with an invariant on the length	
7.2	$n$ -tuples, a sub-type of sequences	
7.3	Finite types and their theory	
7.4	The ordinal subtype	
7.5	Finite functions	
7.6	Finite sets	
7.7	Permutations	
7.8	Matrix	
<b>8</b>	<b>Organizing Theories</b> .....	<b>155</b>
8.1	Structure interface	
8.2	Telescopes	
8.3	Packed classes	
8.4	Parameters and constructors	
8.5	Linking a custom data type to the library	





## 6. 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 [Pol92]. 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<sup>1</sup> 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 who is able to guess the intended meaning of a mathematical expressions from the context thanks to her background knowledge.

This chapter also introduces the key concepts of *interface* and *instance*. An interface is

---

<sup>1</sup>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.

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.

## 6.1 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).

## 6.2 Type inference by example

Most of the code we have discussed so far uses some amount of type inference. In this section, we go over a few elementary examples, so as to clarify them.

For once, let us start with an example of code meant to be executed in a COQ session before *importing any library, nor positioning any global option*. This example is one of the simplest possible: it defines the polymorphic identity function and checks its application to 3.

```
1 Definition idfun A (a : A) : A := a.
2 Check (idfun nat 3).
3 Check (idfun _ 3).
```

```
idfun nat 3 : nat
idfun nat 3 : nat
```

In the expression `(id nat 3)` no subterm was omitted, therefore COQ accepts the term and prints its type. In the third line, even if the subterm `nat` is omitted, COQ accepts the term. Type inference finds a value for the placeholder for the user by proceeding in the following way: it traverses the term recursively from left to right, ensures 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 leaves 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 placeholder has been replaced by the value type inference assigned to it.

In the light of this process, it becomes clear 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 common that one can ask the system to insert the right number of `_` for us. Here we only provide a simple example. For more details refer to [Coq, “Extensions of Gallina”].

```
1 Arguments idfun {A} a.
2 Check (id 3).
3 Check (@id nat 3).
```

```
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. In our case, argument `A` is thus declared as implicit. The declaration of implicit arguments can be locally disabled by prefixing the name of the constant with the `@` symbol.

Very often, it is possible to determine the most plausible status of each argument, explicit or implicit, using a heuristic. Such a heuristic is applied in a systematic way when a global option is declared: every definition posterior to the declaration may feature implicit arguments even if not `Argument` command is used to declare them manually. This has been

the case so far, and in the rest of the section and of the book, we will work again under the assumption that following the global options have been set, in accordance with Chapter ??:

```
1 Unset Strict Implicit.
2 Unset Printing Implicit Defensive.
```

Of course, the `Argument` command remains useful on specific cases when a finer, manual tuning of implicit arguments remains needed. The command `About` provides the user with the type of a given constant, and with the status, implicit or not, of each argument of the constant.

Another piece of information that is often not explicit is the type of abstracted or quantified variables.

```
1 Check (fun x => @id nat x).
2
3 Lemma prime_gt1 p :
4   prime p -> 1 < p.
5
```

```
fun x : nat => id x : nat -> nat
p : nat
=====
prime p -> 1 < p
```

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 time 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 placeholders were standing for types, but the user is also allowed to put `_` in place of a term.

```
1 Lemma example q : prime q -> 0 < q.
2 Proof.
3 move=> pr_q.
```

```
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`.

```
1 have q_gt1 := @prime_gt1 _ pr_q.
2 exact: ltnW q_gt1.
3 Qed.
```

In the expression `(prime_gt1 _ pr_q)`, the placeholder, 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 placeholder has been set to be `q`. As we did for the

identity function, we can declare the `p` argument of `prime_gt1` as implicit: this is actually the case in the library. In this case, the script is:

```
1 have q_gt1 := prime_gt1 pr_q.
2 exact: ltnW q_gt1.
3 Qed.
```

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 form 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 [Asp+12]. The reader interested in a technical presentation of a simplified version of the unification algorithm implemented in COQ can read [Gon+13b; SZ14].

### 6.3 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:

```
1 Inductive eqType : Type :=
2   Pack sort of sort -> sort -> bool.
3 Definition sort (c : eqType) : Type :=
4   let: Pack t _ := c in t.
5 Definition eq_op (c : eqType) : sort c -> sort c -> bool :=
6   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.

```
1 Definition nat_eqType : eqType := @Pack nat eqn.
```

Projections, when applied to a record instance like `nat_eqType` compute and extract the desired component.

```
1 Eval simpl in sort nat_eqType.
2 Eval simpl in @eq_op nat_eqType.
```

```
= 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:

```
1 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<sup>2</sup>. 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.
```

<sup>2</sup>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 6.4.



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.

```
1 Notation "x == y" := (@eq_op _ x y).
2 Check (@eq_op bool_eqType true false).
3 Check (@eq_op nat_eqType 3 4).
```

```
true == false : bool
3 == 4 : bool
```

As a printing rule, the placeholder 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 placeholder 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 placeholder  $?_e$ . 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 6.1: 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` is `bool`. In the language of [Canonical Structures](#), such a program is expressed as follows.

```
1 Canonical nat_eqType.
2 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.

```
1 Check (3 == 4).
2 Check (true == false).
3 Compute (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.

(projection ?*s*) versus 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 play 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. 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.

```
1 Check (3, true) == (4, false).
2
3
```

```
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.

```
1 Definition prod_cmp eqA eqB x y :=
2   @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.

```

1 Definition prod_eqType (eqA eqB : eqType) : eqType :=
2   @Pack (sort eqA * sort eqB) (@prod_cmp eqA eqB).
3 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

$$(\text{sort } ?_e) \text{ versus } (T1 * T2)$$

is encountered, a solution for  $?_e$  is found by proceeding in the following way. Two new unification problems are generated:  $(\text{sort } ?_{eqA})$  versus  $T1$  and  $(\text{sort } ?_{eqB})$  versus  $T2$ . If both are successful and  $v1$  is the solution for  $?_{eqA}$  and  $v2$  for  $?_{eqB}$ , the solution for  $?_e$  is  $(\text{prod\_eqType } v1 \ v2)$ .

After the table of canonical solutions has been extended, our example is accepted.

```

1 Check (3, true) == (4, false).

```

```
(3, true) == (4, false) : bool
```

The term synthesized by COQ is the following one:

```

1 @eq_op (prod_eqType nat_eqType bool_eqType) (3, true) (4, false).

```

## 6.4 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.


```

1 Module Equality.
2
3 Structure type : Type := Pack {
4   sort : Type;
5   op : sort -> sort -> bool;
6   axiom : ∀ x y, reflect (x = y) (op x y)
7 }.
8
9 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.

 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`, `op` and `axiom`, four 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 6.3, the record plays the role of a relation and its `sort` component is again the only field that drives canonical structure inference. 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:

```


1  Module Equality.
2
3  Definition axiom T (e : rel T) := ∀ x y, reflect (x = y) (e x y).
4
5  Record mixin_of T := Mixin {op : rel T; _ : axiom op}.
6  Notation class_of := mixin_of.
7
8  Structure type : Type := Pack {sort :> Type; class : class_of sort; }.
9
10 End Equality.
11
12 Notation eqType := Equality.type.
13 Definition eq_op T := Equality.op (Equality.class T).
14 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.

That being said, 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 it possible to write  $(\forall T : \text{eqType}, \forall x y : T, P)$  even if `T` is not a type and only `(sort T)` is.

 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 a function is correct. Declaring the `eqType` instance for `nat` now requires some extra work, namely proving the correctness of the `eqn` function.

```

1 Lemma eqnP : Equality.axiom eqn.
2 Proof.
3 move=> n m; apply: (iffP idP) => [|<-]; last by elim n.
4 by elim: n m => [|n IHn] [|m] // = /IHn->.
5 Qed.

```

We now have all the pieces to declare `eqn` as canonical.

```

1 Definition nat_eqMixin := Equality.Mixin eqnP.
2 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.

## 6.5 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.

```

1 Lemma eqP (T : eqType) : Equality.axiom (@Equality.op T).
2 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`

```

1 Lemma test (x y : nat) : x == y -> x + y == y + y.
2 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`:

```

1 Section ProdEqType.
2 Variable T1 T2 : eqType.
3
4 Definition pair_eq := [rel u v : T1 * T2 | (u.1 == v.1) && (u.2 == v.2)].
5
6 Lemma pair_eqP : Equality.axiom pair_eq.
7 Proof.
8 move=> [x1 x2] [y1 y2] /=; apply: (iffP andP) => [|<|<- <-] // =.
9 by move/eqP->; move/eqP->.
10 Qed.
11
12 Definition prod_eqMixin := Equality.Mixin pair_eqP.
13 Canonical prod_eqType := @Equality.Pack (T1 * T2) prod_eqMixin.
14 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)`

```
1 Lemma test (x y : nat) (a b : bool) : (a,x) == (b,y) -> fst (a,x) == b.
2 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`.

```
1 Lemma test (x y : nat) : (true,x) == (false,y) -> false.
2 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.

```
1 Lemma test_EM (x y : nat) : if x == y.+1 then x != 0 else true.
2 Proof. by case: ifP => // /eqP ->. Qed.
```

**R** Whenever we want to state equality between two expressions, if they live in an `eqType`, always use `==`.

## 6.6 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 a theory. If we try to develop the theory of the type `(seq T)` for an arbitrary `T`, we can't go very 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.

```
1 Section SeqTheory.
2 Variable T : eqType.
3 Implicit Type s : seq T.
4
```

```

5 Fixpoint mem_seq s x :=
6   if s is x :: 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, as well as a procedure for removing duplicates from a sequence.

```

1 Fixpoint uniq s :=
2   if s is x :: s' then (x \notin s') && uniq s' else true.
3 Fixpoint undup s :=
4   if s is x :: s' then
5     if x \in s' then undup s' else x :: undup s'
6   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.

```

1 Lemma in_cons y s x : (x \in y :: s) = (x == y) || (x \in s).
2 Proof. by []. Qed.
3
4 Lemma mem_undup s : undup s =i s.
5 Proof.
6 move=> x; elim: s => // = y s IHs.
7 case Hy: (y \in s); last by rewrite in_cons IHs.
8 by rewrite in_cons IHs; case: eqP => // ->.
9 Qed.

```

where  $(A =i B)$  is a synonym for  $(\forall x, x \in A = x \in B)$ ,

The `in_cons` lemma is just a convenient 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`).

```

1 Lemma undup_uniq s : uniq (undup s).
2 Proof.
3 by elim: s => // = x s IHs; case sx: (x \in s); rewrite // = mem_undup sx.
4 Qed.

```

The proof of `undup_uniq` requires no new ingredients and completes the specification of `undup`.

The 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.

```

1 Fixpoint eqseq s1 s2 {struct s2} :=
2   match s1, s2 with
3     | [::], [::] => true
4     | x1 :: s1', x2 :: s2' => (x1 == x2) && eqseq s1' s2'
5     | _, _ => false
6   end.
7
8 Lemma eqseqP : Equality.axiom eqseq.
9 Proof.
10 elim=> [|x1 s1 IHs] [|x2 s2] / =; do? [exact: ReflectT | exact: ReflectF].
11 case: (x1 =P x2) => [<-|neqx]; last by apply: ReflectF => -[eqx _].

```

```

12 by apply: (iffP (IHs s2)) => [<-| []].
13 Qed.
14
15 Definition seq_eqMixin := Equality.Mixin eqseqP.
16 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 corresponding to `ReflectF` has its negation  $x1 \neq x2$ .

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`.

```

1 Let s1 := [:: 1; 2 ].
2 Let s2 := [:: 3; 5; 7].
3 Let ss : seq (seq nat) := [:: s1 ; s2 ].
4 Check (ss != [::]) && s1 \in ss.
5 Check undup_uniq ss.

```

As we have anticipated in chapter 1, functional programming and lists can model definite, iterated operations like the “big” sum  $\Sigma$ . The next section describes how the generic theory of iterated operations can be built and made practical thanks again to programmable type inference.

## 6.7 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:

$$\sum_{i=1}^n f(i) = f(1) + f(2) + \dots + f(n) \quad \bigcup_{a \in A} g(a) = g(a_1) \cup g(a_2) \cup \dots \cup g(a_{|A|})$$

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, sometimes 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  $m, m+1, \dots, m+(n-m)-1$  corresponding to the range  $m \leq i < n$  can be built using the `iota` function as follows:



```
1 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.

$$\sum_{i < 2n, 2 \nmid i} i + \sum_{i < 2n, 2 \mid i} i = \sum_{i < 2n} i$$

Last, not all filtering conditions can be naturally expressed in the general term. An example is not being a prime number.

In the light of that, our formal statement concerning the sum of odd numbers is the following:

```
1 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:

```
1 Lemma sum_odd n :
2   foldr (fun acc i => if odd i then i + acc else acc)
3     0 (index_iota 0 n.*2)
4   = n^2
```

The following section details how the generic notation for iterated operation is built and specialized to frequent operations like  $\Sigma$ . Section 6.7.2 focuses on the generic theory of iterated operations.

### 6.7.1 The generic notation for `foldr`

The generic notation for iterated operations has to be attached to a slightly specific variant of the `foldr` iterator, in order to clearly identify all components. This variant, called `bigop`, takes as argument a range ( $r : \text{seq } I$ ), a neutral element ( $op : R$ ), and a binary operation ( $op : R \rightarrow R \rightarrow R$ ), to be iterated. Note that the type of the elements in the range are not the same as the type of the arguments of the iterated operation. Indeed, `bigop` also uses a function ( $F : I \rightarrow R$ ), as well as a filtering predicate ( $P : \text{pred } I$ ), to construct the actual iteration range. In the end, `bigop` filters the range  $r$  using predicate  $P$ , and precomposes by the function  $F$  before iterating the binary operation  $op$ . And `idx` is the value output when the filtered range is empty.

```

1 Definition bigop R I idx op r (P : pred I) (F : I -> R) : R :=
2   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

```

1 Lemma sum_odd n :
2   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`.

```

1 Notation "\big [ op / idx ]_ ( i <- r | P ) F" :=
2   (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.

```

1 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.

```

1 Local Notation "+%N" := addn (at level 0, only parsing).
2 Notation "\sum_ ( i <- r | P ) F" :=
3   (\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:

```

1 Notation "\big [ op / idx ]_ ( m <= i < n | P ) F" :=
2   (bigop idx op (index_iota m n) (fun i : nat => P%B) (fun i => F))
3   : big_scope.
4 Notation "\sum_ ( m <= i < n ) F" :=
5   (\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.

```

1 Lemma sum_odd_3 : \sum_(0 <= i < 3.*2 | odd i) i = 3^2.
2 Proof.
3 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.

```
1 Lemma big_nat_recr n m F : m <= n ->
2   \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.

```
1 Lemma big_mkcond I r (P : pred I) F :
2   \big[*M/1]_(i <- r | P i) F i =
3     \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.

```
1 Lemma sum_odd_3 :
2   \sum_(0 <= i < 3.*2 | odd i) i = 3^2.
3 Proof.
4 rewrite big_mkcond big_nat_recr //.
5 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.

```
1 Lemma prod_fact_4 :
2   \prod_(1 <= i < 5) i = 4`.
3 Proof.
4 rewrite big_nat_recr //.
```

```
=====
\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.

### 6.7.2 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 8 adds other interfaces to the picture and organizes the bigop library around them.

```
1 Module Monoid.
2 Section Definitions.
3 Variables (T : Type) (idm : T).
4
5 Structure law := Law {
6   operator : T -> T -> T;
7   _ : associative operator;
8   _ : left_id idm operator;
9   _ : right_id idm operator
10 }.
```

The `Monoid.law` structure relates the operator (the key used by canonical structures) to the three properties of monoids.

We can then use this interface as a parameter for a bunch of lemmas, describing the theory shared by its instances. The `(Monoid.law idx)` type annotation may seem puzzling at first. It combines various mechanisms we have seen earlier in the book: once the section `Definitions` is closed, the `Variables (T : Type)` and `(idm : T)` are abstracted in the definition of `law` (and the `operator` projection). Moreover `T` becomes an implicit argument: applying `Monoid.law` to `idx` thus builds a type.

```
1 Coercion operator : law >-> Funclass.
2 Section MonoidProperties.
3 Variable R : Type.
4
5 Variable idx : R.
6 Local Notation "1" := idx.
7
8 Variable op : Monoid.law idx.
9 Local Notation "*/M" := op (at level 0).
10 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 ]`.

```
1 Lemma big_nat_recr n m F : m <= n ->
2   \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 addOn addn0.
```

This command adds the following rule to the canonical structures index:

canonical structures Index		
projection	value	solution
<code>Monoid.operator</code>	<code>addn</code>	<code>addn_monoid</code>

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.

### 6.7.3 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.

## 6.8 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

```
1 Lemma sum_odd n :
2   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?

```
1 Lemma sum_0 n :
2   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.

```
1 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:

```
1 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`:

```
1 Definition applybig {R I} (body : bigbody R I) acc :=
2   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.

```
1 Definition bigop R I idx r (body : I -> bigbody R I) :=
2   foldr (applybig \o body) idx r.
```

And a generic notation can be attached to it.

```

1 Notation "\big [ op / idx ]_ ( i <- r | P ) F" :=
2   (bigop idx r (fun i => BigBody i op P%B F)) : big_scope.

```

## 6.9 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 losing the head constant to which the overloaded notation is attached. For example the following term:

```

1   (@eq_op bool_eqType true false).

```

can be simplified (reduced) to the following one

```

1   (@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.

```

1 Arguments eq_op {_} _ _ : simpl never.
2 Eval simpl in ∀ x y : bool, x == y.
3 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.

```

1 Implicit Type l : seq nat.
2 Lemma example F l1 l2 :
3   \sum_(i <- l1 ++ l2) F i =
4   \sum_(i <- l2 ++ l1) F i.
5 Proof.
6 rewrite big_cat.
7

```

```

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. The proof concludes as follows:

```

1 by rewrite addnC -big_cat.
2 Qed.

```

## 6.10 Ad-hoc polymorphism

### 6.10.1 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 : T) := 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 Type 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`.

### 6.10.2 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 `eqTypes` are accepted.

The rest of this section introduces the general mechanism of phantom types used to trigger canonical structure resolution.





## 7. 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 than 7 form a finite subset of the integers (called `'I_7`). Integers and integers smaller than 7 are interchangeable data: if  $(n : \text{nat})$  and  $(i : 'I_7)$  one can clearly add  $n$  to  $i$ , and possibly 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 the  $\Sigma$ -type  $\Sigma_{(n:\text{nat})} n < 7$  which pairs a natural number  $n$  and proof that it is smaller than 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.

## 7.1 *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.

```
1 Structure tuple_of n T := Tuple { tval :> seq T; _ : size tval == n }.
2 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:

```
1 Lemma size_tuple T n (t : n.-tuple T) : size t = n.
2 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.

```
1 Example seq_on_tuple n (t : n.-tuple nat) :
2   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.

```
1 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.

```

1 Example just_tuple_attempt n (t : n.-tuple nat) :
2   size (rev [seq 2 * x | x <- rev t]) = size t.
3 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  $(\forall T, \text{seq } T \rightarrow \text{seq } T)$ . 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:

```

1 Notation "X (*...*)" :=
2   (let x := X in let y := _ in x) (at level 100, format "X (*...*)").
3 Notation "[LHS 'of' equation ]" :=
4   (let LHS := _ in
5     let _infer_LHS := equation : LHS = _ in LHS) (at level 4).
6 Notation "[unify X 'with' Y]" :=
7   (let unification := erefl _ : X = Y in True).

```

We can now simulate the unification problem encountered by `rewrite size_tuple`

```

1 Check ∀ T n (t : n.-tuple T),
2   let LHS := [LHS of size_tuple _] in
3   let RDX := size (rev t) in
4   [unify LHS with RDX].

```

The corresponding error message is the following one:

```

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
<code>tval N A</code>	<code>rev A S</code>	<code>rev_tuple N A T</code>	$T \leftarrow (\text{tval } N \text{ A}, S)$
<code>tval N B</code>	<code>map A B F S</code>	<code>map_tuple N A B F T</code>	$T \leftarrow (\text{tval } N \text{ A}, S)$
<code>tval N.+1 A</code>	<code>X :: S</code>	<code>cons_tuple N A T X</code>	$T \leftarrow (\text{tval } N \text{ A}, S)$

Thanks to the now extended capability of type inference, we can prove our lemma by just reasoning about tuples.

```

1 Example just_tuple n (t : n.-tuple nat) :
2   size (rev [seq 2 * x | x <- rev t]) = size t.
3 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.

## 7.2 *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.

```
1 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

```
1 Lemma eqtupleP n (T : eqType) : Equality.axiom (@tcmp n T).
2 Proof.
3 move=> x y; apply: (iffP eqP); last first.
4   by move=> ->.
5 case: x; case: y => s1 p1 s2 p2 /= E.
6 rewrite E in p2 *.
7 by rewrite (eq_irrelevance p1 p2).
8 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 7.2.2.

```
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
```

```
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  $\forall t : 3\text{-tuple nat}, [:: t] == [:::].$ 
2 Check  $\forall t : 3\text{-tuple bool}, \text{uniq } [:: t; t].$ 
3 Check  $\forall t : 3\text{-tuple } (7\text{-tuple nat}), \text{undup\_uniq } [:: t; t].$ 

```

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 := [subType for tval].
2 Definition tuple_eqMixin := [eqMixin of n.-tuple T by <:].
3 Canonical tuple_eqType := 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. Note that `hnf` is an evaluation strategy that focuses on the head symbol of the expression and that ignores its sub terms, see [Coq, section 5.3.7, “Performing computations”]. The following section details the implementation of the sub-type kit.

### 7.2.1 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 overloaded injectivity lemma `val_inj` saying that two objects equal in `T` are also equal in `ST`.

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* [Ch14, 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.<sup>1</sup>

Both `Sub` and `inSub` expect the typing context to specify the sub-type.

Here is a few example uses, using `tuple`:

```

1 Variables (s : seq nat) (t : 3.-tuple nat).
2 Hypothesis size3s : size s == 3.
3 Let t1 : 3.-tuple nat := Sub s size3s.

```

<sup>1</sup>The equations describing the computation of `inSub` are called `inSubT` and `inSubF`.

```

4 Let t2 := if insub s is Some t then val (t : 3.-tuple nat) else nil.
5 Let t3 := insubd t s. (* : 3.-tuple nat *)

```

We put `insub` to good use in section 7.4 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:

```

1 Section SubTypeKit.
2 Variables (T : Type) (P : pred T).
3
4 Structure subType : Type := SubType {
5   sub_sort :> Type;
6   val : sub_sort -> T;
7   Sub : ∀ x, P x -> sub_sort;
8   (* elimination rule for sub_sort *)
9   _ : ∀ K (_ : ∀ x Px, K (@Sub x Px)) u, K u;
10  _ : ∀ x Px, val (@Sub x Px) = x
11 }.

```

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.

```

1 Notation "[ 'subType' 'for' v ]" := (SubType _ v _
2   (fun K K_S u => let (x, Px) as u return K u := u in K_S x Px)
3   (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`” annotation, see [Coq, 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.

### 7.2.2 A note on boolean $\Sigma$ -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 [Hed98] 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.

```
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.

**R** It is convenient to define new types as sub-types of existing ones, since they inherit all the theory.

### 7.3 Finite types and their theory

Before describing other sub-types, we introduce the interface of types equipped with a finite enumeration.

```
1 Notation count_mem x := (count [pred y | y == x]).
2 Module finite.
3 Definition axiom (T : eqType) (e : seq T) :=
4   ∀ x : T, count_mem x e = 1.
5
6 Record mixin_of (T : eqType) := Mixin {
7   enum : seq T;
8   _ : axiom T enum;
9 }
10 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:

```
1 Definition mytype_finMixin := Finite.Mixin mytype_enum mytype_enumP.
2 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.

```
1 Lemma myenum_uniq : uniq myenum.
2 Lemma mem_myenum : ∀ x : T, x \in myenum.
3 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]`.



```

1 Lemma cardT (T : finType) : #|T| = size (enum T).
2 Lemma forallP (T : finType) (P : pred T) : reflect (∀ x, P x) [∀ x, P x].

```

Given that  $[\forall x, P x]$  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  $(\backslash\text{sum\_}(i : T) F)$  are used to express the iteration over the inhabitants of the finite type  $T$ .

## 7.4 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:

```

1 Inductive ordinal (n : nat) : Type := Ordinal m of m < n.
2 Notation "'I_' n" := (ordinal n)
3
4 Coercion nat_of_ord i := let: @Ordinal m _ := i in m.
5
6 Canonical ordinal_subType := [subType for nat_of_ord].
7 Definition ordinal_eqMixin := Eval hnf in [eqMixin of ordinal by <:].
8 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.

```

1 Definition ord_enum n : seq (ordinal n) := pmap insub (iota 0 n).

```

The `iota` function produces the sequence  $[:: 0; 1; \dots; n.-1]$ . 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  $(m \leq n)$ . What we are left to show is that such an enumeration is complete and non-redundant.

```

1 Lemma val_ord_enum : map val ord_enum = iota 0 n.
2 Proof.
3 rewrite pmap_filter; last exact: insubK.
4 by apply/all_filterP; apply/allP=> i; rewrite mem_iota isSome_insub.
5 Qed.

```

```

6
7 Lemma ord_enum_uniq : uniq ord_enum.
8 Proof. by rewrite pmap_sub_uniq ?iota_uniq. Qed.
9
10 Lemma mem_ord_enum i : i \in ord_enum.
11 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`.

```

1 Definition ordinal_finMixin n :=
2   Eval hnf in UniqFinMixin (ord_enum_uniq n) (mem_ord_enum n).
3 Canonical ordinal_finType n :=
4   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.

```

1 Lemma tnth_default T n (t : n.-tuple T) : 'I_n -> T.
2 Proof. by rewrite -(size_tuple t); case: (tval t) => [|/] []. Qed.
3
4 Definition tnth T n (t : n.-tuple T) (i : 'I_n) : T :=
5   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.

```

1 Definition enum_rank (T : finType) : T -> 'I_#|T|.

```

## 7.5 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 [Coh+15], 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<sup>2</sup> of the type of finite functions in the Mathematical Components library uses this remark, plus the tricks explained in section 6.10.2

```

1 Section FinFunDef.
2 Variable (aT : finType). (* domain type *)
3 Variable (rT : Type). (* codomain type *)
4
5 Inductive finfun_type : Type := Finfun of #|aT|. -tuple rT.
6 Definition finfun_of of phant (aT -> rT) := finfun_type.
7 Definition fgraph f := let: Finfun t := f in t.
8
9 Canonical finfun_subType := Eval hnf in [newType for fgraph].
10
11 End FinFunDef.
12
13 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.

```

1 Definition fun_of_fin aT rT f x := tnth (@fgraph aT rT f) (enum_rank x).
2 Coercion fun_of_fin : finfun >-> FunClass.
3 Definition finfun aT rT f := @Finfun aT rT (codom_tuple f).
4 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.

```

1 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`.

```

1 Definition finfun_eqMixin aT (rT : eqType) :=
2   Eval hnf in [eqMixin of finfun aT rT by <:].
3 Canonical finfun_eqType :=
4   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. The `all_words` program takes in input a length `n` and an

<sup>2</sup>Since Mathematical Components version 1.9 the definition of finite functions changed. The new implementation is based on a specific list-like data type indexed over the list of the elements of the domain. Thanks to this more sophisticated encoding, the current Coq type checker accepts, for example, the following declaration of a 3-branch tree: `Inductive tree3 := Leaf of nat | Node of {ffun 'I_3 -> tree3}`. At the time of writing Coq rejects the declaration above with the definition of finite functions given in this section since the so called “positivity checker” is too incomplete.

alphabet (a sequence of symbols (`enum T`)) and generates a list of all words of size `n` using the symbols from the alphabet.

```

1 Definition tuple_enum (T : finType) n : seq (n.-tuple T) :=
2   pmap insub (all_words n (enum T)).
3 Lemma enumP T n : Finite.axiom (tuple_enum T n).
4
5 Definition tuple_finMixin := Eval hnf in FinMixin (@FinTuple.enumP n T).
6 Canonical tuple_finType := Eval hnf in FinType (n.-tuple T) tuple_finMixin.
7
8 Definition finfun_finMixin (aT rT : finType) :=
9   [finMixin of (finfun aT rT) by <:].
10 Canonical finfun_finType aT rT :=
11   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  $\#|rT| \wedge \#|aT|$ .

```

1 Lemma card_ffun (aT rT : finType) : #| {ffun aT -> rT} | = #|rT| ^ #|aT|.

```

Also, as expected, finite functions validate extensionality.

```

1 Definition eqfun (f g : B -> A) : Prop := ∀ x, f x = g x.
2 Notation "f1 =1 f2" := (eqfun f1 f2).
3
4 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.

```

1 Lemma bigA_distr_bigA (I J : finType) F :
2   \big[*M/1]_(i : I) \big[+M/0]_(j : J) F i j
3   = \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.

$$\prod_{i \in I} \sum_{j \in J} Fij = \sum_{f \in I \rightarrow J} \prod_{i \in I} F_i(f_i)$$

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.

## 7.6 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.

```

1 Section finSetDef.
2 Variable T : finType.
3 Inductive set_type : Type := FinSet of {ffun pred T}.
4 Definition finfun_of_set A := let: FinSet f := A in f.

```

Recall that  $(\text{pred } T)$  is the type of functions from  $T$  to  $\text{bool}$ .

Using the sub-type kit we can easily transport the  $\text{eqType}$  and  $\text{finType}$  structure over finite sets.

```

1 Canonical set_subType := Eval hnf in [newType for finfun_of_set].
2 Definition set_eqMixin := Eval hnf in [eqMixin of set_type by <:].
3 Canonical set_eqType := Eval hnf in EqType set_type set_eqMixin.
4 Definition set_finMixin := [finMixin of set_type by <:].
5 Canonical set_finType := Eval hnf in FinType set_type set_finMixin.
6 End finSetDef.
7 Notation "{ 'set' T }" := (set_type T).

```

We omit again the trick to statically enforce that  $T$  is a finite type whenever we write  $\{\text{set } T\}$ , exactly as we did for finite functions. Finite sets do validate extensionality and are equipped with subset-wise and point-wise operations:

```

1 Lemma setP A B : A =i B <-> A = B.
2
3 Lemma example (T : finType) (x : T) (A : {set T}) :
4   (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  $\text{bool}$ .

Since  $T$  is finite, values of type  $\{\text{set } T\}$  admit a complement and  $\{\text{set } T\}$  is closed under power-set construction.

```

1 Lemma setCP x A : reflect (~ x \in A) (x \in ~ : A).
2 Lemma subsets_disjoint A B : (A \subset B) = [disjoint A & ~ : B].
3 Definition powerset D : {set {set T}} := [set A : {set T} | A \subset D].
4 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.

## 7.7 Permutations

Another application of finite functions is the definition of the type of permutations.

```

1 Inductive perm_of (T : finType) : Type :=
2   Perm (pval : {ffun T -> T}) & injectiveb pval.
3 Definition pval p := let: Perm f _ := p in f.
4 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  $(s : \{\text{perm } T\})$  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  $\{\text{perm } T\}$  the `eqType` and `finType` structures of `ffun T -> T`.

```

1 Canonical perm_subType := Eval hnf in [subType for pval].
2 Definition perm_eqMixin := Eval hnf in [eqMixin of perm_type by <:].
3 Canonical perm_eqType := Eval hnf in EqType perm_type perm_eqMixin.
4 Definition perm_finMixin := [finMixin of perm_type by <:].
5 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  $'I\_n$ .

```

1 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.

```

1 Definition perm_on T (S : {set T}) : pred {perm T} :=
2   fun s => [set x | s x != x] \subset S.
3 Lemma card_perm A : #|perm_on A| = #|A| ^!.

```

## 7.8 Matrix

We finally have all the bricks to define the type of matrices and provide compact formulations for their most common operations.

```

1 Inductive matrix R m n : Type := Matrix of {ffun 'I_m * 'I_n -> R}.
2 Definition mx_val A := let: Matrix g := A in g.
3 Notation "'M[' R ]_ ( m , n )" := (matrix R m n).
4 Notation "'M_' ( m , n )" := (matrix _ m n).
5 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.

```

1 Definition mxtrace R n (A : 'M[R]_n) := \sum_i A i i.
2 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  $\{\text{ffun } 'I_m * 'I_n \rightarrow R\}$  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.

```

1 Definition matrix_of_fun R m n F :=
2   Matrix [ffun ij : 'I_m * 'I_n => F ij.1 ij.2].
3 Notation "\matrix_ ( i < m , j < n ) E" :=
4   (matrix_of_fun (fun (i : 'I_m) (j : 'I_n) => E))
5 Notation "\matrix_ ( i , j ) E" := (matrix_of_fun (fun i j => E)).
6
7 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:  
 $\sum_{\sigma \in S_n} \text{sgn}(\sigma) \prod_{i=1}^n A_{i,\sigma(i)}$ .

```

1 Definition determinant n (A : 'M_n) : R :=
2   \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 L<sup>A</sup>T<sub>E</sub>X:

$$\sum_{\{\sigma \in S_n\}} \text{sgn}(\sigma) \prod_{i=1}^n A_{i, \sigma(i)}$$

Matrix multiplication deserves a few comments too.

```

1 Definition mulmx m n p (A : 'M_(m, n)) (B : 'M_(n, p)) : 'M[R]_(m, p) :=
2   \matrix_(i, k) \sum_j (A i j * B j k).
3 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 7.8.3. At the same time type inference lets one omit size information most of the time, playing once again the role of a trained reader.

### 7.8.1 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.

```
1 rewrite exchange_big /=.
2 by do 2!apply: eq_bigr => ? _; apply: mulrC.
3 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.



### 7.8.2 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  $(n_1 + n_2)$ , then the upper left block is a square matrix of size  $n_1$ .

```

1 Definition lsubmx (A : 'M_(m, n1 + n2)) : 'M_(m, n1)
2 Definition usubmx (A : 'M_(m1 + m2, n)) : 'M_(m1, n)
3 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.

```

1 Definition row_mx (A1 : 'M_(m, n1)) (A2 : 'M_(m, n2)) : 'M_(m, n1 + n2)
2 Definition col_mx (A1 : 'M_(m1, n)) (A2 : 'M_(m2, n)) : 'M_(m1 + m2, n)
3 Definition block_mx Aul Aur Adl Adu : 'M_(m1 + m2, n1 + n2)

```

The interested reader can find in [Gar+09] 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  $n - 1$ .

### 7.8.3 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:

```

1 Section SizeCast.
2 Variables (n n1 n2 n3 m m1 m2 m3 : nat).
3
4 Lemma row_mxA (A1 : 'M_(m, n1)) (A2 : 'M_(m, n2)) (A3 : 'M_(m, n3)) :
5   row_mx A1 (row_mx A2 A3) = row_mx (row_mx A1 A2) A3.

```

Observe that the left hand side has type  $'M_(m, n_1 + (n_2 + n_3))$  while the right hand side has type  $'M_(m, (n_1 + n_2) + n_3)$ . The `castmx` operator, and all its companion lemmas, let one deal with this inconvenience.

```

1 Lemma row_mxA (A1 : 'M_(m, n1)) (A2 : 'M_(m, n2)) (A3 : 'M_(m, n3)) :
2   let cast := (erefl m, esym (addnA n1 n2 n3)) in
3   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  $(n_1 + (n_2 + n_3) = (n_1 + n_2) + n_3)$ .

Lemmas like the following two let one insert or remove additional casts.

```

1 Lemma castmxKV (eq_m : m1 = m2) (eq_n : n1 = n2) :
2   cancel (castmx (esym eq_m, esym eq_n)) (castmx (eq_m, eq_n)).
3 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.

```

1 Definition conform_mx (B : 'M_(m1, n1)) (A : 'M_(m, n)) :=
2   match m =P m1, n =P n1 with
3   | ReflectT eq_m, ReflectT eq_n => castmx (eq_m, eq_n) A
4   | _, _ => B
5   end.

```

Remember that the notation  $(m =P m1)$  stands for  $(@eqP \text{ nat\_eqType } m \ m1)$ , 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.

```

1 Lemma conform_mx_id (B A : 'M_(m, n)) : conform_mx B A = A.
2 Lemma nonconform_mx (B : 'M_(m1, n1)) (A : 'M_(m, n)) :
3   (m != m1) || (n != n1) -> conform_mx B A = B.
4
5 Lemma conform_castmx (e_mn : (m2 = m3) * (n2 = n3))
6   (B : 'M_(m1, n1)) (A : 'M_(m2, n2)) :
7     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 5.2.1 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.



## 8. 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.

### 8.1 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 which 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 zero value, named `ord0`. 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  $(\forall m d : \text{nat}, 0 < d \rightarrow m \% d < d)$ , 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 using the constructor `ZmodMixin`, which bundles a carrier type, an internal

binary operation on this type, an inhabitant of the carrier type, and proofs that these data are endowed with the properties of a commutative group:

```

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`.

**R** 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 needs 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 is important as the value inferred by COQ could be different. Indeed line 2 performs some nontrivial inference, because `zmodType` is not at the bottom of the hierarchy. In particular `zmodType` derives from `eqType`, so that the `==` test can be used on all `zmodType` elements. The `ZmodType` constructor thus infers a parent structure instance from its first argument, and 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 8.4 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_add1 _)
6     Zp_nontrivial.
7 Canonical Zp_ringType := RingType 'I_p Zp_ringMixin.
8 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 like the one at line 4 is needed to define `Zp_comRingType`, as 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`.

```

1 Canonical Zp_finRingType := [finRingType of 'I_p].

```

## 8.2 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* [Bru91], 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 6.7.2, structures can be used to associate properties to any logical object, not just types, and the `Monoid.law` structure introduced in section 6.7.2 is part of a telescope hierarchy. Recall that `Monoid.law` associates an identity element and `Monoid` axioms to a binary operator:

```

1 Module Monoid.
2 Variables (T : Type) (idm : T).
3 Structure law := Law {
4   operator : T -> T -> T;
5   _ : associative operator;
6   _ : left_id idm operator;
7   _ : right_id idm operator
8 }.
9 Coercion operator : law >-> Funclass.

```

The `Coercion` declaration helps writing generic `bigop` simplification rules such as

```

1 big1_eq R (idx : R) (op : Monoid.law idx) I r (P : pred I) :
2   \big[op/idx]_(i <- r | P i) idx = idx

```

Because the `Monoid` structure is at the bottom of the hierarchy, 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`

```

1 Canonical andb_monoid := Law andbA andTb andbT.

```

This declaration makes it possible to use `big1_eq` to simplify the “big and” expression

```

1 \big[and/true]_(i in A) true

```

to just `true`,<sup>1</sup> 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

```

1 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.

```

1 Structure com_law := ComLaw {
2   com_operator : law;
3   _ : commutative com_operator
4 }.
5 Coercion com_operator : com_law >-> law.

```

At this stage, one can declare the canonical instance:

```

1 Canonical andb_comoid := ComLaw andbC.

```

and then use `pair_bigA` to factor nested “big ands” such as

<sup>1</sup>There is no spurious `add_monoid` because the identity element is a manifest field stored in the structure type.

```
1 \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 `pair_bigA` really stands for `operator (com_operator op)`. Thus applying `pair_bigA` 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

```
1 Structure add_law (mul : T -> T -> T) := AddLaw {
2   add_operator : com_law;
3   _ : left_distributive mul add_operator;
4   _ : right_distributive mul add_operator
5 }.
6 Coercion add_operator : add_law >-> com_law.
```

The instance declaration

```
1 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

```
1 bigA_distr_bigA :
2   \big[times/one]_(i : I) \big[plus/zero]_(j : J) F i j
3   = \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 a 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.

### 8.3 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 8.1), 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” which 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 6.4 while commenting on the formalisation of the `eqType` structure, which we recall here:

```

1  Module Equality.
2
3  Definition axiom T (e : rel T) := ∀ x y, reflect (x = y) (e x y).
4
5  Record mixin_of T := Mixin {op : rel T; _ : axiom op}.
6  Notation class_of := mixin_of.
7
8  Structure type := Pack {sort :> Type; class : class_of sort}.
9  ...
10 Module Exports.
11 Coercion sort : type >-> SortClass.
12 Notation eqType := type.
13 ...
14 End Equality.
15 Export Equality.Exports.

```

The `Exports` submodule, which we had omitted in section 6.4, 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 which provides effective choice for decidable predicates:

```

1  Module Choice.
2
3  Record mixin_of T := Mixin {
4    find : pred T -> nat -> option T;
5    _ : ∀ P n x, find P n = Some x -> P x;
6    _ : ∀ P : pred T, (exists x, P x) -> exists n, find P n;
7    _ : ∀ P Q : pred T, P =1 Q -> find P =1 find Q
8  }.
9
10 Record class_of T :=
11   Class {base : Equality.class_of T; mixin : mixin_of T}.
12
13 Structure type := Pack {sort; _ : class_of sort}.

```

The `mixin` provides a specific, depth-based, strategy, the `find` operator, for searching a witness. The second argument of this operator, of type `nat` acts as a “fuel” auxiliary argument, and bounds the depth of the search. But the main operation provided by the `choiceType` structure `T` is a choice function for decidable predicates (`choose : pred T -> T -> T`) satisfying the following properties:

```

1 Lemma chooseP P x0 : P x0 -> P (choose P x0).
2 Lemma choose_id P x0 y0 : P x0 -> P y0 -> choose P x0 = choose P y0.
3 Lemma eq_choose P Q : P =1 Q -> choose P =1 choose Q.

```

The `choose` operation can be defined using the `find` operator provided by the interface, and its properties follow from the axiom of countable choice. Indeed, although the choice axiom is not provable in general in Calculus of Inductive Constructions, its restriction to countable type can be proved. which is derivable in Calculus of Inductive Constructions.

```

1 Lemma find_ex_minn (P : pred nat) :
2   (exists n, P n) -> {m | P m & ∀ n, P n -> n >= m}.

```

Requiring a `find` operator is stronger than requiring a `choose` operator: this strengthened requirement makes it possible to compose instances of `choiceType`, so that the type of pairs of instances of `choiceType`, or the type of sequences of a `choiceType` are themselves instances of `choiceType`. 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`, as 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`:

```

1 Coercion base : class_of >-> Equality.class_of.
2 Coercion mixin : class_of >-> mixin_of.
3 Coercion sort : type >-> Sortclass.
4 Variables (T : Type) (cT : type).
5 Definition class := let: @Pack _ c as cT' := cT return class_of cT' in c.
6 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 ≡ sort cT` and `class` equal to `base class`. The actual link between `choiceType` and `eqType` is established by the following two lines:

```

1 Coercion Exports.eqType : type >-> Equality.type.
2 Canonical Exports.eqType.

```

Line 1 merely lets us explicitly use a `choiceType` where an `eqType` is expected, which is rare as structures are almost always 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 lower 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 do not 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,<sup>2</sup> 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 instances of `choiceType`. On the other hand, postulating that any type can be endowed with a structure of `choiceType` is a consistent extension to the Calculus of Inductive Constructions.

Similarly to the definition of `eq_op` in `eqtype`, the operations afforded by `zmodType` are defined just after the `Zmodule` module.

```

1  Definition zero V := Zmodule.zero (Zmodule.class V).
2  Definition opp V := Zmodule.opp (Zmodule.class V).
3  Definition add V := Zmodule.add (Zmodule.class V).
4  Notation "+%R" := (@add V).

```

These are defined inside the `GRing` module which encloses most of `ssralg`. They are 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.

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.

```

1  Module Ring.
2
3  Record mixin_of (R : zmodType) : Type := Mixin {
4    one : R;
5    mul : R -> R -> R;
6    _ : associative mul;
7    _ : left_id one mul;
8    _ : right_id one mul;
9    _ : left_distributive mul +%R;
10   _ : right_distributive mul +%R;
11   _ : one != 0
12 }.
13
14 Record class_of (R : Type) : Type := Class {
15   base : Zmodule.class_of R;
16   mixin : mixin_of (Zmodule.Pack base)
17 }.
18
19 Structure type := Pack {sort; _ : class_of sort}.

```

<sup>2</sup>Identities are said to be strict when they are stated using the equality predicate, unlike other equivalence relations such as the ones induced by a quotient.

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.

```
1 Definition eqType := Equality.Pack class.
2 Definition choiceType := Choice.Pack class.
3 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:  $2 \times 2$  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 respective definitions 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.

```
1 Module ComUnitRing.
2
3 Record class_of (R : Type) : Type := Class {
4   base : ComRing.class_of R;
5   mixin : UnitRing.mixin_of (Ring.Pack base)
6 }.
7
8 Structure type := Pack {sort; _ : class_of sort}.
```

The two structures that are compounded this way can be declared indifferently as `base` or as `mixin`. 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  $?_R : \text{comUnitRingType}$  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 [MT13, 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, `unitRingType` 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 `ringgroup` does this), or to add explicit “join” links to a telescope hierarchy (`ssralg` does this for its algebraic predicate hierarchy).

## 8.4 Parameters and constructors

We have noted already that instances of structures are often hard to provide explicitly, because they are in fact expected to be constructed automatically via inference, rather than being provided explicitly by the user. For example the explicit `ringType` structure for `int * rat` is

```
1 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 6.10.2 how to do this, using the `phantom` type

```
1 Inductive phantom T (p : T) := Phantom.
2 Arguments phantom : clear implicits.
```

Here we can use a simpler type, equivalent to `phantom Type`

```
1 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`.

```
1 Module Lmodule.
2
3 Variable R : ringType.
4
5 Record mixin_of (R : ringType) (V : zmodType) := Mixin {
6   scale : R -> V -> V;
7   ...}.
8
9 Record class_of V := Class {
```

```

10   base : Zmodule.class_of V;
11   mixin : mixin_of R (Zmodule.Pack base)
12 }.
13
14 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`

```

1 Notation lmodType R := (type (Phant R)).

```

allows us to write `(V : lmodType (int * rat))`. Indeed, type inference fills in the unsightly expression `(pair_ringType int_ringType rat_ringType)` for `R`. This elaboration happens when solving the unification problem triggered by the need to unify `(Phant ?)` with `(int * rat)`.

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:

```

1 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<sup>3</sup>, following the introduction of section variables `T` and `cT`, and the definition of `class`:

```

1 Definition clone c & phant_id class c := @Pack T c.

```

Recall that with `SSREFLECT` loaded, `& T` introduces an anonymous parameter of type `T`. As with `lmodType` above we use `Notation` to supply the identity function for this dummy functional parameter

```

1 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.

More precisely, 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.class_of (Choice.sort ?cT)) (Choice.class ?cT))`, which it will then unify with `(phantom (Choice.class_of 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, because it only extends `eqType` with a mixin that does not depend on `eqType`. Note that this definition allows `COQ` to infer `T` from `m`.

<sup>3</sup>More precisely, the situation is slightly different in the case of structure with a parameter, like modules.

```

1 Definition pack T m :=
2   fun bT b & phant_id (Equality.class bT) b => Pack (@Class T b m).
3 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.

```

1 Definition pack T b0 (m0 : mixin_of (@Zmodule.Pack T b0)) :=
2   fun bT b & phant_id (Zmodule.class bT) b =>
3     fun m & phant_id m0 m => Pack (@Class T b m).
4 Notation RingType T m := (@pack T _ m _ _ id _ id).

```

Type-checking the second `id` will set `m` to `m0` after checking that the inferred base class `b`  $\equiv$  `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`.

```

1 Definition pack T :=
2   fun bT b & phant_id (ComRing.class bT) (b : ComRing.class_of T) =>
3     fun mT m & phant_id (UnitRing.class mT) (@UnitRing.Class T b m) =>
4       Pack (@Class T b m).
5 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.

```

1 Definition pack T b0 mul0 (axT: @axiom R (@Lmodule.Pack R _ T b0 T) mul0) :=
2   fun bT b & phant_id (Ring.class bT) (b : Ring.class_of T) =>
3     fun mT m & phant_id (@Lmodule.class R phR mT) (@Lmodule.Class R T b m) =>
4     fun ax & phant_id axT ax =>
5     Pack (Phant R) (@Class T b m ax) T.
6 ...
7 Notation LalgType R T a := (@pack _ (Phant R) T _ _ a _ _ id _ _ id _ id).

```

The interested reader can also refer to [MT13, section 7] for a description of this technique.

## 8.5 Linking a custom data type to the library

The sub-type kit of chapter 7 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.

```

1 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.

```

1 Definition w2o (w : windrose) : 'I_4 :=
2   match w with
3   | N => inord 0 | S => inord 1 | E => inord 2 | W => inord 3
4   end.

```

More generally, the `inord` constructor lets us postpone the (trivial by computation) proofs that 0, 1, 2, 3 are smaller than 4.

We provide only an artificially partial function, so as to fit the signature expected by the choice mixin, which is more liberal than needed here.

```

1 Definition o2w (o : 'I_4) : option windrose :=
2   match val o with
3   | 0 => Some N | 1 => Some S | 2 => Some E | 3 => Some W
4   | _ => None
5   end.

```

Then we can show that these two functions cancel out.

```

1 Lemma pcan_wo4 : pcancel w2o o2w.
2 Proof. by case; rewrite /o2w /= inordK. Qed.

```

Now, thanks to the `PcanXXMixin` family of lemmas, one can inherit on `windrose` the structures of ordinals.

```

1 Definition windrose_eqMixin := PcanEqMixin pcan_wo4.
2 Canonical windrose_eqType := EqType windrose windrose_eqMixin.
3 Definition windrose_choiceMixin := PcanChoiceMixin pcan_wo4.
4 Canonical windrose_choiceType := ChoiceType windrose windrose_choiceMixin.
5 Definition windrose_countMixin := PcanCountMixin pcan_wo4.
6 Canonical windrose_countType := CountType windrose windrose_countMixin.
7 Definition windrose_finMixin := PcanFinMixin pcan_wo4.
8 Canonical windrose_finType := FinType windrose windrose_finMixin.

```

Only one tiny detail has been left on the side so far. To use `windrose` in conjunction with the `\in` infix notation or with the notation `#|...|` for cardinality, the type declaration has to be tagged as an instance of the `predArgType` structure, for types which model a predicate, as it is the one which supports the latter notations. It can be done as follows.

```

1 Inductive windrose : predArgType := N | S | E | W.

```

After that, our new data type can be used exactly as the ordinal one can.

```

1 Check (N != S) && (N \in windrose) && (#| windrose | == 4).

```

A generic technique is available in order to equip a data type with structures of `eqType`, `choiceType`, and `countType`. It consists in providing a correspondence with the generic tree data type (`GenTree.tree T`): an  $n$ -ary tree with nodes labelled with natural numbers and leaves carrying a value in  $T$ .







# Cheat Sheet

## Terminology

Context  $\left\{ \begin{array}{l} x : T \\ S : \{\text{set } T\} \\ xS : x \text{ \textit{in} } S \end{array} \right.$

The bar  $\text{=====}$

Goal  $\left\{ \begin{array}{l} \forall y, y == x \rightarrow y \text{ \textit{in} } S \\ \text{Assumptions} \\ \text{Conclusion} \end{array} \right.$

**Top** is the first assumption, **y** here  
**Stack** alternative name for the list of *Assumptions*

## Popping from the stack

Note: in the following we assume *cmd* does nothing, exactly like *move*.

**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,
P x -> Q x -> G
----->
x : T
px : P x
-----
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 them **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

```

=====
∀ s : seq nat,
0 < size s -> P s
----->
x : nat
xs : seq nat
=====
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**

```

=====
A && B -> C -> D
----->
pa : A
pb : B
=====
C -> D

```

**cmd** => /= {px}

Run *cmd* then simplify the goal then discard **px** from then context

```

x : nat
px : P x
-----
true && Q x -> R x
----->
x : nat
-----
Q x -> R 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
-----
(exists2 y,
x = y & Q y) -> P x
----->
y : nat
-----
Q y -> P y

```

**cmd** => /(\_ x) h

Introduce **h** specialized to **x**

```

P : nat -> Prop
x : nat
-----
(∀ n, P n) -> G
----->
P : nat -> Prop
x : nat
h : P x
=====
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
px : P x
-----
Q x y
----->
x : nat
px : P x
-----
∀ 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
-----
P x
----->
x : nat
-----
∀ 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 < x.+1
----->
x : nat
=====
  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
Exc : ∀ x, x = c
=====
P a
----->
Eab : a = b
Exc : ∀ x, x = c
=====
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
=====
P b
----->
Eab : a = b
=====
P c
-----

```

```
rewrite /(_ && _).
```

Unfold the definition of `&&`

```

a : bool
=====
a && a = a
----->
a : bool
=====
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
c := b + 3 : nat
=====
true && (a == b + 3)
----->
a, b : nat
c := b + 3 : nat
=====
0 + a == c
-----

```

```
apply: H.
```

Apply H to the current goal

```

H : A -> B
=====
B
----->
=====
A
-----

```

```
case: ab.
```

Eliminate the conjunction or disjunction

```

ab : A /\ B
=====
G
----->
=====
A -> B -> G
-----

```

```
case: (leqP a b).
```

Reason by cases using the leqP spec lemma

```

a, b : nat
=====
a <= b && b > a
----->
a, b : nat
=====
true && false
-----

a, b : nat
=====
false && true
-----

```

```
elim: s.
```

Perform an induction on s

```

s : seq nat
=====
P s
----->
=====
P [::]
-----

=====
∀ n s,
P s -> P (n :: s)
-----

```

```
elim/last_ind: s
```

Start an induction on s using the induction principle last\_ind

```

s : seq nat
=====
P s
----->
=====
P [::]
-----

=====
∀ s x,
P s ->
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
=====
G
-----
→
a : T
=====
P a
```

```
a : T
pa : 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
b : bool
=====
reflect P b
-----
→
```

```
P : Prop
b : bool
=====
b -> P
```

**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
b2 : bool
=====
b1 = ~~ b2
-----
→
```

```
b1 : bool
b2 : bool
=====
b1 -> ~ b2
```

```
b1 : bool
b2 : bool
=====
~ b2 -> b1
```

**rewrite**: (eqP Eab)

rewrite with the boolean equality Eab

```
Eab : a == b
=====
P a
-----
→
Eab : a == b
=====
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 (1tnSn n) => {n} // n

General induction over n, note that the first goal has a false assumption  $\forall n, n < 0 \rightarrow \dots$  and is thus solved by //

```
n : nat
=====
P n
-----
→
n : nat
=====
(∀ 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`

`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 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 + b) + (a + b) =
(a + b) + (0 + a + b)
```

→

```
n := a + b
=====
(a + b) + n =
(a + b) + n
```

# IW Indexes and Bibliography

<b>Concepts</b> .....	<b>177</b>
<b>Ssreflect Tactics</b> .....	<b>179</b>
<b>Definitions and Notations</b> .....	<b>181</b>
<b>Coq Commands</b> .....	<b>183</b>
<b>Bibliography</b> .....	<b>185</b>







## Concepts

- abbreviation, 37
- Abstracting variables, 34
- abstraction, 14
  
- binder, 14, 19
  
- case analysis, 46
  - naming, 48
- coercion, 112
- computation, 16, 24, 45
  - symbolic, 34
- consistency, 81
- convertibility, 75
- currying of functions, 16
  
- dependent function space, 73, 74
  
- equality, 41, 79
  
- formal proof, 45
- forward reasoning, 89, 90
- function application, 14
- functional extensionality, 146
  
- general term, *see also* higher-order, 60
- goal stack model, 84
- ground equality, 41
  
- higher-order, 17, 60
  
- identity, 43
- implicit argument, 29
- improving **by**  $\square$ , 56
- induction, 60
  - generalizing, 62
- inductive type, 20
  - constructor, 20
- injectivity of constructors, 76
  
- keyed matching, 65
  
- list comprehension, 30
  
- machine checked proof, 46
- matching algorithm, 64
  
- natural number, 21
- notation, 37
- notation scope, 32
  
- partial application, 17
- pattern, 51
- pattern matching, 23, 76
  - exhaustiveness, 23
  - irrefutable, 33
- polymorphism, 28
- positivity, 81
- proof irrelevance, 141, 143
- proposition, 41

provide choiceType structure, 167  
provide eqType structure, 167  
provide finType structure, 167

record, 32

recursion, 24, 30

    termination, 25

reflection view, 100

reordering goals, 49

rewrite rule, 52

rewriting, 51

search, 66

section, 33

simplifying equation, 110

symmetric argument, 90

terminating tactic, 46

termination, 25, 81

type, 15

type error, 15

typing an application, 17

unfolding equation, 110

view, 85

well typed, 15



## Ssreflect Tactics

```
;[... ..]102
apply:, 56
by [], 45
case: name => [m], 48
case: name, 47
elim: name => [m IHm], 61
gen have name : name / type, 152
have name : type by tactic, 89
have name : type, 89
last first, 49
rewrite, 51
  /= (simplify close), 63
  // (close trivial goals), 59
  /= (simplification), 63
  ! (iteration), 52
  -/ (folding), 65
  -[term]/(term) (changing), 66
  - (right-to-left), 53
  / (unfolding), 64
  ? (optional iteration), 63
  [in RHS] (focusing), 63
  {name} (clear), 66
suff also suffices, 92
tactic : .., 87
  : {name} (disposal), 88
  : term (generalization), 88
  name: term (equation), 89
tactic => .., 84
  => -> (rewriting L2R), 87
  => /(_ arg) (specialization), 87
  => // (close trivial goals), 85
  => /= (simplification), 85
  => /view/view (many views), 86
  => /view (view application), 85
  => <- (rewriting R2L), 87
  => [ ..| .. ] (case), 84
tactic in name, 110
wlog also without loss, 93
tactic => ..
  => {name} (disposal), 85
```





## Definitions and Notations

`(_ * _) (pair)`, 31  
`(_ ++ _)`, 31  
`(_ , _)`, 31  
`(_ .*2)`, 26  
`(_ .+1)`, *see also* `s`  
`(_ .-1)`, 26  
`(_ :: _)`, 29  
`(_ =1 _)`, 148  
`(_ =P _)`, 128, *see also* `eqP`  
`.. <= .. < ..`, 38  
`.. <= .. <= ..`, 38  
`.. <= ..`, 38  
`AddLaw`, 160  
`ComLaw`, 159  
`EqMixin`, 156  
`EqType`, 156  
`Equality.sort`, 156  
`Equality.type`, 156  
`False`, 79  
`GenTree.tree`, 168  
`I`, 79  
`Law`, 159  
`0`, 21  
`PcanEqMixin`, 156  
`s`, 21  
`True`, 79  
`[&& .. , .. & ..]`, 30  
`[:: .. , .. & ..]`, 29  
`[==> .. , .. => ..]`, 30  
`[eqMixin of .. by <:]`, 156  
`[eqType of ..]`, 156  
`[pred x : T , E]`126  
`[rel x y : T , E]`126  
`[seq .. ; ..]`, 29  
`[seq .. <- .. | ..]`, 31  
`[seq .. | .. <- ..]`, 31  
`[| | .. , .. | ..]`, 30  
`\big[../..]_(..|..) ..`, 130  
`\sum`, 36, 130  
`_`, 24  
`'`, 89  
`addSn`, 46  
`addn`, 24, 25, 35  
`add`, 35  
`andP`, 102  
`andb`, 21  
`associative`, 55  
`block_mx`, 153  
`cancel`, 55  
`castmxKV`, 153  
`castmx_id`, 153  
`castmx`, 153  
`choiceType`, 161  
`col_mx`, 153  
`comRingType`, 164  
`com_unitRingType`, 164

commutative, 54  
 conform\_mx, 154  
 conj, 78  
 contraLR, 57  
 elimTF, 104  
 eqP, 125  
 eqType, 156, 161  
 eqnP, 102  
 erefl, 79  
 ex\_intro, 78  
 exists2, 89  
 fix, 110  
 foldr, 33  
 forall, 28  
 fun .. => .., 14  
 idP, 103  
 if .. then .. else .., 21  
 ifP, 108  
 iffP, 102  
 implyP, 102  
 injective, 55  
 iota, 36  
 isT, 75  
 iter, 33  
 left\_distributive, 55  
 left\_id, 55  
 leqP, 108  
 leqn0, 48  
 leq, 37  
 let: .. := .. in .., 33  
 lsubmx, 153  
 ltngtP, 108  
 map, 30  
 muln\_eq0, 49  
 muln, 49  
 nat\_ind, 80  
 nat, 21  
 negbK, 46  
 not, 79  
 of, 145  
 option, 31  
 orP, 102  
 or\_introl, 79  
 or\_intror, 79  
 pcancel, 55  
 preArgType, 168  
 predn , *see also* .-1  
 pred (predicate), 112  
 reflect, 101, 107  
 ringType, 164  
 row\_mx, 153  
 size\_map, 54  
 size, 30  
 subn, 25  
 ubnPgeq, 109  
 ulsubmx, 153  
 unitRingType, 164  
 usubmx, 153  
 zmodType, 163



## Coq Commands

[About](#), 15  
[Admitted](#), 43  
[Arguments](#), 107  
[Check](#), 15  
[Compute](#), 16  
[Definition](#), 14  
[Fixpoint](#), 24  
[Hint Resolve](#), 56  
[Hint View](#), 104  
[Implicit Type](#), 33

[Inductive](#), 20, 77  
[Lemma](#), 42  
[Notation](#), 37  
[Print](#), 15, 17  
[Record](#), 32  
[Section](#), 33  
[Structure](#), 122  
[Theorem](#), 42  
[Variable](#), 33







## Bibliography

- [Asp+12] Andrea Asperti et al. “A Bi-Directional Refinement Algorithm for the Calculus of (Co)Inductive Constructions”. In: *Logical Methods in Computer Science* 8.1 (2012). DOI: [10.2168/LMCS-8\(1:18\)2012](https://doi.org/10.2168/LMCS-8(1:18)2012). URL: [http://dx.doi.org/10.2168/LMCS-8\(1:18\)2012](http://dx.doi.org/10.2168/LMCS-8(1:18)2012) (cited on page 119).
- [BC04] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq’Art: The Calculus of Inductive Constructions*. Springer, 2004 (cited on pages 3, 25, 77).
- [Bou04] Nicolas Bourbaki. *Theory of sets*. Elements of Mathematics (Berlin). Reprint of the 1968 English translation [Hermann, Paris; MR0237342]. Springer-Verlag, Berlin, 2004, pages viii+414. ISBN: 3-540-22525-0. DOI: [10.1007/978-3-642-59309-3](https://doi.org/10.1007/978-3-642-59309-3). URL: <http://dx.doi.org/10.1007/978-3-642-59309-3> (cited on page 69).
- [Bru91] N. G. de Bruijn. “Telescopic Mappings in Typed Lambda Calculus”. In: *Information and Computation* 91.2 (1991), pages 189–204 (cited on page 158).
- [Chl14] Adam Chlipala. *Certified Programming with Dependent Types*. The MIT Press, 2014. ISBN: 9780262026659 (cited on pages 3, 142).
- [Coh+15] Cyril Cohen et al. *Cubical Type Theory: a constructive interpretation of the univalence axiom*. Preprint. 2015 (cited on page 146).
- [Coq] Coqdev - The Coq development team. *The Coq proof assistant reference manual*. Inria. URL: <http://coq.inria.fr> (cited on pages 3, 16, 25, 29, 32, 33, 35, 37, 59, 69, 71, 75, 81, 102, 117, 142, 143).
- [CH88] Thierry Coquand and Gérard Huet. “The calculus of constructions”. In: *Information and Computation* 76.2-3 (1988), pages 95–120. ISSN: 0890-5401 (cited on page 69).

- [CP90] Thierry Coquand and Christine Paulin-Mohring. “Inductively defined types”. In: *Colog’88*. Volume 417. LNCS. Springer-Verlag, 1990 (cited on pages 69, 75).
- [Gar+09] François Garillot et al. “Packaging Mathematical Structures”. In: *Theorem Proving in Higher Order Logics*. Edited by Tobias Nipkow and Christian Urban. Volume 5674. Lecture Notes in Computer Science. Munich, Germany: Springer, 2009. URL: <https://hal.inria.fr/inria-00368403> (cited on page 153).
- [Gon08] Georges Gonthier. “Formal Proof – The Four-Color Theorem”. In: *Notices of the American Mathematical Society* 55.11 (Dec. 2008), pages 1382–1393. URL: <http://www.ams.org/notices/200811/tx081101382p.pdf> (cited on pages 3, 75).
- [GMT15] 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. URL: <https://hal.inria.fr/inria-00258384> (cited on pages 3, 50, 53, 62, 64, 87, 108, 110).
- [Gon+13a] Georges Gonthier et al. “A Machine-Checked Proof of the Odd Order Theorem”. In: *ITP 2013, 4th Conference on Interactive Theorem Proving*. Edited by Sandrine Blazy, Christine Paulin, and David Pichardie. Volume 7998. LNCS. Rennes, France: Springer, July 2013, pages 163–179. DOI: 10.1007/978-3-642-39634-2\_14. URL: <https://hal.inria.fr/hal-00816699> (cited on page 3).
- [Gon+13b] Georges Gonthier et al. “How to make ad hoc proof automation less ad hoc”. In: *J. Funct. Program.* 23.4 (2013), pages 357–401. DOI: 10.1017/S0956796813000051. URL: <http://dx.doi.org/10.1017/S0956796813000051> (cited on page 119).
- [GM05] Benjamin Gregoire and Assia Mahboubi. “Proving Equalities in a Commutative Ring Done Right in Coq”. In: *TPHOLs 2005*. Edited by Joe Hurd and Tom Melham. Volume 3603. Lecture Notes in Computer Science. Oxford, United Kingdom: Springer, Aug. 2005, pages 98–113. DOI: 10.1007/11541868\_7. URL: <https://hal.inria.fr/hal-00819484> (cited on page 75).
- [Hed98] Michael Hedberg. “A Coherence Theorem for Martin-Löf’s Type Theory”. In: *J. Funct. Program.* 8.4 (July 1998), pages 413–436. ISSN: 0956-7968. DOI: 10.1017/S0956796898003153. URL: <http://dx.doi.org/10.1017/S0956796898003153> (cited on page 143).
- [How80] William Howard. “The formulae-as-types notion of construction”. In: (1980). Edited by J. Roger Seldin and Jonathan P. Hindley, pages 479–490 (cited on page 69).
- [MT13] Assia Mahboubi and Enrico Tassi. “Canonical Structures for the working Coq user”. In: *ITP 2013, 4th Conference on Interactive Theorem Proving*. Edited by Sandrine Blazy, Christine Paulin, and David Pichardie. Volume 7998. LNCS. Rennes, France: Springer, July 2013, pages 19–34. DOI: 10.1007/978-3-642-39634-2\_5. URL: <https://hal.inria.fr/hal-00816703> (cited on pages 165, 167).
- [Mar80] Per Martin-Löf. *Intuitionistic Type Theory*. Edited by Notes by Giovanni Sambin of a series of lectures given in Padua. Bibliopolis, June 1980 (cited on page 69).

- [NGV94] Rob Nederpelt, Herman Geuvers, and Roel de Vrijer, editors. *Selected Papers on Automath*. Volume 133. Studies in Logic and the Foundations of Mathematics. North-Holland, 1994 (cited on page 83).
- [Pau93] Christine Paulin-Mohring. “Inductive Definitions in the System Coq - Rules and Properties”. In: *Proceedings of the conference Typed Lambda Calculi and Applications*. Edited by M. Bezem and J.-F. Groote. Lecture Notes in Computer Science 664. LIP research report 92-49. 1993 (cited on page 76).
- [Pau15] Christine Paulin-Mohring. “Introduction to the Calculus of Inductive Constructions”. In: *All about Proofs, Proofs for All*. Edited by Bruno Woltzenlogel Paleo and David Delahaye. Volume 55. Studies in Logic (Mathematical logic and foundations). College Publications, Jan. 2015. URL: <https://hal.inria.fr/hal-01094195> (cited on page 3).
- [Pie+15] Benjamin C. Pierce et al. *Software Foundations*. Electronic textbook, 2015 (cited on page 3).
- [Pol92] Robert Pollack. “Implicit Syntax”. In: *Informal Proceedings of First Workshop on Logical Frameworks*. 1992, pages (cited on page 115).
- [SZ14] Matthieu Sozeau and Beta Ziliani. “Towards A Better Behaved Unification Algorithm for Coq”. In: *Proceedings of The 28th International Workshop on Unification*. 2014. URL: [http://www.pps.univ-paris-diderot.fr/~sozeau/research/publications/Towards\\_A\\_Better\\_Behaved\\_Unification\\_Algorithm\\_for\\_Coq-UNIF14-130714.pdf](http://www.pps.univ-paris-diderot.fr/~sozeau/research/publications/Towards_A_Better_Behaved_Unification_Algorithm_for_Coq-UNIF14-130714.pdf) (cited on page 119).
- [Uni13] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. Institute for Advanced Study: <https://homotopytypetheory.org/book>, 2013 (cited on pages 3, 80).
- [Wei85] Gerald M. Weinberg. *The Psychology of Computer Programming*. New York, NY, USA: John Wiley & Sons, Inc., 1985. ISBN: 0442292643 (cited on page 83).