

Mathematical Component

Assia Mahboubi Enrico Tassi

with contributions by
Yves Bertot Georges Gonthier

August 24, 2016
version: draft 0

Chapter 0

Introduction

Mathematical Components is the name of a library of formalized mathematics for the Coq system. It covers a variety of topics, from the theory of basic data structures (e.g., numbers, lists, finite sets) to advanced results in various flavors of algebra. This library constitutes the infrastructure for the machine checked proofs of the Four Color Theorem [8] and of the Odd Order Theorem [9].

The reason of existence of this book is to break down the barriers to entry. While there are several books around covering the usage of the Coq system [3, 19, 4] and the theory it is based on [22, Chapter 4][18, 23], 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 one hand newcomers, even the more mathematical 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 this book pretends to be a complete description of Coq or Ssreflect: both tools already come with a comprehensive user manual [22, 10]. However, the reader is quickly put in the middle of a large library of formalized concepts and given sufficient tools to prove non-trivial mathematical results by reusing parts of the library. For example around page one-hundred the reader learns how to show, formally, the infinitude of prime numbers, or the correctness of the Euclidean's division algorithm, in a few lines of proof text.

0.1 Structure of the book

The book is divided into three parts. The first two parts of the book are conceived to be read linearly, while the third one can be read in any order.

0.1.1 Part 1: Definitions and proofs

This part introduces two languages and a formalization approach.

The first language is *Gallina*, the mathematical foundation of the COQ system. For the expert reader: the type theory called Calculus of Inductive Constructions. Such language is both a programming and a specification language: it is used to declare data, define programs, specify their behavior and represent their correctness proofs. Its most intriguing features are two: a rich system of types, that can depend on other types or even values, and the notion of computation, that is considered a first class citizen, i.e. a true form of proof.

The second language, Small Scale Reflection (*Ssreflect* for short), is used to write proofs. Its most characterizing features are two: it lets one write concise and stable proof scripts enabling the development of large libraries over a long time frame, and provides good support for the formalization approach the Mathematical Components library is based on, from which its name is also taken.

The *Small Scale Reflection* approach was initially conceived for the formal proof of the Four Colors Theorem, and it later served as the pillar for the Mathematical Components library and the formal proof of the Odd Order Theorem. Such approach takes major advantage from the symbolic computation capabilities provided by Gallina.

The COQ system provides a rich platform in which all this is put together. COQ implements the Gallina language, and a checker for it. It also provides a platform for the Ssreflect proof language and many other facilities that are crucial for the Mathematical Components library like type inference.

0.1.2 Part 2: Formalization techniques

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

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

Formalized knowledge is organized by means of interfaces (in the spirit of algebraic structures) and relations between them. Type inference is programmed to play the role of an 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 prop-

erty. Type inference is programmed to transport all the theory available on the original concept to the derived one.

0.1.3 Part 3: Mathematics in Mathematical Components

This part provides a, unfinished by design, panoramic view of the Mathematical Components library. It interleaves a catalog of formalized theories with formalization choices that play a crucial role in the definition of the formal object.


0.2 Conventions

Sections that are labelled with one (★) are of medium complexity and are of interest to the reader willing to extend the Mathematical Components library. Sections that are labelled with two (★★) are of high complexity and are of interest to the reader willing to understand the technical, implementation, details of the Mathematical Components library.

Tricky details typically overlooked by beginners are signalled as follows:

Warning


Mind this detail.



Advice one should keep in mind are signalled as follows:

Advice

Remember this advice.



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

A sample snippet

```

1 Example Gauss n : \sum_(0 <= i < n.+1) i = (n * n.+1) %/ 2.
2 Proof.
3 elim: n =>[|n IHn]; first by apply: big_nat1.
4 rewrite big_nat_recr // = IHn addnC -divnMD1 //.
5 by rewrite mulnS muln1 -addnA -mulSn -mulnS.
6 Qed.
```

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

Output from Coq after line 3

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

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

Contents

0	Introduction	3
0.1	Structure of the book	3
0.1.1	Part 1: Definitions and proofs	4
0.1.2	Part 2: Formalization techniques	4
0.1.3	Part 3: Mathematics in Mathematical Components	5
0.2	Conventions	5
I	Definitions and proofs	11
1	Computational definitions	
	<i>Defining concepts by writing programs</i>	13
1.1	Functions	13
1.1.1	Higher order functions	16
1.2	Data: construction and destruction	17
1.2.1	Boolean values	17
1.2.2	Natural numbers	18
1.3	Data: recursion	21
1.3.1	Recursion for integers	22
1.4	Data: generic containers	25
1.4.1	The (polymorphic) sequence data type	26
1.4.2	Recursion for sequences	28
1.4.3	Other containers	29
1.5	The Section mechanism	30
1.6	Symbolic computations under a context	31
1.7	Iterators in mathematics	32
1.8	Notations and abbreviations	33
1.9	Aggregated data: records	34
1.10	Exercises	36
1.10.1	Solutions	37
2	Statements and proofs	
	<i>First steps in formal proofs</i>	39
2.1	Formal statements	39

2.1.1	Ground equalities	39
2.1.2	Identities	41
2.1.3	From boolean predicates to formal statements	42
2.1.4	Conditional statements	43
2.2	Formal proofs	43
2.2.1	Proofs by computation	44
2.2.2	Case analysis	45
2.2.3	Rewriting	49
2.3	Quantifiers	52
2.3.1	Universal quantification, first examples	52
2.3.2	Proving with conditional lemmas	54
2.3.3	Proofs by induction	57
2.4	Searching the library	60
2.4.1	Search by pattern	60
2.4.2	Search by name	60
2.5	Rewrite, a swiss army knife	61
2.5.1	Rewrite contextual patterns (★)	62
2.6	Exercises	65
2.6.1	Solutions	65
3	Type Theory	
	<i>The Curry-Howard correspondence</i>	67
3.1	Connectives	68
3.1.1	Primitive types and terms formers	68
3.1.2	Inductive types	69
3.1.3	Proof commands	72
3.2	Managing the proof context	72
3.2.1	The stack model	73
3.3	Inductive reasoning	78
3.3.1	Strong induction	79
3.4	On the status of Axioms	80
3.5	Terms, types, proofs: all in the same pot	81
4	Boolean reflection	
	<i>Towards real proofs</i>	83
4.1	Views	84
4.1.1	Relating statements in <code>bool</code> and <code>Prop</code>	84
4.1.2	Proving views	86
4.1.3	Using views in intro patterns	88
4.1.4	Using views with tactics	89
4.2	Advanced, practical, statements	90
4.2.1	Inductive specs with indices	90
4.3	Real proofs, finally!	93
4.3.1	Primes, a never ending story	93
4.3.2	Order and max, a matter of symmetry	95
4.3.3	Euclidean division, simple and correct	99

4.4	Notational aspects of specifications	100
4.5	Exercises	101
4.5.1	Solutions	102

II Formalization techniques 105

5 Type Inference

	<i>Teaching Coq how to read Math</i>	107
5.1	Type inference and Higher Order unification	108
5.2	Recap: type inference by examples	109
5.3	Records as relations	111
5.4	Records as first class relations	115
5.5	Records as (first class) interfaces	116
5.6	Using a generic theory	118
5.7	The generic theory of sequences	119
5.8	The generic theory of “big” operators	121
5.8.1	The generic notation for <code>foldr</code>	123
5.8.2	Assumptions of a bigop lemma	125
5.8.3	Searching the bigop library	127
5.9	Stable notations for big operators ($\star\star$)	127
5.10	Working with overloaded notations (\star)	128
5.11	Querying canonical structures ($\star\star$)	129
5.11.1	Phantom types (\star)	129
5.12	Exercises	130
5.12.1	Solutions	130

6 Sub-Types

	<i>Terms with properties</i>	131
6.1	n -tuples, lists with an invariant on the length	132
6.2	n -tuples, a sub-type of sequences	135
6.2.1	The sub-type kit (\star)	136
6.2.2	A note on boolean Σ -types	138
6.3	Finite types and their theory	138
6.4	The ordinal subtype	140
6.5	Finite functions	141
6.6	Finite sets	143
6.7	Permutations	144
6.8	Matrix	145
6.8.1	Example: matrix product commutes under trace	146
6.8.2	Block operations	147
6.8.3	Size casts (\star)	148

7 Hierarchies	
<i>Organizing knowledge</i>	151
7.1 Structure interface	152
7.2 Telescopes	155
7.3 Packed classes (★)	157
7.4 Parameters and constructors (★★)	162
7.5 Linking a custom data type to the library	164
 III Mathematics in Mathematical Components	 167
8 Numbers	169
9 Polynomials, Linear algebra	171
10 Finite group theory	173
11 Representation and Character theory	175
12 Galois Theory	177
13 Real closed fields	179
 IV Indexes	 181
Index: Concepts	183
Index: Ssreflect tactics	185
Index: Coq terms	187
Index: Coq commands	189

Part I

Definitions and proofs

Chapter 1

Computational definitions

Defining concepts by writing programs

We will take the point of view that mathematics can be understood as the study of objects, operations on these objects, and abstractions. Children start learning mathematics by counting, where the objects are numbers and the operations are enumerating and adding. In fact numbers are already the result of an abstraction process.

What are the objects, the operations, and the abstractions in a given mathematical corpus is quite fluctuant. The objects of group theory may sometimes be viewed as operations in another context. Nevertheless, in this chapter, we describe how some objects are described with the intent that they are *inert*, that is, they are meant to be manipulated by operations. We also show how operations are described.

1.1 Functions

In this section, we expect readers to get acquainted with the notion of function and the associated syntax. We rely on natural numbers and the addition operation, noted with the $+$ symbol. Both concepts will only be described fully a few pages later.

We start by considering the expression

$$2 + 1$$

We can read this expression as an object, but also as the result of an operation on an object, or as the result of an operation on two objects. Let us first consider the operation of *adding one to a natural number*. This operation is written in the following manner.

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

To write such an operation, we can start from the expression $(2 + 1)$ and isolate the object in this expression that is considered a variable component for the operation, here 2, we then replace this object with a symbolic name, here n . We then encapsulate the expression $(n + 2)$ with the prefix “`fun n =>`” to relate the result produced by the operation to the variable n . We commonly say that “`fun n =>`” binds the variable n in the expression $n + 2$.

The “`fun`” construction stands for *function*. Functions are the main tool provided by COQ to describe operations, so we will use the two terms interchangeably. Sometimes we also call operations *programs*, since the functions we talk about are effective: i.e. described by a code a computer can execute. As a consequence we borrow from computer science some jargon, like *input* or *return*. For example we say that an operation takes in input a number and returns its double to mean that the corresponding program computes the double of its input, or that the function maps a number to its double.

So we have an expression which represents the operation of adding 1. Applying this operation to the object 2 will be written this way:

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

From the point of view of mathematical syntax, there are two unusual things in this line. First, applying a function to an argument is simply by writing the function on left of the argument, *not necessarily with parentheses around the argument*; we will come back to this later. Second, this object `(fun n => n + 1)` is unusual. When defining an operation, we usually rely on a sentence *consider the function f from \mathbb{N} to \mathbb{N} which maps n to $n + 1$* . This would also be written without natural language as follows:

$$f : \begin{array}{c} \mathbb{N} \rightarrow \mathbb{N} \\ n \mapsto n + 1 \end{array} \quad (1.1)$$

The main difference is that the usual mathematical practice includes the step of giving a name, here f to the operation.

When using COQ, the step of giving a name can also be performed, in the following manner:

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

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

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

The mathematical practice when defining a function is to give information about its domain and codomain, as we did when writing $\mathbb{N} \rightarrow \mathbb{N}$. In COQ syntax, we will write the same information as follows, because `nat` is the name used in COQ to represent \mathbb{N} :

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

In this description, `nat` is called a *type*. Types describe collections of objects that can be treated in a uniform way.

Once a definition was made, the COQ system provides a command to describe the relevant information about the constant that is defined:

Gathering information on f	Response
1 <code>About f.</code>	<code>f : nat -> nat</code>

This response confirms that `f` is a function from \mathbb{N} to \mathbb{N} .

We can be more inquisitive in our requests for information about `f`, we can also require to know what is the value. This is done in the following manner:

1 <code>Print f.</code> 2	<code>f = fun n : nat => n + 1</code> <code>: nat -> nat</code>
------------------------------	--

We see in this example that *type information* is an important feature of the language. Type information is given twice in the result of the `Print` command, once in the fragment “`fun n : nat =>`” and once in the second line, behind the colon “`:`”. The first time, what we get is that the input of the function is expected to be a value of type `nat`. The second time, what we get is that the type of `f` itself is the function type `nat -> nat`. In fact, the colon character (when surrounded by spaces) will always be used to give type information. Remark the similarity between the output of `Print` and the mathematical notation in (1.1).

As we detail in chapter 3 types play an important role in COQ. In particular they are used to avoid confusion. 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 that an expression is *well typed*.

Check well-typedness	Response
1 <code>Check f 3.</code>	<code>f 3 : nat</code>

On the contrary `f` cannot be applied to something that is not a natural number, for example a function.

Type error	Response
1 <code>Check f (fun x : nat => x).</code> 2 3	Error: The term “(fun x : nat => x)” has type “nat -> nat” while it is expected to have type “nat”.

Indeed it makes little sense to compute the addition between a function and 1.

Expressions that are well typed can be *computed* by COQ, meaning that they are *reduced* to a “simpler” form, also called *normal*. For example `(f 3)` returns 4.

Evaluating a function	Response
1 <code>Eval compute in f 3.</code>	<code>= 4 : nat</code>

This seemingly trivial capability of Coq plays a crucial role in the Mathematical Components library, as we will see in the next chapter when doing proofs.

1.1.1 Higher order functions

The function `f` operates on natural numbers, of type `nat`, but we can also define functions that operate on functions, 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`.

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

Reading the first line of this statement, we see that a new object is being defined, called `repeat_twice`. We also see that this object is a function the argument of which is 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 to 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. This fragment 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`. The lesson here is that the application of a function to an argument is not systematically written using parentheses, as is customary in mathematics, but it is usually only written by juxtaposing the function (on the left) and the argument (on the right). Parentheses are only added when they are needed to resolve ambiguity. Also expressions written with several subexpressions side by side should be read as if there were parentheses around the subgroups on the left. We will illustrate this in the next few examples.

When we want to apply the function `repeat_twice` to the function `f` and the number 2 and perform the computation, this can be written as follows:

```
1 Eval compute in repeat_twice f 2. = 4 : nat
```

From the syntax point of view, we wrote three subexpressions only separated by space. By default, the parser understands this as `((repeat_twice f) 2)`, so we actually have a first subexpression which is `repeat_twice` applied to `f`. According to the definition of the first function, the value 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.

It is worth spending a few more words on the `(repeat_twice f)` subexpression. The `repeat_twice` program takes two arguments in input. However one can pass to it only the former one and obtain a perfectly valid term.

Partial application	Response
1 Check (repeat_twice f).	repeat_twice f : nat -> nat

If we look at the type of `repeat_twice` and add redundant parentheses we obtain $((\text{nat} \rightarrow \text{nat}) \rightarrow (\text{nat} \rightarrow \text{nat}))$: once the first argument is passed, $(f : \text{nat} \rightarrow \text{nat})$, we obtain a term which type is the right hand side of the main arrow, that is $(\text{nat} \rightarrow \text{nat})$. By passing another argument, like $(2 : \text{nat})$, we obtain an expression which type is, again, the right hand side of the main arrow, here `nat`. Remark that the type of each argument matches the left hand side of the arrow (as depicted with different underline styles). When this is not the case COQ issues a type error.

In COQ functions with multiple inputs are just functions producing functions. As a consequence *it is legit to apply functions partially*, and the `->` sign associates to the right.

1.2 Data: construction and destruction

It is possible to define a data type by giving in one declaration the type name and the methods to *build* elements of this type. The dual method to use (i.e *destruct*) elements of a data type is provided by COQ in the form of pattern matching construct.

This approach is used systematically in COQ to define a variety of basic data types, among which boolean values, natural numbers, pairs and sequences of values are the most prominent examples. We will study these examples in turn.

1.2.1 Boolean values

The type `bool` contains two values that can be distinguished in computations. 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. On the contrary they are a notion defined as follows.

Declaration of bool
1 Inductive bool := true false.

This declaration states explicitly that there are only two cases for elements of type `bool`. The first case is given by the constant `true`, the second case by the constant `false`. The constants `true` and `false` are provably distinct, and are the only way to construct a `bool`.

All definitions in COQ, like this one, which define a type by providing at the same time a type name and the only ways to build elements of the type are called *inductive* definitions. The two objects `true` and `false` are called *constructors*.

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

Queries	Response
1 <code>Check true.</code>	<code>true : bool</code>

To use boolean values in computations, we need a construct that makes it possible to build different values depending on whether some boolean input is `true` or `false`. This construct is written “`if .. then .. else ..`”. An example follows:

1 <code>if true then 3 else 2</code>

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 <code>Definition f23 b := if b then 2 else 3.</code>
--

As one expects, when `b` is `true` the expression `(f23 b)` evaluates to 2, while it evaluates to 3 otherwise.

Evaluating f23	Response
1 <code>Eval compute in f23 true.</code>	<code>= 2 : nat</code>
2 <code>Eval compute in f23 false.</code>	<code>= 3 : nat</code>

The Mathematical Components library provides a collection of boolean operations that mirror reasoning steps on truth value. The functions are called `negb`, `orb`, `andb`, `implyb`, respectively with notations `~~`, `||`, `&&`, and `=>` (the last three operators are infix, they appear between the arguments as in `b1 && b2`). In particular, it should be noted that the symbol `~~` uses two characters `~`, but it should not be confused with two consecutive occurrences of the one-character symbol `~` (that has a meaning in COQ, but is almost never used in the Mathematical Components library). In practice, readers become quickly accustomed to the distinction between the two symbols.

For instance, the function `andb` is defined as follows and evaluates to `true` if and only if both inputs are `true`.

1 <code>Definition andb b1 b2 := if b1 then b2 else false.</code>

1.2.2 Natural numbers

The description of natural numbers in the COQ system is inspired from the work of Peano at the beginning of the twentieth century. The basic idea is that any natural number is either 0 or the successor of an existing natural number. Moreover, the successor operation is injective. Peano also considers supplementary operations, like addition and multiplication, but we shall see that the COQ system provides enough machinery so that these operations can be described as defined functions on top of the basic tools.

The description of natural numbers as essentially generated by a constant `o` (capital “o” letter, to represent 0) and a function symbol `s` to represent the successor operation is given in the definition of the type `nat`.

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

This declaration states that the only ways to produce natural numbers are by using the constant `0`, or by applying the function `s` to an already existing natural number. So `0` is a natural number, `(s 0)` is a natural number, `(s (s 0))`, and so on, and these are the only natural numbers. When interacting with COQ, we will often see decimal notations, but these are only a parsing and display facility provided to the user for readability. So `0` is displayed `0`, `(s 0)` is displayed `1`, etc. Users can also type decimal numbers to describe values, but these are automatically translated into terms built with `0` and `s`.

The Mathematical Components library provides a few notations to make use of the constructor `s` more intuitive to read. In particular, if `x` is a value of type `nat`, `x.+1` is another way to write `(s x)`. The “`.+1`” notation binds more strongly than function application, so that this notation makes it possible to avoid some needs for parentheses. Similar notations are provided for up to 4 uses of the `s` constructor. This can be illustrated by the following dialog (where we assume that `f` is a function of type `nat -> nat`). For instance, we can write the following expression, when `f` is already defined with type `nat -> nat`.

Queries	Response
<pre>1 Check fun x => f x.+1. 2 Locate ".+4".</pre>	<pre>fun x => f x.+1 : nat -> nat Notation "n.+4" := S (S (S (S n)))</pre>

When writing functions that compute on natural numbers, we can proceed by cases, as was done in the previous section for boolean values. Again, there are two cases, either the natural number used in the computation is `0` or it is `n.+1` for some `n`, but in the latter case, it is often useful to also be able to use `n` for further computations. The usual approach in this case is to describe the two cases as patterns: if the data fits one of the patterns, the computation proceeds with the corresponding computation. Here is an example:

```
1 match n with 0 => true | p.+1 => false end
```

In this expression we describe a computation where some decision is taken depending on the value of the number `n`. The fragment “`0 => true`” expresses that when `n` is `0`, the final value is `true`. In this expression, `0` is a pattern and the computation checks whether `n` matches this pattern. The fragment “`p.+1 => false`” expresses that when `n` is the successor of some number `p` the final value is `false`. In other words, when `n` is the successor of somebody, then the returned value is `false`. The value bound to the name `p` mentioned in this pattern is not known in advance, it is actually computed at the moment the `n` is matched against the pattern. For instance, if `n` is `5`, then the value computed for `p` is `4`. In this particular instance, `p` is not used, so that we can also write as follows:

```
1 match n with 0 => true | _.+1 => false end
```

The symbol `_` is used to represent a pattern whose value will never be used.

This pattern can also be written using an “`if .. then .. else ..`” like syntax:

```
1 if n is 0 then true else false
```

For an example where the pattern variable is actually used, we can observe the function that returns the predecessor of a natural number when it exists, or 0 otherwise.

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

Reading this function in detail, we see that the argument `n` is matched with the pattern `u.+1`. When this comparison succeeds, the variable `u` receives the value of the immediate subterm of `n` the computation proceeds by computing the expression in the `then` branch of the `if` statement. This branch contains exactly `u`, so this is the result value. When the comparison fails, this means that `n` has the form `0`, the returned value is `n`, or equivalently `0`.

Assume for instance that `n` is 2, in other words `n` is the successor of the successor of `0`, which we could also write `0.+1.+1`. When matching `0.+1.+1` against the pattern `u.+1` the value of `u` is found to be `0.+1`, in other words 1.

Remark that we did omit the type of the input `n`. Being `n` scrutinized as a natural number, i.e. matched against the `u.+1` pattern, COQ fixes its type to `nat`. Indeed the `s` constructor belongs *exactly* to one inductive definition, the one of `nat`.

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

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

On the other hand, if we want to describe a different computation for three different cases and use variables in more than one case, we need to revert to the “`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`” contains an arbitrary 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 `|`.

For the symbols that are allowed in the pattern, they are essentially restricted to `0`, `s` and variable names. Thanks to notations, a pattern can also contains occurrences of the notation “`.+1`” which represents `s`, and decimal numbers, which represent the corresponding terms built with `s` and `0`. When a variable name occurs, this variable can be re-used in the result part. When computing with a natural number, all the pattern matching rules are tried successively against this number. For instance, if the input is 2, in other words `0.+1.+1`, the first rule cannot match, because this would require that `0` matches `u.+1.+1.+1` and we know that `0` is not the successor of any natural number, when it comes to the second rule `0.+1.+1` matches `v.+1`, because the rightmost `.+1` in the value of 2 matches the rightmost `.+1` part in the pattern and `0.+1` matches the `v` part in the pattern.

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

Non exhaustive case analysis	Response
<pre> 1 Definition wrong (n : nat) := 2 match n with 0 => true end. 3 </pre>	<pre> Error: Non exhaustive pattern matching: no clause found for pattern S _ </pre>

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

```

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

```

This 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
5   end.

```

1.3 Data: recursion

With only pattern-matching, we do not cope well with the fact that the type `nat` of natural numbers actually contains an infinity of elements. To really handle infinity, the COQ system provides another facility, known as *recursivity*. The idea behind recursivity is that we can define a function while supposing that it is already defined for a subset of the type. This adds a new element for which the function is defined, and progressively this makes that the function is defined for all elements of the type.

1.3.1 Recursion for integers

Many usual functions to deal with natural numbers are defined recursively. For instance, we can define addition in the following manner:

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

In this line, we learn that the keyword for defining a recursive function is **Fixpoint**. We see also that the function being defined, here called **addn**, is used in the definition of the function **addn** itself. This text expresses that the value of $(\text{addn } p.+1 \ m)$ is $(\text{addn } p \ m).+1$ and that the value $(\text{addn } 0 \ m)$ 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 $(\text{addn } 2 \ 3)$, we can know the value by performing the following computation:

$(\text{addn } 2 \ 3)$	use the “then” branch, $p = 1$
$(\text{addn } 1 \ 3).+1$	use the “then” branch, $p = 0$
$(\text{addn } 0 \ 3).+1.+1$	use the “else” branch
$3.+1.+1$	remember that $5 = 3.+1.+1$

When the computation finishes, the symbol **addn** disappears. In this sense, the recursive definition is really a definition. Remark that what the $(\text{addn } n \ m)$ program does is simply to count n times the successor starting from m .

An alternative way of writing **addn** relies explicitly on pattern-matching rules 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 systems imposes the constraint that the described computation must be *guaranteed to terminate*. The reason for this requirement is sketched in section 3.3. This guarantee relies on an analysis of the function’s description, to make sure that recursive calls always happen with a given argument that decreases. The most frequent technique to establish this guarantee is that recursive calls happen on a variable and this variable was introduced by a pattern in a pattern-matching construct or in an “**if** .. **is** ..” statement, with the additional constraint that the pattern-matching construct observes a variable that was one of the function’s inputs.

For instance, in the case of **addn**, the recursive call happens with p as the first argument, p was introduced by the pattern $p.+1$ in the corresponding pattern-matching rule, the pattern-matching construct was observing n , and n is the first

argument of the function `addn`. An erroneous, in the sense of non terminating, definition is rejected by COQ.

Non-terminating program	Response
<pre> 1 Fixpoint loop n := 2 if n is 0 then loop n else 0. 3 </pre>	<pre> Error: recursive call to loop has principal argument equal to "n" instead of a subterm of "n". </pre>

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

```

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

```

From a mathematical point of view, this definition can be quite unsettling. The second pattern matching rule indicates that when the second argument of the subtraction is 0, then the result is the first argument. But it also covers the case where the second argument is non-zero while the first argument is 0. This phenomenon is imposed by the fact that the function must be total (all functions are total in the COQ system) and the type `nat` does not contain any element to represent negative numbers. When the second argument is non-zero and the first argument is zero, a value must still be returned and *this value must be a natural number*. In the long run, this problem will be solved by introducing another type of numbers with negative integers.

Recursive functions also make it possible to test data for equality. We can for example write a 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.

<pre> 1 Eval compute in 0 == 0. 2 Eval compute in 1 == S 0. 3 Eval compute in 1 == 0.+1. 4 Eval compute in 2 == S 0. 5 Eval compute in 2 == 1.+1. 6 Eval compute in 2 == add 1 0.+1 </pre>	<pre> = true : bool = true : bool = true : bool = false : bool = true : bool = true : bool </pre>
--	---

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. The trailing `n` in the names is indeed to signal that these operations are on the `nat` data type. Postfix notations as `.-1` and `.*2` are provided for the predecessor and double functions.

The Mathematical Components library also provides concepts that makes sense only for the `nat` data type (hence the trailing `n` is omitted) and that we will use many times in this book. In particular the test identifying `prime` and `odd` numbers.

We advise the reader not to “explore” the library from COQ, but rather to browse the source files.¹ While the `Print` command can be quite useful in some circumstances, there is so much more information in the source files. The relevant files for this chapters are `ssrbool`, `ssrnat` and, later, `seq`.

Advice

Each file in the Mathematical Components library starts with a banner describing all concepts and associated notations. There is not better way to browse the library than reading these banners.



Before moving on it is worth detailing the definition of `leq` and some associated notations.

```

1 Definition leq m n := m - n == 0.
2 Notation "m < n" := (m.+1 <= n).
3 Notation "n > m" := (m.+1 <= n) (parsing only).
4 Notation "a <= b <= c" := ((a <= b) && (b <= c)).

```

Given that subtraction computes to 0 whenever `m` is less or equal that `n`, such definition is correct. Finally note that the strict order relation is just a notation, and that `(m > n)` is only accepted in input and is always printed as `(n < m)`.

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

Warning

There is no function testing if a natural number is strictly smaller than another one. $(a < b)$ is just syntactic sugar for $(a.+1 \leq b)$



1.4 Data: generic containers

It is often the case that we wish to group several objects in a sequence that 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 such data type as follows.

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

Such data type can hold zero or more natural numbers, 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, it can't hold boolean values. So if we need to manipulate a list of booleans we have to define a similar data type.

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

It is clear that this approach is problematic. First, every time we write a function that manipulates a list we have to choose a priori if such list holds numbers or booleans, even if such program does not really use the objects held by the list. A concrete example is the function that computes the length of the list, and in the current setting such function has to be written twice. Worse, starting from the next chapter we will prove properties of programs, and given the two size functions are “different” we are going to prove such properties twice.

However it is clear that the two data types we just defined follow a schema, as well as the two functions computing the length and the theorems we may prove about them. In other words one would write something like the following, where α is a schematic variable.

```
1 Inductive list := nil | cons (hd :  $\alpha$ ) (tl : list).
```

Even if this may look familiar jargon to some reader, we really want this α to be a first class citizen, as well as the type of lists carrying any type of values and not appeal to some notion of schema that is intuitively added on top of the COQ language. The reason being that we want to reason and prove theorems about such data type and related programs inside the formal language COQ provides,

and that we want the proofs of such theorems to be first class citizen too (see for example chapter 6).

COQ's language is powerful enough to fully characterize the generic list data type without appealing to the intuitive, but external, notion of schema.

1.4.1 The (polymorphic) sequence data type

The Mathematical Components library provides a generic data type to hold several objects of any type under the name `seq`.

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

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 a 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 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` and an existing list of type (`seq A`). 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`, `Type` is a keyword that 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 *type constructor*. Indeed `seq` alone is not a data type, but if one passes to it another data type, then it builds one. Remark that this also means that (`seq (seq nat)`) is a valid data type, and that the construction can be iterated. Types again avoid confusion: it is not licit to form the type (`seq 3`), since the argument `3` has type `nat`, while the function `seq` expects an argument of type `Type`.²

In principle, all objects of the data type definition have a type argument: `nil` is a function that takes a type `A` as argument and returns an empty list of type (`seq A`). The notation “`.. -> ..`” is not well suited for this kind of situation and the type of `nil` is rather written as follows:

```
1 ∀ A : Type, seq A
```

The same goes for the other constructor of list, 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
```

This information can actually be obtained from the system by using the command `About`.

²For historical reasons COQ may display the type of `nat` or `bool` as `Set` and not `Type`. We beg the reader to ignore such detail, that indeed plays no role in the Mathematical Components library.

Query	Response
1 About cons. 2 3	cons : $\forall A: \text{Type}, A \rightarrow \text{seq } A \rightarrow \text{seq } A$ Argument A is implicit and maximally inserted

In practice, the COQ system implements a mechanism to avoid that people need to give the type argument to the `cons` function. This is the information meant by the message “Argument A is implicit and ..”. Every time users write `cons`, the system automatically inserts an argument in place of `A`, so that this argument does not need to be written (the argument is *implicit*). It is then the job of the COQ system to guess what this argument is when looking at the first explicit argument given to the function. The same happens to the type argument of `nil`. In the end, this makes that user can write the following expression.

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

This example shows that the function `cons` is only applied explicitly to two arguments (the two arguments effectively declared for `cons` in the inductive type declaration). The first argument, which is implicit, has been guessed so that it matches the actually type of `2`. For `nil` also, 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.

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

Query	Response
1 Check 1 :: 2 :: 3 :: nil. 2 Check fun 1 => 1 :: 2 :: 3 :: 1. 3	[:: 1; 2; 3] : seq nat fun 1 => [:: 1, 2, 3 & 1] : seq nat -> seq nat

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

When programming with sequences, we can again use the pattern matching construct to express how data can be retrieved from a sequence. For instance, to retrieve the first element of a sequence of natural numbers or 0 if the sequence does not contain any element, we can write the following code:

³For the sake of completeness, some n-ary notation use a different, but more evocative, last separator. For example one writes `[| b1, b2 | b3]` and `[==> b1, b2 => b3]`.

```

1 Definition first_element_or_0 (s : seq nat) :=
2   match s with
3   | nil => 0
4   | cons a _ => a
5   end.

```

Using the notations for `cons` and `nil`, we can also write

```

1 Definition first_element_or_0 (s : seq nat) :=
2   match s with
3   | [::] => 0
4   | a :: _ => a
5   end.

```

Using the `if` syntax, we can write:

```

1 Definition first_element_or_0 (s : seq nat) :=
2   if s is a :: _ then a else 0.

```

1.4.2 Recursion for sequences

Because the second argument of `cons` can also be a sequence, it is possible to build sequences that are quite long, but always finite. In this respect, the sequences are very similar to the natural numbers. Recursion is again the solution to cope with the arbitrary length of sequences with functions that can be described concisely.

For instance, we can define a size function that counts the number of elements in a sequence.

```

1 Fixpoint size A (s : seq A) :=
2   if s is _ :: t1 then (size t1).+1 else 0.

```

During computation on a given sequence, this function will traverse the whole list, adding 1 to the result for ever `cons` pattern that is encountered. It should be noted that in this definition, the function `size` is described as a two argument function, but the recursive call (`size t1`) is done by providing explicitly only one argument, `t1`. This is because the COQ system makes arguments of functions that can be guessed from the type of following arguments automatically implicit. This feature is effective directly at definition time.

Another example of recursive function on lists is a function that constructs a new list where all elements are values of a given function applied to the elements of a list. This function can be defined in this manner.

```

1 Fixpoint map A B (f : A -> B) s :=
2   if s is e :: t1 then f e :: map f t1 else nil.

```

With this function it is also interesting to understand how we can improve the 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:

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

In addition to the function `map` and the associated notation we studied 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.

1.4.3 Other containers

The simplest polymorphic data type one can think about is the option type

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

It represents a box that can be either empty or contain a value. It is sometimes used to represent a partial function, or a filter.

```
1 Definition only_odd n : option n := if odd n then Some n else None.
```

For example the sub-type kit presented in chapter 6 makes use of the option type to describe a partial injection.

Another widespread polymorphic data type is the one of pairs, that lets one put together any two values. While we leave its definition as an exercise (see 1) we briefly discuss the associated notations: `(a,b)` builds the pair of `a` and `b` while `c.1` projects the first component of the pair `c`. The type of pairs is denoted by an infix `*` symbol: `(nat * bool)` is the type of pairs that have a natural number as the first component and a boolean value as the second one.

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

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

As one expects pairs can be nested, as in `(3,true,4)`, that is sugar for `((3,true),4)`. In such situation the value `true` can be accessed by writing `(3,true,4).1.2`.

Given the notation for pairs, the expression `(a * b)` is now “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 ambiguity is somewhat justified by the fact that the pair data type can be seen as the (cartesian) product of the arguments. Such ambiguity can be resolved by annotating the expression with a specific label: `(a * b)%N` interprets the infix `*` as multiplication, 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 [22, section 12.2]).

1.5 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 and rely in a uniform way on a given type and 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 x :: xs then f x (foldr a xs) else a.
13
14 End iterators.
```

The **Section** and **End** keyword delimit a scope in which the variables **T**, **A** and **f** are available. Such variables are used in the definition of **iter** and **foldr**. When the section is closed, such variables are abstracted.

Query	Response
<pre> 1 About iter. 2 About foldr. 3</pre>	<pre> iter : ∀ T : Type, nat -> (T -> T) -> T -> T foldr : ∀ T A : Type, (T -> A -> A) -> A -> seq T -> A</pre>

Variables are abstracted in their order of declaration, see **T** and **A** in the type of **foldr** and unused ones are omitted, for example **iter** is not abstracted on **f**.

The **Implicit Type** annotation tells COQ that, whenever we name an input **x** its type is supposed to be **T**.

Once the section is closed, **iter** and **foldr** behave exactly as if they were defined as follows:

```

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

For example we can compute the subtraction of 5 from 7, or the addition of all numbers in `[:: 1; 2; 3]` as follows:

<pre> 1 Eval compute in iter 5 pred 7. 2 Eval compute in foldr addn 0 [:: 1; 2; 3].</pre>	<pre> = 2 : nat = 6 : nat</pre>
---	---------------------------------

The **Section** mechanism lets one factor variable declaration and type annotations among multiple definitions. Incidentally it also let us experiment with

the crucial notion of symbolic computation.

1.6 Symbolic computations under a context

Which is the status of the `foldr` program *before* the `Section` is closed? What are `T` and `A` and `f`?

```
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.
```

The variables `T`, `A` and `f` form the so called *local context* under which all operations take place. For example, computing the type of `foldr` gives the following:

Query	Response
1 About foldr.	foldr : A -> seq T -> A

This indicates that `foldr` is not a polymorphic function (yet) and that it takes only two arguments, an element of `A` and a list of `T`.

If the operation we intend to perform is computation, then `T`, `A` and, most importantly, `f` are symbols. `f` in particular represents an unknown function. Computation involving `f` become hence symbolic:

<pre>1 Variable init : A. 2 Variables x1 x2 x3 : T. 3 Eval compute in 4 foldr init [:: x1; x2; x3].</pre>	<pre>= f x1 (f x2 (f x3 init)) : A</pre>
---	--

COQ has developed the expression symbolically. If we substitute in such expression the values we passed earlier we obtain

```
1 addn 1 (addn 2 (addn 3 0))
```

that indeed is an expression that computes to 6.

In later chapters we are going to prove properties about `foldr`. In such case the local context has to be understood as containing names (and type annotations) of fixed, but unknown, objects. Hence proving a property of `foldr` when the iterated function `f` is unknown means proving that such property is true for any value `f` can possibly take.

The last observation to be made about symbolic computation is that it is going to play an important role in the activity of proving and that the way programs are written has an impact on the way they compute symbolically. For example, let's consider this alternative definition of the addition between natural numbers

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

It is a sensible definition, and we can show that the two additions are equivalent. Still, their computational behavior differ. This time we use the `simpl` evaluation strategy. The relevant difference is that `simpl` leaves expressions in nicer forms whenever they contain variables.

<pre>1 Variable n : nat. 2 Eval simpl in pred (add n.+1 7). 3 Eval simpl in pred (addn n.+1 7).</pre>	<pre>= pred (add n 8) = addn n 7</pre>
---	--

Here we see that since `addn` exposes bits of its final result early the `pred` function can compute, and cancels the `.+1` coming out of the sum. On the contrary `add` does not expose a successor, instead it carries it on the second argument, hence `pred` is stuck. In some sense `addn` is more explicit about its result, and this helps, for example, to show that `(addn n.+1 7)` is different from 0 (see for example the proof of `muln_eq0` in section 2.2.2). As we have seen COQ computes automatically and in this case computation would expose a `s` symbol and no natural number of the form `(s ..)` can be equal to 0.

1.7 Iterators in mathematics

Numbers and sequences of objects are common in mathematics, so we feel no need to justify our choice to present them in this chapter. On the contrary it is legit to wonder which role programs like `foldr` play in mathematics. Let's take the left hand side of these two formulas:

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

They can both be faithfully represented by using `foldr`.

<pre>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))).</pre>
--

The `iota` function generates the list of natural numbers corresponding to the range of the summation. We also provide to `foldr` the neutral element for the addition as the initial value of the iteration.

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

It is important that the iteration happens following the order of the list. In this case the operation, addition, is commutative, so it does not really matter, but it may not be the case for example if the iterated operation is group multiplication. What is even more crucial is that `foldr` takes in input a *functional argument that is able to represent faithfully any general term*.

The ability to write programs that manipulate functions, like `foldr`, is crucial not only to provide convenient notations. As we sketched in the previous

section we will be able to prove properties about `foldr` without knowing the iterated function. By assuming properties linking the iterated operation to the initial value, like forming a Monoid, we will be able to provide a generic theory of iterated operations in section 5.8. Such theory becomes richer when the operation is also commutative, and even richer when we combine two operations that distribute.

1.8 Notations and abbreviations

We mentioned many times already the `Notation` command only appealing to the reader's intuition: a notation relates some symbols with a term. The actual syntax is more complex, letting one specify associativity and precedence to the parsing engine, as well as the group (called scope and identified by a scope delimiter) to which the notation belongs and some hint for printing, like good line breaking points.

```
1 Notation "m + n" := (addn m n) (at level 50, left associativity).
2 Notation "n .+1" := (succn n) (at level 2, left associativity,
3   format "n .+1") : nat_scope.
```

A comprehensive description of such command goes behind the scope of this book, the interested reader shall refer to [22, section 12].

What is worth mentioning here is some conventions either imposed by COQ or adopted in the Mathematical Components library.

- A symbol starts with a non alpha-numerical character followed by any character. For example `.+1` is a single symbol, as `.+4` and `%/` and `<=` and `[::`. Only existing symbols are taken into account: `3.+1.+1` is parsed as a number followed by two occurrences of the `.+1` symbol; `.+1.+1` could be a symbol, in principle, but no notation declares it.
- Notations that are typically denoted with a letter, like $N(G)$ for the normalizer of the group G are represented with 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.
- n-ary notations begin with `[` followed by the symbol of the operation being repeated, as in `[&& true, false & false]`.
- postfix notations begin with `.`, as in `.+1` and `.-group` to let one write `(p.-group G)`.

Another frequently used form of notation is called syntactic abbreviation. It simply lets one specify a shorter name. For example the `s` constructor of natural numbers (placed in `Datatype` Module) 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 := Datatypes.S.
```

Abbreviations can also be local to a term, to avoid repetitions.

```
1 let p := n.-1 in
2   if n is m.+1 then p == m else p == 0
```

Here `p` is a name given to the expression `n.-1` and used twice the following expression. Let-in abbreviations are fully transparent, exactly as notations. There is no semantic difference between the previous expression and the following one.

```
1   if n is m.+1 then n.-1 == m else n.-1 == 0
```

Still the computation engine of Coq uses to let-in as hints to compute the named expression only once, so the execution time of the two expressions above may vary.

Last, we recall the notion of implicit argument sketched in section 1.4. Once implicit arguments are active for the `cons` and `nil` constructors the writing `(cons 3 nil)` shorthand for `(cons _ 3 (nil _))`. The `_` denotes a term, here a type, to be inferred by Coq. In this case the typing constraints of impose both `_` to be `nat`. 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 3 nil)`.

1.9 Aggregated data: records

Sometime one wants to aggregate already available data into a single object. For example a point in the 3-dimensional space could be represented with three natural numbers.

Such need is so frequent that Coq provides a specialized command to declare such class of data type.

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

Such writing is mostly equivalent to the following one

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

The main advantage of using `Record` is that Coq uses the names of the constructor fields to generate functions to access the field's contents. We call these programs *projections*. I.e. after the `Record` declaration one can write:

Query	Response
<pre>1 Eval compute in x (Point 3 0 2). 2 Eval compute in y (Point 3 0 2).</pre>	<pre>= 3 : nat = 0 : nat</pre>

Where the code for the `x` projection is, as expected:

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

Inductive data with a single constructor, as records, comes equipped with a specific notation for single-branch pattern matching:

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

Records can as well be polymorphic, for example the pair data type of Exercise 1 can carries two values of any type.

Records play a major role in the Mathematical Components library, in particular in the modelling of algebraic structures. As we see in later chapters records lets one pack together types, operations on these types and finally the properties of these operations. For example a record will model the signature of a Ring.

1.10 Exercises

Exercise 1. *The pair data type*

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

```
1 Notation "( A , B )" := (pair A B).
2 Notation "p .1" := (fst p).
3 Notation "p .2" := (snd p).
4 Eval compute in (4, 5).1.
5 Eval compute in (true, false).2.
```

```
= 4 : nat
= true : bool
```

Remark that the pair data type has to be polymorphic in order to be applicable to both natural numbers and booleans.

Exercise 2. *Addition with iteration*

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

Exercise 3. *Multiplication with iteration*

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

Exercise 4. *Find the n -th element*

Define a program taking a default value, a list and a natural number. Such program returns the n -th element of the list if n is smaller than the size of the list. It returns the default value otherwise.

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

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

Exercise 5. *List reversal*

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

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

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

Exercise 6. * *List flattening*

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

```

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

```

```

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

```

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

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

```

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

```

```

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

```

1.10.1 Solutions

Answer of Exercise 1

```

1 Inductive prod (A B : Type) := pair (a : A) (b : B).
2 Notation "A * B" := (prod A B) : type_scope.
3 Notation "( A , B )" := (pair A B).
4 Definition fst A B (p : A * B) := let: (a, _) := p in a.
5 Definition snd A B (p : A * B) := let: (_, b) := p in b.

```

Answer of Exercise 2

```

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

```

Answer of Exercise 3

```

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

```

Answer of Exercise 4

```

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

```

Answer of Exercise 5

```

1 Fixpoint catrev T (s1 s2 : seq T) :=
2   if s1 is x :: xs then catrev xs (x :: s2) else s2.
3
4 Definition rev T (s : seq T) := catrev s [::].

```

Answer of Exercise 6

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

Answer of Exercise 7

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

Chapter 2

Statements and proofs

First steps in formal proofs

The COQ system lets one not only define data types and programs, but also prove properties about them. In this chapter we explain how theorems are stated and proved, focusing on simple statements and basic proof commands.

Stating a lemma is a very delicate matter. Even if two formulations for the same statements are logically equivalent, one can be much simpler to prove or invoke in the proof of another theorem. The approach we follow in the current chapter and that we fully develop in chapter 4 is paradigmatic in the Mathematical Components library and roughly consist in favouring the use of computable definitions and equational reasoning whenever possible.

2.1 Formal statements

In this section, we illustrate how to state rather simple (tentative) theorems, starting with the simplest, and most pervasive, kind of mathematical sentences: 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 $(a+b)^2=a^2+2ab+b^2$ is not.

In chapter 1, we have used the `Check` vernacular command to query the type of a defined object. We can use the very same command to *check* whether a formal statement is well formed:

```

1 Check 3 = 3.
2 Check false || true = true.

```

```

3 = 3 : Prop
false || true = true : 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. One first important thing to keep in mind is that, in COQ, formal statements are themselves *terms*. And therefore, statements also have a *type*. An equality statement is obtained by applying the constant `eq` to two arguments, of the same type, which 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. Sometimes we say *boolean proposition* to stress that the proposition is made by equating two boolean expressions, as in Line 2 above.

The `About` vernacular command, which lets one obtain information on a constant, like its type and the status of its argument, applies just as well to propositions and their components. For instance we can learn more about the constant `eq`:

The polymorphic equality predicate	Response
<pre> 1 Locate "_ = _". 2 3 About eq. 4 </pre>	<pre> "x = y" := (eq x y) eq : ∀ A : Type, A -> A -> Prop Argument A is implicit ... </pre>

We learn here that the constant `eq` is a *predicate*, i.e. a function letting build a proposition. Sometimes we say *boolean predicate* to indicate a function to `bool`, i.e. a test one can use to form a proposition by equating its result to `true`, as in section 2.1.3.

The equality predicate is polymorphic: exactly as we have seen in the previous chapter the `forall` quantifier is used to make the variable `A` range over types. This is why we were able to use the same equality constant in both our examples, once for equating two terms of type `nat` and once for equating two terms of type `bool`.

Since its first argument is implicit, its value is not provided by the user but rather guessed from the type of the hand-sides of the equality we state: `(3 = 3)` unfolds to `(eq _ 3 3)` and the missing value is set to `nat`, the type of `3`. Similarly, `(true = false || true)` unfolds to `(eq _ true (false || true))` and the missing value is found to be `bool`, the common type of `true` and `(false || true)`.

As the COQ system checks the well-typedness of statements, the two hand-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` (or `Theorem`, `Remark`, `Corollary`,... which are all synonyms for what matters here). The `Lemma` keyword is followed by the name chosen for the lemma and then by the statement itself. `Lemma` (and its siblings) is in fact a variant of the `Definition` syntax we used in chapter 1: everything we mentioned about it also applies here. The `Proof` keyword marks the beginning of the proof text and after it is executed, the system displays the current state of the formal proof in a dedicated window.

Stating a lemma and starting its proof

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

Response after line 2: proof state

```
1 subgoal
```

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

Indeed, COQ is now in so-called *proof mode*. The current proof state consists in a list of named hypotheses, on top of the horizontal bar (empty here), and the statement of the current goal (the conjecture to be proved) below the bar.

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

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

From now on we will omit the `Admitted` proof terminator, and simply list the statement of the lemmas to discuss their formulation. Indeed the rest of the present section is dedicated to a tour of more complex formal statements.

2.1.2 Identities

Ground equalities are just a very special case of mathematical statements called *identities*. An *identity* is an equality relation $A = B$, which 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 associativity has been prescribed at the definition time of the infix `+` notation. Also `Lemma`, just like `Definition`, allows for dropping type annotations if these types can be inferred from the statement itself:

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

Boolean identities are an important special class of identities. For instance, the `orbT` statement expresses that `true` is right absorbing for the boolean disjunction operation `orb`. Recall 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 formulae are equivalent, or that `(b || true)` is a propositional tautology. We provide below some other examples of such propositional equivalences stated as boolean identities.

```
1 Lemma orbA b1 b2 b4 : 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.
```

Their proofs are left as an exercise at the end of this Chapter.

2.1.3 From boolean predicates to formal statements

More generally, boolean identities involve arbitrary boolean predicates (not only boolean connectives) and feature variables of an arbitrary type (not only of type `bool`). For instance, we can state that for any term `n : nat`, the following boolean test holds:

Stating that a boolean predicate holds

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

We are expressing here that the statement $0 \leq n$ holds, because the truth value of the boolean `(0 <= n)` is `true`, whatever term of type `nat` is substituted for the parameter `n` in the statement. In section 2.2.2, we will see how to build a formal proof of such a statement.

Given how frequently we state that a boolean predicate is equal to `true`, we are going to omit the “`. = true`” part of the statements. Indeed COQ is able to insert such piece of “text” automatically, whenever it fits. The COQ mechanism for doing that is out of scope here, it will be explained in section 4.4.

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

We can express that two boolean statements are equivalent using identities, generalizing the examples of propositional tautologies we gave in section 2.1.2. Here is a few examples, built with the boolean predicate on natural numbers that we crafted in chapter 1: the boolean equality `==` and its negation `!=`, the order relation `<` and its large version `<=`, and the divisibility predicate `%|`. We can omit the type of the parameters: they are all of type `nat`.

```

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 which hold for *any* value of their parameters. A property which holds only when some condition is verified by its parameters is stated using an *implication*. The COQ syntax for such connective is “ \rightarrow ”. For instance:

Implication

```

1 Lemma leq_pmul1 m n : n > 0 -> m <= n * m.
2 Lemma odd_gt0 n : odd n -> n > 0.

```

The attentive reader might have noticed that this arrow \rightarrow is the same as the one we have used in chapter 1 in order to represent function types. This is no accident, but we will explain this phenomenon later in section 3.1.1. For now let us remark that \rightarrow associates on the right: 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

```

The `Section` mechanism we saw in chapter 1 is also convenient in order to factor the parameters and the hypotheses shared by a handful of lemmas.

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 Admitted.

```

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

Scheme of a complete proof

```

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

```

Once the proof is complete, we can replace the `Admitted` command by the `Qed` one. This 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 nature of proof steps and the corresponding tactics.

2.2.1 Proofs by computation

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

Reflexivity of equality	Proof finished
<pre>1 Lemma my_first_lemma : 3 = 3. 2 Proof. by [].</pre>	<pre>No more subgoals.</pre>

Indeed, this statement trivially holds, because the two hand sides of the equality are the same. The tactic “`by []`” is the command that implements this nature of *trivial* proof step. `by` typically prefixes another tactic (or a list of thereof) and checks that such tactic trivializes the goal. In this case the goal is already trivial, hence the `[]` standing for an empty list of tactics.

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

<pre>1 Lemma my_first_lemma : 3 = 3. 2 Proof. by []. Qed. 3 About my_first_lemma.</pre>	<pre>No more subgoals. my_first_lemma is defined. my_first_lemma : 3 = 3</pre>
---	--

Just like when it was `Admitted`, this script results in a new definition being added in our context, that can 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`. On the contrary `Admitted` happily accepts false statements!

What makes the `by []` tactic interesting is that it can be used not only when both hand 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 $2 + 1 = 3$.

<pre>1 Lemma my_second_lemma : 2 + 1 = 3. 2 Proof. by []. Qed.</pre>
--

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

Computation is not limited to ground terms, it is really about using the rules of the pattern matching describing the code of the function. For instance the proof of the `addSn` identity:

Reflexivity by symbolic computation

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

is trivial because it is a direct application of one of the branches of the definition of the `addn` function. Statements like $(0 + n = n)$ or $(0 < n.+1)$ can be proved in a similar way, but also $(2 + n = n.+2)$, which requires a longer computation.

The last word to be said about `by []` is that it is a terminating tactic. If it is unable to solve the goal it fails and stops COQ from processing the proof script. For this reason it is also coloured 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:

Double negation elimination

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

Failing proof script

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

Indeed, proving this identity requires not only unfolding the definition of `negb`, but also performing a *case analysis* on the boolean value of the parameter `b`. The tactic `case` implements this action:

Reasoning by cases

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

Case analysis

```
2 subgoals

=====
 $\neg\neg\neg\neg \text{true} = \text{true}$ 
subgoal 2 is:
 $\neg\neg\neg\neg \text{false} = \text{false}$ 
```

More precisely, the tactic “`case: b`” indicates that we want to perform a case analysis on `b`, whose name follows the separator `:`. The COQ system displays the state of the proof after this command: the proof now has two parallel branches, one in which the parameter `b` takes the value `true` and one in which the parameter `b` takes the value `false`.

As a consequence, we are going to provide two distinct pieces of script, one per each subproof to be constructed. We start with the branch associated with the `true` value, and to signal it is a sub proof we indent the corresponding script. Now that `b` is a concrete value each proof is trivial by computation, in the spirit of what we did in section 2.2.1.

First trivial case

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

Second goal

```
1 subgoal

=====
 $\neg\neg\neg\neg \text{false} = \text{false}$ 
```

Once the first goal is solved we have only one subgoal left, which we solve by the very same means.

Second trivial case	Finished proof
<pre> 1 Lemma negbK b : ~~ (~~ b) = b. 2 Proof. 3 case: b. 4 by []. 5 by []. 6 Qed.</pre>	<pre> No more subgoals. negbK is defined</pre>

In fact, we can use the `by` command as a true prefix, to have the system check that after the `case` tactic, the proof becomes trivial, in both branches of the case analysis. The proof script is then a one-liner:

Double negation elimination: final script
<pre> 1 Lemma negbK b : ~~ (~~ b) = b. 2 Proof. by case: b. Qed.</pre>

Case analysis with naming

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

<pre> 1 Lemma leqn0 n : (n <= 0) = (n == 0). 2 Proof.</pre>	<pre> n : nat ===== (n <= 0) = (n == 0)</pre>
--	--

The proof requires unfolding the definition of the comparison operations `<=` and `==`, both defined by case analysis on their first argument, and to inspect their respective values in the case when the second argument is zero. A case analysis on term `(n : nat)` has two branches: one in which `n` is 0 and one in which `n` is `(S k)` for some `(k : nat)`. We hence need a variant of the `case` tactic, in order to *name* the parameter `k` which pops up in the second branch:

case with naming	Proof state
<pre> 1 Lemma leqn0 n : (n <= 0) = (n == 0). 2 Proof. 3 case: n => [k].</pre>	<pre> 2 subgoals, subgoal 1 ===== (0 <= 0) = (0 == 0) subgoal 2 is: (k < 0) = (k.+1 == 0)</pre>

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

The first goal can be easily solved by computation, as both hand 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`. It can also be solved by computation, as both hand sides of the equality evaluates to `false`. The final proof script is hence:

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

A similar example, again of a boolean equivalence, let us introduce the need of more powerful proof techniques. We claim that the product of two (natural) numbers is zero if and only if one of the numbers is zero. We also recall the definition of `muln`.

Specifying `muln`: a double case analysis attempt

```
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, and whatever value `n` takes, both hand 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 [].
```

One goal left

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

Note that we can use the name `m` in our case analysis without ambiguity. The next step in the proof is a case analysis on the number `n`. In order to let us treat

first the successor case, despite it corresponds by default to the second subgoal, we are going to use the “; *last first*” tactic suffix:

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

Two goals left (in reverse order)

2 subgoals, subgoal 1

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

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

Indeed 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 also evaluates to **false**: by the definition of `muln` term $(m.+1 * k.+1)$ evaluates to $(k.+1 + (m * k.+1))$ and by definition of the addition `addn`, this in turn reduces to $(k + (m * k.+1)).+1$. The left hand side term hence is of the form $t.+1 == 0$, where t stands for $(k + (m * k.+1))$, and this reduces to **false**. To conclude, this branch of the case analysis is also solved 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 be made more compact and more linear by using an optional part of the introduction pattern, which makes COQ inspect the subgoals created by the case analysis and solve the trivial ones using the `by []` tactic. For instance in our case, let us add the optional `// simplifying` switch to the introduction pattern of the first case analysis:

A simplify intro pattern

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

Proof state

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

Only the first generated subgoal is trivial, and has been closed, so 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 [10, 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 on the code of the multiplication.

To conclude the proof we introduce one more proof command, `rewrite`, that lets us appeal to an already existing lemma. This is the subject of the next section.

2.2.3 Rewriting

This section explains how to perform local replacements of 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 eponym `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:

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

which is equal modulo computation to `(true = true)`, hence trivial. But because of the definition of `muln` 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 2.3.3. Anyway, a lemma proving what we need is already available in the library:

```

1 Lemma muln0 n : n * 0 = 0.

```

This is not a coincidence since the Mathematical Components library is huge. Finding the “right” lemma is a key ability in writing formal proofs. Section 2.4 is dedicated to this activity.

The `rewrite` tactic can use such lemma to perform the desired replacement.

First rewrite	Proof state
<pre> 1 Lemma muln_eq0 m n : 2 (m * n == 0) = (m == 0) (n == 0). 3 Proof. 4 case: m => [m] //. 5 case: n => [k] //. 6 rewrite muln0. </pre>	<pre> m : nat ===== (0 == 0) = (m.+1 == 0) (0 == 0) </pre>

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 we use $_$ to denote a wild-card, which reflects the fact that 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 the 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:

```

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_mul2l`, which provides a necessary and sufficient condition for the comparison $(m * n1 \leq m * n2)$ to hold:

Another example: ordered products
<pre> 1 Lemma leq_mul2l m n1 n2 : (m * n1 <= m * n2) = (m == 0) (n1 <= n2). </pre>

The proof goes as follows: by definition of the order \leq , the left hand side can be written as $(m * n1 - m * n2 == 0)$, which in turns factors into $(m * (n1 - n2) == 0)$. We then conclude using lemma `muln_eq0`, since $(n1 - n2 == 0)$ is the definition of $(n1 \leq n2)$.

The first step hence consists in using the definition of the `leq` relation, de-

noted by the `<=` infix notation, to replace the right hand side by a subtraction. This can be performed using the following equation:

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

For this purpose, we use the rewrite tactic. However, the command `rewrite leqE` only affects the first occurrence of `<=` and we would like to treat both.

Unfolding leq	Proof state
<pre>1 Lemma leq_mul2l m n1 n2 : 2 (m * n1 <= m * n2) = (m == 0) ... 3 Proof. 4 rewrite leqE.</pre>	<pre>m, n1, n2 : nat ===== (m * n1 - m * n2 == 0) = (m == 0) (n1 <= n2)</pre>

In order to rewrite *all* the possible instances, we use a repetition flag:

<pre>1 Lemma leq_mul2l m n1 n2 : 2 (m * n1 <= m * n2) = (m == 0) ... 3 Proof. 4 rewrite !leqE.</pre>	<pre>m, n1, n2 : nat ===== (m * n1 - m * n2 == 0) = (m == 0) (n1 - n2 == 0)</pre>
--	--

Now the definition of `<=` has been exposed *everywhere* in the goal, i.e. at both its occurrences in this initial goal. We can now factor `m` on the left, using 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 wild-cards coincide. The syntax for right-to-left rewriting consists in adding a minus `-` to the name of the rewrite rule:

Rewrite right-to-left	Proof state
<pre>1 Lemma leq_mul2l m n1 n2 : 2 (m * n1 <= m * n2) = (m == 0) ... 3 Proof. 4 rewrite !leqE. rewrite -mulnBr.</pre>	<pre>m, n1, n2 : nat ===== (m * (n1 - n2) == 0) = (m == 0) (n1 - n2 == 0)</pre>

Consecutive rewrite steps can be collapsed as follows:

<pre>1 Lemma leq_mul2l m n1 n2 : 2 (m * n1 <= m * n2) = (m == 0) ... 3 Proof. 4 rewrite !leqE -mulnBr.</pre>	<pre>m, n1, n2 : nat ===== (m * (n1 - n2) == 0) = (m == 0) (n1 - n2 == 0)</pre>
--	--

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

<pre> 1 Lemma leq_mul2l m n1 n2 : 2 (m * n1 <= m * n2) = (m == 0) ... 3 Proof. 4 rewrite !leqE -mulnBr muln_eq0. </pre>	<pre> m, n1, n2 : nat ===== (m == 0) (n1 - n2 == 0) = (m == 0) (n1 - n2 == 0) </pre>
---	--

The proof can now be completed by prefixing the tactic with the `by` command, which concludes the proof.

So far we have learnt the basic use of the `rewrite` tactic. The interested reader can find in section 2.5.1 more details on the matching algorithm and on the flags supported by `rewrite`. Lets now move on to a more interesting class of formal statements.

2.3 Quantifiers

2.3.1 Universal quantification, first examples

In chapter 1, we have defined functions with boolean values; they are named *boolean predicates*. For instance, the `leq` predicate is defined as a boolean predicate on natural numbers, as:

```

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

```

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

```

1 Definition leq := fun n m => m - n == 0.

```

and the type of the constant `leq` is:

```

1 About leq.

```

```

leq : nat -> nat -> bool

```

In the present Chapter, we have used a similar syntax when stating parametric statements, like:

```

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

```

The curious reader might already have tested the answer of the `About` command on the parametric lemmas we have used or proved in the previous sections, like for instance:

Inspecting lemma `leqn0`

```

1 About leqn0.

```

Universal quantification

```

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

```

She has thus observed that COQ's output features a prenex `forall` quantifier. This universal quantifier binds a natural number, and expresses –as expected– that the equation holds for any natural number. The type 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 do not need to be in prenex position. In the following example the

second hypothesis is itself a quantified formula. Recall the definition of `nth` in exercise 4.

```
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: “ $\forall (f : T1 \rightarrow T2) (s : \text{seq } T1), \dots$ ” is syntactic sugar for “ $\forall f : T1 \rightarrow T2, \forall s : \text{seq } T1, \dots$ ”. However in this case of prenex quantification we would rather 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.

We can also use quantifiers 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 (T : Type) (op : T -> T -> 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 provides many predicates to state in a consistent and compact way standard properties. Here we recall only few of them. Remark that `(rel T)` is an abbreviation for the type `(T -> T -> bool)`.

```
1 Section StandardPredicates.
2 Variable T : Type.
3 Implicit Types (op add : T -> T -> T) (R : rel 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.
```

Note that 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 convention is also useful to search the library, as detailed in section 2.4.

Other relevant predicates worth mentioning are the one describing standard function properties:

```

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.

```

It is now time to discuss the type of these predicates.

```
1 About commutative.
```

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

`commutative` is a polymorphic function, in τ , taking a binary operation and building a *proposition*, i.e. it has the same status as the identity statements that we have seen previously in this chapter:

```

1 Check 3 = 3.
2 Check (commutative addn).

```

```

3 = 3 : Prop
commutative addn : Prop

```

Indeed, the class of propositions is closed under universal quantification `forall` (and also under implication `->`). Chapter 3 will provide a more in-depth study of these propositions.

2.3.2 Proving with conditional lemmas

In order to use a lemma with a parameter, we can use the `apply` tactic, which finds the appropriate instance (as the `rewrite` tactic does):

```

1 Lemma example a b : a + b <= a + b.
2 Proof. apply: leqnn. Qed.

```

Find the appropriate instance works up to computation:

```

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

```

By the way, in order to save the effort of mentioning too trivial steps in the proof script, we can extend the power of the `by` terminator to make it aware of some (concluding) 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 the lemma 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`.

The `apply` tactic can also be used on non-atomic statements. For instance, the following lemma can be used to perform a flavour of proof by contra position:

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

We see this how this lemmas is used in an excerpt of the proof that there are infinitely many primes (the full proof will be object of study in section 4.3.1). Given a number m we show that any prime divisor p of $(m'! + 1)$ is greater than m , or better that it is not the case that m is bigger.

```
1 Lemma example m p : prime p -> p %| m '! + 1 -> ~~ (p <= m).
```

We start our formal writing of such proof by naming 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.

```
1 Lemma example m p : prime p ->
2   p %| m '! + 1 -> ~~ (p <= m).
3 Proof.
4 move=> prime_p.
```

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

By reasoning by contraposition, via `apply: contraL`, we are left to prove that p is not a divisor of $(m'! + 1)$ under the assumption that p is smaller then m :

```
1 Lemma example m p : prime p ->
2   p %| m '! + 1 -> ~~ (p <= m).
3 Proof.
4 move=> prime_p.
5 apply: contraL.
```

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

First observe that the `apply:` tactic has found the correct values of the two parameters c , b of lemma `contraL`, namely $(p <= m)$ and $(p \%| m'! + 1)$, by comparing the statement to be proved with the conclusion $(b \rightarrow \sim c)$ of the statement of the lemma. Indeed lemma `contraL` provides a sufficient condition for a statement of this shape to hold. After the execution of the tactic, we are hence left with proving this sufficient condition.

The rest of proof informally goes as follows: if $p \leq m$, then p certainly divides $m!$. But then p divides $m! + 1$ if and only if p divides 1. Since p is a prime, it cannot be 1.

We start our formal writing of such proof by naming the hypothesis $(p <= m)$.

```
1 Lemma example m p : prime p ->
2   p %| m '! + 1 -> ~~ (p <= m).
3 Proof.
4 move=> prime_p.
5 apply: contraL.
6 move=> leq_p_m.
```

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

The next step is to use the following lemma:

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

It states a conditional equivalence, expressed as a conditional identity. We can replace our current goal with $\sim (p \%| 1)$ by rewriting (the appropriate instance of) this identity. This operation will open an extra goal requiring a proof of (the corresponding instance of) the side condition.

```

1 Lemma example m p : prime p ->
2   p %| m '! + 1 -> ~~ (p <= m).
3 Proof.
4 move=> prime_p.
5 apply: contraL.
6 move=> leq_p_m.
7 rewrite dvdn_addr.

```

2 subgoals

```

m, p : nat
pr_p : prime p
p_le_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 $p \leq m$. This can be proved combining the following lemmas: stated by lemma:

```

1 Lemma dvdn_fact m n : 0 < m -> m <= n -> m %| n '!.
2 Lemma prime_gt1 p : prime p -> 1 < p.

```

The first goal can be solve by appealing to the following lemmas:

```

1 Lemma gtnNdvd n d : 0 < n -> n < d -> (d %| n) = false.
2 Lemma prime_gt0 p : prime p -> 0 < p.

```

The resulting script would then be:

```

1 Lemma example m p : prime p -> p %| m '! + 1 -> ~~ (p <= m).
2 Proof.
3 move=> prime_p.
4 apply: contraL.
5 move=> leq_p_m.
6 rewrite dvdn_addr.
7   rewrite gtnNdvd.
8     by []. (* ~~ false *)
9     by []. (* 0 < 1 *)
10    by apply: prime_gt1. (* 1 < p *)
11 apply: dvdn_fact.
12   by apply: prime_gt0. (* 0 < p *)
13 by []. (* p <= m *)
14 Qed.

```

For brevity we report in comments the goals solved by tactics.

We improve such 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. It is good practice to solve such side conditions as soon as possible. The first two side conditions are indeed trivial, and, as we did for `case`, can be solved by `//`. We also combine on the same line the first three step, using the semicolon.


```

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

```

If one carefully compares the conclusion of `gtnNdvd` and `prime_gt1` he can spot they are both rewriting rules. While the former features an explicit “`.. = false`”, in the latter one the “`.. = true`” part is hidden, but is there. This means both lemmas can be used as identities.

Advice

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



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

```

1 Lemma example m p : prime p -> p %| m ' ! + 1 -> ~~ (p <= m).
2 Proof.
3 move=> prime_p; apply: contral; 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 -> ~~ (p <= m).
2 Proof.
3 move=> prime_p; apply: contral; move=> leq_p_m.
4 rewrite dvdn_addr.
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 sub goals for its eventual premises. Hypotheses can be named using `move=>`. The proof script can be further reduced in size, as we describe in section 2.5.1.

2.3.3 Proofs by induction

Lets take the well known induction principle for Peano’s natural numbers and lets formalize it in the language of COQ. It reads: let \mathcal{P} be a property of natural numbers, if \mathcal{P} holds on 0 and if for all natural number n if \mathcal{P} hold on n then it also hold on $n + 1$, 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.

As for the type variable in the declaration of the (polymorphic) data type `seq`, the language of COQ gives to \mathcal{P} a regular status, and induction, instead of being a “schema” becomes is a regular lemma.

```
1 About nat_ind.
```

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

Here P is quantified exactly as n is, but its type is a bit more complex and deserves an explanation. As we have seen in the first chapter the \rightarrow 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. At 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)
```

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

`P 0`

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

Equivalent by computation

```
odd 0 && prime 0
```

Remark the similarity between the function argument to `foldr` that is used to describe the general term of an iterated sum in section 1.7 and the predicate P here used to describe a general property.

While the language of COQ lets one express the induction principle for natural numbers in a general way, the shape of the lemma is tied to such data type. For example the induction principle over sequences is similar, but not identical.

```
1 About list_ind.
```

```
list_ind : ∀ A : Type, ∀ P : list A -> Prop,
  P [] -> (∀ x s, P s -> P (x :: s)) -> ∀ s, P s.
```

At the light of that, 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 provides such functionalities, while `apply` does not. Hence `elim` has to be used when reasoning by induction.

Lets 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
```

We used the `//` switch to rule out the base case, since if m is 0 both sides evaluate to zero. In order to replace use the induction hypothesis we pull out of the sum the `.+1` by rewriting with `addSn`. In this proof by induction the value of P synthesized by `elim` for us is `(fun n : nat => n + 0 = n)`.

Unfortunately proof 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 mention a recursive function that uses an accumulator: its value is going to change during recursive

calls, hence the induction hypothesis must be general.

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

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

where `rcons` is a name for 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 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 .
```

The proof uses the following lemmas:

```
Tools
1 Lemma cats1 T s (z : T) : s ++ [:: z] = rcons s z.
2 Lemma foldr_cat T f (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, `IHs` states:

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

The quantification on `z` is crucial since the goal in the induction step, just before we use `IHs` is the following one:

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

The instance of the induction hypothesis that we need is one where `z` takes `(foldr (fun y w => f w y) z [:: x])`, i.e. the value of the accumulator to `foldr` for a list which last element is `x`.

2.4 Searching the library

Finding the “right” lemma in a library that contains thousands of them may be quite a challenge. Even more since the state of the art of search tools for formal libraries is not as advanced as the one for, say, the world wide web.

The Ssreflect proof language provides the `Search` command to filter the set of available lemmas providing patterns, like `(_ * _ + _)` or `(addn _ _)`, and partial names, like `"rev"` `"cons"`.

2.4.1 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 wild cards, `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, among many other things, the following two lemmas:

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

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

2.4.2 Search by name

Searching by name is the more effective way to find things in the Mathematical Components library. However one needs to be acquainted to the naming schema consistently used to name lemmas. Indeed the name `my_first_lemma` we chose in section 2.1.1 is actually a very bad name: hard to guess what the lemma is about by just reading its name. The name can, and should, convey as much information as possible while striving to stay short and handy. We hence refrain from naming lemmas with numbers, as is typically done in standard mathematical texts.

The naming convention is documented in the reference manual of Ssreflect [10, Section 11]. Let’s get a flavour of it by examples:

- the name of the lemma begins with the name of the main operation involved in the statement: `addn` for `addnA`, `odd` for `odd_mul`, `leq` in `leq0n`.
- In `leq0n`, the other symbol involved in the name is “0”, that indeed plays an important role in the lemma. The relative position of “0” and “n” is

chosen to recall to the user if 0 occurs the left or the right argument of the binary test `leq`.

- The capital letter suffix “A” in `addnA` stands for associativity.

Capital letters in the suffix denote standard properties. In particular “A” stands for associativity and “C” for commutativity.

The `Search` command accepts as filters partial names. 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.

A last remark about naming convention worth making here is that uninteresting and frequently used lemmas must have a short names, so that they can be easily type in and at the same time quickly identified and disregarded when reading a proof.

2.5 Rewrite, a swiss army knife

Approximatively one third of the proof script in the Mathematical Components library is made of invocations of the `rewrite` tactic. Such proof command provides many features we cannot extensively cover here. We just sketch a very common idiom involving conditional rewrite rules and the mention the `RHS` pattern for the casual reader. The interested reader can find more about the pattern language in section 2.5.1 or in the user manual of the `Ssreflect` language [10].

So far we have solved side conditions 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 time. For example, one could consider the second sub goal that follows `rewrite dvdn_addr` a side condition and solve it on the same line.

Lets take the last version of the proof script for this example.

```
1 Lemma example m p : prime p -> p %| m '! + 1 -> ~~ (p <= m).
2 Proof.
3 move=> prime_p; apply: contral; move=> 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.
```

Optionally rewriting with `dvdn_fact` on all goals affects only the side condition, since the main goal mentions no divides predicate. The same holds for `prime_gt0`.

```

1 Lemma example m p : prime p -> p %| m ' ! + 1 -> ~~ (p <= m).
2 Proof.
3 move=> prime_p; apply: contral; move=> 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 [::] = 0.
2 Proof.
3 rewrite /=.

```

```

=====
0 = 0

```

The `/=` flag simply invokes the COQ standard `simpl` tactic. Whilst being handy, `simpl` tends to oversimplify expressions, hence we advice using it with care. In section 4.3.3 we propose a less risky alternative. The sequence “`// /=`” can be collapsed into `//=. index[ssr]rewrite!/=` (simplify close)

Last, the unfolding equation `leqE` that we used in the proof of `leq_mul21` in section 2.2.3 does not exist. The entire class of “equations” linking (the name of) a definition to its body can be accessed in `rewrite` by just prefixing the name of the constant with `/` as in `rewrite /leq`. Indeed *unfolding* a definition is a free operation in COQ, it requires not proof, hence there is no lemma for it.

2.5.1 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 which 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 subterm `(m * n1 <= m * n2)`. Now suppose we want to pick the other instance of 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 term $(n1 <= n2)$, in the right hand-side.

User provided pattern	Proof state
<pre> 1 Lemma leq_mul2l m n1 n2 : 2 (m * n1 <= m * n2) = (m == 0) (n1 <= n2). 3 Proof. 4 rewrite [n1 <= _]leqE.</pre>	<pre> m, n1, n2 : nat ===== (m * n1 <= m * n2) = (m == 0) (n1 - n2 == 0)</pre>

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

RHS pattern	Proof state
<pre> 1 Lemma leq_mul2l m n1 n2 : 2 (m * n1 <= m * n2) = (m == 0) (n1 <= n2). 3 Proof. 4 rewrite [in RHS]leqE.</pre>	<pre> m, n1, n2 : nat ===== (m * n1 <= m * n2) = (m == 0) (n1 - n2 == 0)</pre>

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 [10, 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.

Lemma `leqE` in fact does not exist in the library. Indeed the identity it states holds by symbolic computation, and we have seen in section 2.2.1 that such class of proof steps can be typically omitted. However if we try to omit the first `rewrite !leqE` command, then the next one, namely `rewrite -mulnBr` fails:

<pre> 1 Lemma leq_mul2l m n1 n2 : 2 (m * n1 <= m * n2) = (m == 0) (n1 <= n2). 3 Proof. 4 rewrite -mulnBr.</pre>	<pre> The RHS of mulnBr (_ * _ - _ * _) does not match any subterm of the goal.</pre>
--	---

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

<pre> 1 Lemma leq_mul2l m n1 n2 : 2 (m * n1 <= m * n2) = (m == 0) (n1 <= n2). 3 Proof. 4 rewrite -[_ _]muln_eq0. </pre>	<pre> m, n1, n2 : nat ===== (m * n1 <= m * n2) = (m * (n1 - n2) == 0) </pre>
---	---

Indeed, the `[_ || _]` pattern identifies term `(m == 0) || (n1 <= n2)`, as their head symbols coincide. Now that we have elected a subterm, the `rewrite` tactic is able to identify it with term `(m == 0) || (n1 - n2 == 0)`, itself an instance of the right instance of `muln_eq0`. Note that the `[_ || _]` pattern is redundant here: 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.

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

Focused unfold	Proof state
<pre> 1 Lemma leq_mul2l m n1 n2 : 2 (m * n1 <= m * n2) = (m == 0) (n1 <= n2). 3 Proof. 4 rewrite [in LHS]/leq. </pre>	<pre> m, n1, n2 : nat ===== (m * n1 - m * n2 == 0) = (m == 0) (n1 <= n2) </pre>

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

Refold	Proof state
<pre> 1 Lemma leq_mul2l m n1 n2 : 2 (m * n1 - m * n2 == 0) = 3 (m == 0) (n1 <= n2). 4 Proof. 5 rewrite -/(leq _ _). </pre>	<pre> m, n1, n2 : nat ===== (m * n1 <= m * n2) = (m == 0) (n1 <= n2) </pre>

More generally, the `rewrite` tactic can be used to replace a certain subterm of the goal by an equal by computation one:

Replacing computationally equal terms	Proof state
<pre> 1 Lemma leq_mul2l m n1 n2 : 2 (m * n1 <= m * n2) = (m == 0) (n1 <= n2). 3 Proof. 4 rewrite -[n1]/(0 + n1). </pre>	<pre> m, n1, n2 : nat ===== (m * (0 + n1) <= m * n2) = (m == 0) (0 + n1 <= n2) </pre>

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

2.6 Exercises

Exercise 8. *Truth tables*

Prove the following boolean tautologies:

```
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.
```

Exercise 9. *Rewriting*

Prove the following lemma by rewriting:

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

The $(_ \wedge _)$ notation is attached to the `expn` function.

Exercise 10. * *Induction*

Prove the following lemma by induction:

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

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

Exercise 11. ** *Multiple induction*

Prove the following lemma by induction.

```
1 Lemma size_all_words n T (alphabet : seq T) :
2   size (all_words n alphabet) = size alphabet ^ n.
```

It requires two inductions: first on the length of words, then on the alphabet. In this last case, a non trivial sub expression has to be generalized just before starting the induction.

2.6.1 Solutions

Answer of Exercise 8

```

1 Lemma orbA b1 b2 b3 : b1 || (b2 || b3) = b1 || b2 || b3.
2 Proof. by case: b1; case: b2; case: b3. Qed.
3 Lemma implybE a b : (a ==> b) = ~~ a || b.
4 Proof. by case: a; case: b. Qed.
5 Lemma negb_and (a b : bool) : ~~ (a && b) = ~~ a || ~~ b.
6 Proof. by case: a; case: b. Qed.

```

Answer of Exercise 9

```

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

```

Answer of Exercise 10

```

1 Lemma odd_exp m n : odd (m ^ n) = (n == 0) || odd m.
2 Proof.
3 elim: n => // n IHn.
4 rewrite expnS odd_mul {}IHn orbC.
5 by case: odd.
6 Qed.

```

Answer of Exercise 11

```

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

```

Chapter 3

Type Theory

The Curry-Howard correspondence

By now, the informed reader is likely to wonder when the authors of the book will eventually mention the mathematical foundation of COQ, a variant of type theory called Calculus of Inductive Constructions, and in particular explain the Curry-Howard isomorphism. Indeed this text departs from the standard presentation of COQ, that typically starts by presenting the logic and showing how standard connectives like conjunction or disjunction are defined in term of inductive types.

We made a deliberate choice to put forward the programming aspects of type theory and how programs can be used both to express computable functions and decidable predicates or boolean connectives. Indeed this formalization choice is one of the main ingredients of the Mathematical Components library. The computational behavior of programs is at the core of type theory. The fact that a function computes to a value requires no proof: 5 and 2+3 are indistinguishable from a logical standpoint. This fact provided a powerful form of automation, and as we will see later will also ease the construction of “sub sets”.

Now it is finally time to provide a minimal presentation of the statements as types correspondence. The universe of boolean predicates is in Mathematical Components limited: one cannot for example express a general \exists quantification, and also the status of the \forall quantification has been so far never really explained. This presentation is far from being exhaustive, the interested reader can find a modern presentation of type theory in [\[23\]](#).

3.1 Connectives

3.1.1 Primitive types and terms formers

In type theory we say that *logical statements* correspond to *types* and that *proofs* correspond to *programs*.

The simplest example one could think of is the formal statement saying that a proposition A implies itself, namely $A \rightarrow A$. If one reads the implication symbol \rightarrow as the type of functions, the statements reads as a function from A to A . A program with that type would be `(fun x : A => x)`. Such program takes in input a term x of type A , that we here see as a proof of A , and returns it, i.e. it produces in output a proof of A . We say that such program is a proof of $A \rightarrow A$.

What is striking here is that the same class of terms represents both programs and proofs, for example the identity function of natural numbers has the very same shape of the proof we've just seen. So we are left with only two concepts, types and terms, playing a double role. We now see how types and terms can be formed.

Regarding types, the primitive type formers we have seen are the function space \rightarrow , also logical implication, and the universal quantification \forall , used to describe polymorphic function as well as parametric lemmas (like induction principles). Standard logical rules describing how one can prove formulas involving these connectives are:

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

The former reads: to prove $A \rightarrow B$ one can prove B under the assumption A . The latter: to prove $\forall x, B$ one can prove B assuming that x is fresh. We annotate such rules with the terms corresponding to these proof rules.

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

The anonymous function constructor `(fun .. => ..)` serves as a proof for both rules: the term b is in both cases a proof of B . The only difference is that x is only allowed to occur in B in the second rule. Said otherwise in type theory \rightarrow is a special case of \forall where the bound variable does not occur in the quantified formula. For example there is no semantic difference between `(nat -> bool)` and `(\forall x : nat, bool)`.

Logical connectives come with elimination rules, in particular

$$\frac{A \rightarrow B \quad A}{B} \rightarrow_E \qquad \frac{\forall x, B}{B[t/x]} \forall_E \ (t \text{ a term})$$

Function application serves as a proof for both rules.

$$\frac{f : A \rightarrow B \quad a : A}{(f \ a) : B} \rightarrow_E \qquad \frac{f : \forall x : A, B \quad t : A}{(f \ a) : B[t/x]} \forall_E$$

Here the programs as proofs 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. That is the difference between the *dependent function space* (\forall) and the standard function space (\rightarrow).

Advice

Lemma names can be used as functions and you can pass to them arguments. 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 $(p_pr : \text{prime } p)$.



Finally note that the \forall quantification specifies the type of the bound variable, and that the \forall_E checks the term t has the “right type”, otherwise we get a type error. What is particularly relevant to the Mathematical Components library is the notion of right type, or how types are compared. Indeed the \rightarrow_E rule (and also the \forall_E) should be written as follows.

$$\frac{f : A \rightarrow B \quad a : A' \quad A \equiv A'}{(f \ a) : B} \rightarrow_E$$

We say that A and A' are checked for *convertibility* (\equiv), i.e. if they are equal up to computation. As a consequence the term `(isT : true)` is a perfectly valid proof of $(0 < n.+1)$, as well as `(odd 7)` or `(prime 27)` and can be passed to a lemma expecting any of these.

3.1.2 Inductive types

In the previous chapters we used many data types already. For example we have seen the type `nat` and its constructors `0` and `s`. Through the looking glasses of the Curry-Howard correspondence `0`, being a term of type `nat` can represent a “proof” of `nat`. For data types we prefer to say *inhabited* rather than *proved*, but there is no real difference.

Inductive “data” types can be used to represent logical connectives, in particular the ones we miss so far, lie the existential quantifier.

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

The first rule reads: to prove $A \wedge B$ one needs to prove both A and B . The latter lets one prove A whenever one can prove the stronger $A \wedge B$ statement.

In COQ such connective can be characterized by the following inductive definition.

```
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 only constructor `conj` that can be used to inhabit `and` takes two arguments, a proof of `A` and a proof of `B`, faithfully modelling the logical rule \wedge_I . Note that `A` and `B` are parameters of the inductive definition, similarly to the definition of polymorphic lists or pairs.

It is worth to note that the definition of the pair data type, Exercise 1, is almost identical to the one of `and`.

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

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

Pattern matching provides a way to express the elimination rule for conjunction. In particular the elimination rule is just a projection.

```
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 check the type of the `conj` constructor

```
1 About conj.
```

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

we may wonder what happens if the type of the second argument (i.e. `B`) is made dependent on the value of the first argument (of type `A`). What we obtain is 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)).
```

The `ex_intro` constructor is the only mean to prove a statement like `(exists n, prime n)`. In such case 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 `ex` inductive definition is again pretty similar to the pair, but note that the type of the second component depends on the value of the first one. Last note the parameter `P` that is a function representing an arbitrary predicate over a term of type `A`. Hence `(P a)` is the instance of the predicate to `a`. E.g. the predicate of being an even prime number is expressed as `(fun x : nat => ~~ (odd x) && prime x)`, and the statement expressing the existence of such number is `(ex nat (fun x : nat => ~~ (odd x) && prime x))`.

We quickly see the inductive definition of the disjunction.

```
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 in this case. Each branch represents a distinct sub proof. In this case to prove P starting from $A \vee B$ one has to deal with all possibilities: prove P under the assumption A and prove P under the assumption B .

Typically a logic comes with the \top , \perp and \neg constants. They 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).
```

Remark that to prove `True` one has simply to provide `I` that has no argument. 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`). Conversely 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 are left to describe 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 family 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, line 1. 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 $(a = b)$ it only when a and b are equal up to computation (as in 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 and 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 [23]. Later in this chapter we define and use other inductive families to take advantage of the “automatic” substitution of the implicit equations we see here: while the type of px is $(P\ x)$ it is accepted as an inhabitant of $(P\ y)$ because inside the `match` the term y is automatically replaced by x .

3.1.3 Proof commands

Since proofs are just terms one could, in principle, use no proof language and input directly proof terms. Indeed this was the modus operandi in the pioneering work of De Bruijn on Automath (automating mathematics) in the seventies [17]. Of course a dedicated proof language can relief the user from many tedious details. The proof commands we have seen so far can all be explained in terms of the proof terms they produce behind the scenes. For example `case: n` provides a much more compact syntax for `(match .. with .. end)` and it produces a `match` expression with the right shape by looking at the type of n . If n is a natural number then there are two branches, the one for the `s` constructor carries an argument of type `nat`, the other one is for `o` and binds no additional term. The `case:` tactic is general enough to work with any inductive data type and inductive predicate.

The `apply:` tactic generates an application. For example `apply: addnC` generates the term `(addnC t1 t2)` by figuring out the correct values of $t1$ and $t2$, or opening new goals when this cannot be done, i.e. if the lemma takes in input proofs, like `contraL`.

There is a list of proof commands that are shorthands for `apply:` and is only worth mentioning here briefly. `split` proves a conjunction by applying the `conj` constructor, `left` and `right` prove a disjunction by applying `or_introl` and `or_intror` respectively. `exist t` proves an existentially quantified formula by providing the witness t and, later, a proof that t validates the predicate. Finally `reflexivity` proves an equality by applying `erefl`.

3.2 Managing the proof context

The only primitive constructor that remains without an associated proof command is `(fun .. => ..)`. Operationally what the \rightarrow_I and \forall_I logical rule do is to

introduce into the proof context a new entry. So far we either expressed this step at the beginning of proofs by putting such items just after the name of the lemma being prover, or just after a `case:` or `elim:` proof command. The current section expands this subject covering the full management of the proof context.

3.2.1 The stack model

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 deal with 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 action correspond to regular, named, proof commands. It is the use one makes of them that may be twofold: a case analysis in the middle of a proof may start two distinct lines of reasoning, and hence it is worth being noted explicitly with the `case` word; conversely de-structuring a pair to obtain the two components can hardly be a relevant step in a proof, so one may prefer to perform such bookkeeping action with a symbolic, compact, notation corresponding to the same `case` functionality.

Pulling from the stack

Lets 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 goal assumption or quantified variable. 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 out of the stack an item and names it. The `move` proof command is the one that does nothing. We can use it as a placeholder for the postfix `=>` bookkeeping action.

```
1 move=> xy pr_x odd_y
```

```
xy : nat * nat
pr_x : prime xy.1
odd_y : odd xy.2
=====
2 < xy.2 + xy.1
```

Now, en passant, one would like to decompose xy into its first and second component. Instead of the verbose $\Rightarrow xy$; **case**: $xy \Rightarrow x y$ one can use the symbolic notation $[]$ to perform such action.

```
1 move=> [x y] pr_x odd_y
```

```
x, y : nat
pr_x : prime (x,y).1
odd_y : odd (x,y).2
=====
2 < (x,y).2 + (x,y).1
```

One can place the $/=$ switch to force the system to reduce the formulas on the stack, before introducing them in the context, and obtain:

```
1 move=> [x y] /= pr_x odd_y
```

```
x, y : nat
pr_x : prime x
odd_y : odd y
=====
2 < y + x
```

One can also process an assumption through a lemma. For example `prime_gt1` states $(\text{prime } p \rightarrow 1 < p)$ 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
```

```
x, y : nat
x_gt1 : 1 < x
odd_y : odd y
=====
2 < y + x
```

The leading $/$ makes `prime_gt1` do 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 and name assigned to its output.

One could also examine y : it can't be 0, since it would contradict the assumption saying that y is odd.

```
1 move=> [x [//|y]] /= /prime_gt1-x_gt1.
```

```
x, y : nat
x_gt1 : 1 < x
=====
~~ odd y -> 2 < y.+1 + x
```

This time the destruction of y generates to cases, hence the $[\dots | \dots]$ syntax mentioning the two branches. In the first one, when y is 0, the $//$ action solves the goal, by the same trivial means of the **by** $[]$ terminator. In the second branch we name y the new variable.

Now, the fact that y is even is not needed to conclude, so we can discard it using the $_$ dummy name.

```
1 by move=> [x [//|y]] /= /prime_gt1-x_gt1 _; apply: ltn_add1 x_gt1.
```

The way to dispose an already named assumption is to mention its name in curly braces, as $\Rightarrow \{x_gt1\}$.

We finally conclude with the **apply**: command that here we use with two arguments: a function and its last argument.

1 **About** ltn_add1.

ltn_add1 : $\forall m\ n\ p : \text{nat}, m < n \rightarrow m < p + n$

apply: will fill in the blanks between the function (the lemma name) and the argument provided. Note that by passing `x_gt1`, the variable `m` picks the value 1. The conclusion of `ltn_add1` hence unifies with $(2 < y.+1 + x)$ because both `+` and `<` are defined as programs that compute: addition exposes a `.+1` by reducing to $2 < (y+x).+1$, then `<`, or better the underlying `<=`, eats a successor from both sides, leading to $1 < y + x$ that looks like the conclusion of the lemma we apply.

Here we have shown all possible actions one can perform in an intro pattern, squeezing the entire proof into a single line. This has to be seen both as an opportunity and as a danger: one can easily make a proof unreadable by performing too many actions in the bookkeeping operator `=>`. At the same time a trivial sub-proof like this one should take no more than a line, and in that case one typically sacrifices readability in favor of compactness: what would you learn by reading a trivial proof? Of course, finding the right balance only comes with experience.

Warning

The case intro pattern `[...|...]` obeys an exception: when it is the first item of an intro pattern, it does not perform a case analysis, but only branch on the subgoals.^a Indeed in `case: n => [|m]` only one case analysis is performed.



^anot the status quo, but I prefer this exception to the current one

Working on the stack

The stack can also be used as a work place. Indeed there is no need to pull from the stack all items. If we take the previous example

```
=====
 $\forall xy, \text{prime } xy.1 \rightarrow \text{odd } xy.2 \rightarrow 2 < xy.2 + xy.1$ 
```

and we stop just after applying the view, we end up in a valid situation.

1 **move=>** [x y] /= /prime_gt1.

```
x, y : nat
=====
1 < x -> odd y -> 2 < y + x
```

One can also chain multiple views on the same stack item.

1 **move=>** [x y] /= /prime_gt1/ltnW.

```
x, y : nat
=====
0 < x -> odd y -> 2 < y + x
```

Two other operations are available on the top stack item: specialization and substitution. Let's take the following conjecture.

```
=====
(∀ n, n * 2 = n + n) -> 6 = 3 + 3
```

The top stack item is a quantified assumption. To specialize it to, say, 3 one can write as follows:

```
1 move=> /(_ 3).
```

```
=====
3 * 2 = 3 + 3 -> 6 = 3 + 3
```

The idea behind the notation is that when applies a view to the top stack item, as in `/v`, he is forming the term $(v \text{ top})$, while when one specializes the top assumption as in `/(_ x)` he is forming the term $(\text{top } x)$. The `_` is a place holder for the top item, and is omitted in `/(view _)`.

When the top stack item is an equation one can substitute it using `<-` and `->` for right-to-left and left-to-right respectively.

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

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

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

Pushing to the stack

We have seen how to pull items from the stack to the context. Now let's see how to perform the converse via the `:` operator, also called *discharging* operator. Such operator decorates proof commands as `move`, `case` and `elim` with actions to be performed before the command is actually run.

Warning

The colon symbol in `apply:` is not the discharging operator. It is just a marker to distinguish the `apply:` tactic of `Ssreflect` from the `apply` tactic of `COQ`. Indeed the two tactics, while playing similar roles, behave very differently.



Imagine we want to perform case analysis on `y` at this stage

```
x, y : nat
x_gt1 : 1 < x
odd_y : odd y
=====
2 < y + x
```

Let's dissect the command `case: y`. It is equivalent to `move: y; case`, where `move` once again is a place holder, `:` `y` pushes onto the stack the `y` variable and `case` operates on the top item of the stack. Pushing items on the stack is called *discharging*.

Just before running `case` the goal looks like this:

```
x : nat
x_gt1 : 1 < x
odd_y : odd y
=====
∀ y, 2 < y + x
```

Unfortunately the binding for `y` is needed by the `odd_y` context item, so `case: y` fails. One has to push on the stack items in a valid order. `case: y odd_y` would push `odd_y` first, then `y`, leading to

```
x : nat
x_gt1 : 1 < x
=====
∀ y, odd y -> 2 < y + x
```

Via the execution of `case` one obtains:

```
2 subgoals

x : nat
x_gt1 : 1 < x
=====
odd 0 -> 2 < 0 + x

subgoal 2 is:
∀ n : nat, odd n.+1 -> 2 < n.+1 + x
```

An alternative to discharging `odd_y` would be to clear it, i.e. purge it from the context. Listing context entry names inside curly braces has this effect. For example `case: y {odd_y}`.

One can combine `:` and `=>` around a proof command, to first prepare the goal for its execution and finally apply the necessary bookkeeping to the result. For example:

```
1 case: y odd_y => [|y']
```

```
2 subgoals

x : nat
x_gt1 : 1 < x
=====
odd 0 -> 2 < 0 + x

subgoal 2 is:
odd y'.+1 -> 2 < y'.+1 + x
```

At the left of the `:` operator one can also put a name for an equation that links the term at the top of the stack before and after the tactic execution. `case E: y odd_y => [|y']` leads to the two sub goals:

```

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 discharge the goal with any term, not just variables occurring in it. For example “`move: (leqnn 7)`” pushes on the stack the additional assumption $(7 \leq 7)$. This will come handy in section 3.3.1.

3.3 Inductive reasoning

In chapter 1 we have seen the `fun` construction of functions and inductive data types like `nat` as well as the pattern matching. All these constructions have found a correspondence in the Curry-Howard correspondence. The only missing piece is recursive programs. For example `addn` was written by recursion on its first argument, and is a function taking in input two numbers and producing a third one. We can write programs by recursion that take in input, among regular data, proofs and produce in output other proofs. Let’s look at the induction principle for natural numbers through the looking glasses of the Curry-Howard isomorphism.

```
1 About nat_ind.
```

```

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

```

`nat_ind` is a program that produces a proof of $(P\ n)$ for any n proviso a proof for the base case $(P\ 0)$, and a proof of the inductive step $(\forall n : \text{nat}, P\ n \rightarrow P\ n.+1)$. Let’s write such program.

```

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 such program when the `nat` data type is defined. Recall that recursive functions are checked for termination: through the lenses of the proofs as programs correspondence this means that the induction principle just coded is sound, i.e. based on a well founded order relation.

If non-terminating functions are not ruled out, it 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 loosing 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, called *positivity condition*, is out of scope for this book. The interested reader shall refer to [22].

3.3.1 Strong induction

As an exercise we show how the `elim` tactic combined with the bookkeeping operator `:` lets one perform, on the fly, a stronger variant of induction called “course of values”.

Claim: every amount of postage that is at least 12 cents can be made from 4-cent and 5-cent stamps. The proof in the inductive step goes as follows. There are obvious solutions for a postage between 12 and 15 cents, so we can assume it is at least 16 cents. Since the postage amount is at least 16, by using a 4-cent stamp we are back at a postage amount that, by induction, can be obtained as claimed. \square

The tricky step is that we want to apply the induction hypothesis not on $n-1$, as usual, but on $n-4$, since we know how to turn a solution for a stamping amount problem n to one for a problem of size $n+4$ (by using a 4-cent stamp). The induction hypothesis provided by `nat_ind` is not strong enough. However we can use the `:` operator to load the goal before performing the induction.

```

1 Lemma stamps n : 12 <= n -> exists s4 s5, s4 * 4 + s5 * 5 = n.
2 Proof.
3 elim: n {-2}n (leqn n) =>[|n IHn|]; first by case.
4 do 12! [ case; first by [] ]. (* < 12c *)
5 case; first by exists 3, 0. (* 12c = 3 * 4c *)
6 case; first by exists 2, 1. (* 13c = 2 * 4c + 1 * 5c *)
7 case; first by exists 1, 2. (* 14c = 1 * 4c + 2 * 5c *)
8 case; first by exists 0, 3. (* 15c = 3 * 5c *)
9 move=> m'; set m := _.+1; move=> mn m11.
10 case: (IHn (m-4) _ isT) => [|s4 [s5 def_m4]].
11 by rewrite leq_subLR (leq_trans mn) // addSnnS leq_addl.
12 by exists s4.+1, s5; rewrite mulSn -addnA def_m4 subnKC.
13 Qed.

```

Just before the induction step the goal is the following one:

```

1 ∀ n m : nat,
2   m <= n -> 11 < m -> exists s4 s5 : nat, s4 * 4 + s5 * 5 = m

```

The corresponding induction hypothesis `IHn` is:

```

1 IHn : ∀ m : nat,
2   m <= n ->
3   11 < m -> exists s4 s5 : nat, s4 * 4 + s5 * 5 = m

```

Such hypothesis is accessible for each `m` that is at least 12 and smaller than `n`.

Loading the goal works as follows: first `(leqnn n)` is pushed on the stack, then all occurrences of `n` but for the second one are discharged obtaining

```
1  ∀ m : nat,
2    m <= n -> 11 < m -> exists s4 s5 : nat, s4 * 4 + s5 * 5 = m
```

Finally the last occurrence of `n` is discharged too so that induction is performed on such quantified variable.

Lines 4, 9 and 10 deserve a few comments. Line 4 repeats a tactic 12 times. Line 9 uses the `set` proof command to name `m` a term of the form `_.+1` where `_` is a wildcard. It names `m` the term `m'.+16`. Line 10 passes to the induction hypothesis `(m-4)`, then a placeholder for a missing proof of `(m-4 < n)`, finally a proof that `(11 < m-4)`. The introduction pattern in line 10 names `s4` and `s5` the quantities of 4-cents and 5-cents stamps needed to cover the amount of postage `(m-4)`.

3.4 On the status of Axioms

Not all valid reasoning principles can be represented by programs. For example excluded middle can be proved by a program only when the property is decidable, i.e. when we can write in CoQ a program to `bool` that tests if the property holds or not. Excluded middle, in its generality, can only be *axiomatized*, i.e. assumed globally.

The Mathematical Components library is axiom free. This makes the library compatible with as many axioms as possible. Indeed not all combinations of well known, consistent, axioms are consistent, so if a library picks one it rules out another one. Moreover the formalization style we adopted systematically expresses all decidable properties as boolean tests, hence one rarely notices that excluded middle is “missing”.

Sometimes axioms like excluded middle can be confined into “boxes” one can open only in contexts where they can be actually proved. For example:

```
1  Definition classically P : Prop := ∀ b : bool, (P -> b) -> b.
2  Lemma classic_EM P : classically (decidable P).
3  Lemma classic_bind P Q :
4    (P -> classically Q) -> classically P -> classically Q.
```

Here `classically` plays the role of a box, typically called *monad* in computer science, that one can only open when the goal is a boolean, hence decidable, predicate. Inside such box the excluded middle is made available by combining `classic_EM` and `classic_bind`. Nevertheless, when proving a statement that is not a boolean, like `exists n, ...`, one cannot access assumptions in the `classically` box.

In other cases axioms can be avoided by rephrasing the mathematics in a weaker setting. A notable example in the Mathematical Components library is the construction of the real closure of an Archimedean field [5].

3.5 Terms, types, proofs: all in the same pot

For a newcomer, a quite confusing feature of the Calculus of Inductive Constructions is that the same syntax is shared among term, proofs, statements and data types. In other words terms can occur in types and types can occur in terms. And of course terms and types play the double role of proofs and statements respectively. This is strictly related to the range the quantifications are allowed on, and to the fact that they can be used to form types. We have seen quantification on terms, like “ $\forall n : \text{nat}$ ” and on types like “ $\forall P : \text{Prop}$ ” or “ $\forall T : \text{Type}$ ”. The subtle difference between the label `Prop` for propositions and `Type` for data types is out of scope here, we consider them synonyms.

An excellent example of these quantification is the inductive type modelling the existential quantification:

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

The `ex` type constructor is polymorphic in a type `A` and a predicate on `A`. Remark that also `seq` is polymorphic, but only in one argument so the quantification can be non dependent: (`seq : Type \rightarrow Type`).

In the constructor `ex_intro` the quantification on `P` also needs to be dependent: first because it must build a term of type (`ex A P`) (that mentions `P`); second because the last argument needs to be of a proof of `P`, so its type must mention the predicate. Again in the case of `seq` the constructor (`cons : $\forall A, A \rightarrow \text{seq } A \rightarrow \text{seq } A$`) needs no dependent quantification on the head element, while here `a` needs to occur in the type of the argument that follows it, since it must be a proof of (`P a`).

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

```
1 matrix : Type  $\rightarrow$  nat  $\rightarrow$  nat  $\rightarrow$  Type.
2 mulmx :  $\forall R : \text{Type}, \forall m n p : \text{nat},$ 
3   matrix R m n  $\rightarrow$  matrix R n p  $\rightarrow$  matrix R m p.
```

This time the data type of matrices exposes the size, and matrix multiplication can be given a type that rules out incompatible matrices and also describes the dimension of the resulting matrix in terms of the size of the input ones.¹ This last use of of the `forall` quantifier is a classic example of what is called a *dependent type*: a data type depending on data, two natural numbers here.

Last, a quantification can also range on the proofs of a statement so that one can talk about proofs. Such capability finds a rare, but crucial use in Mathematical Components that we discuss in chapter 6.

```
1 Lemma bool_irrelevance (P Q : bool) :  $\forall e1 e2 : P = Q, e1 = e2.$ 
```

¹In veritas, `mulmx` also needs to take in input operations to add and multiply elements of `R`. Such detail plays no role in the current discussion.

Chapter 4

Boolean reflection

Towards real proofs

At this stage, we are in presence of one of the main issues in the representation of mathematics in a formal language: very often, several datastructures can be used to represent a same mathematical definition or statement. But this choice may have a significant impact on the upcoming layers of formalized theories. We have seen so far 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 s' else false.
```

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

```
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, as for instance in order to establish that `(has odd [:: 3; 2; 7]) = true`, we only need to observe that the left hand-side *computes* to `true`.

It is not possible to perform a similar case analysis in a proof using the alternative version `(s_has_aP : has_prop a x0 s)`, since, as sketched in section 3.4, 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 $n : \text{nat}$ and the fact $(\text{asn} : a (\text{nth } x0 \text{ s } n))$. In order to establish that $(\text{has_prop } a \ x0 \ s)$ we cannot resort to computation, but we can on the other hand 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 (contrapositive, reductio ad absurdum,...). They furthermore provide a form of small step automation by computation¹. Specifications in the `Prop` sort are structured logical statements, that can be “destructured” to provide witnesses (of existential statements), instances (of universal statements), subformulae (of conjunctions),... They are proved by deduction, building proof trees made with the rules of the logic. Formalizing a predicate by the 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 $(\text{prime} : \text{nat} \rightarrow \text{bool})$ 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, providing enough infrastructure to ease the bureaucracy of navigating between the two. The `is_true` coercion, which we have been using since the early pages of chapter 2, is in fact one ingredient of this infrastructure.

The `is_true` coercion

```
1 Definition is_true (b : bool) : Prop := b = true.
2 Coercion is_true : bool ->> Sortclass.
```

The `is_true` function is automatically inserted by Coq to turn a boolean value into a `Prop`, i.e. into a regular statement of a theorem. More on this mechanism will be told in section 4.4.

4.1 Views

4.1.1 Relating statements in `bool` and `Prop`

How to best formalize the equivalence between a boolean value b and a statement $P : \text{Prop}$? The most direct way would be to use the conjunction of the two

¹They moreover allow for proof-irrelevant specifications. This feature is largely used throughout the Mathematical Components library but beyond the scope of the present chapter: it will be the topic of chapter 6.

converse applications:

```
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 $(b = \text{false} \leftrightarrow \sim P)$. The following proof script illustrates the kind of undesirable bureaucracy entailed by this wording:

```
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.
```

Last goal

```
1 subgoal
P : Prop
hlr : false = true -> P
hrl : P -> false = true
=====
P \/\ ~ P
```

We could try alternate 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 better solution is to use an ad-hoc inductive definition that resembles a disjunction of conjunctions: we inline the two constructors of a disjunction and each of these constructors has the two arguments of the conjunction's single constructor:

```
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, see Exercise 12.

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.
```

```
b : bool
P : Prop
=====
P -> b = true -> P \/\ ~ P

subgoal 2 is:
~ P -> b = false -> P \/\ ~ P
```

A simple case analysis on the hypothesis `(reflect P b)` exposes in each branch both versions of the statement. Note that the actual `reflect` predicate defined in the `ssrbool` library is slightly different from the one we give here: this version misses an ultimate refinement that will be presented in section 4.2.1.

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

```

1 Lemma andP b1 b2 : reflect (b1 /\ b2) (b1 && b2).
2 Proof. by case: b1; case: b2; [ left | right => //=[ [1 r]] ..]. Qed.
3
4 Lemma orP b1 b2 : reflect (b1 \/ b2) (b1 || b2).
5 Proof.
6 case: b1; case: b2; [ left; by [ move | left | right ] .. ].
7 by right=> // [ [1|r]]].
8 Qed.
9
10 Lemma implyP b1 b2 : reflect (b1 -> b2) (b1 ==> b2).
11 Proof.
12 by case: b1; case: b2; [ left | right | left ..] => //=( _ isT).
13 Qed.

```

Observe that the proof of each of these lemmas is a simple inspection by case analysis of the truth table of the boolean formula. The “;[t1 | t2 .. | tn]” syntax lets one specify a different tactic for each subgoal (see [22, section 9.2]).

More generally, a theorem stating an equivalence between a boolean expression and a `Prop` statement is called a *view*, since it is typically used to view an assumption from a different perspective.

Advice

View names always end with a capital `P`.



Next section is devoted to the proof and usage of more involved views.

4.1.2 Proving views

Views are also used to specify types equipped with a *decidable equality*, by showing that the equality predicate `eq` (seen in section 3.1.2) 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.

```

```

iffP : ∀ (P Q : Prop) (b : bool),
  reflect P b -> (P -> Q) -> (Q -> P) -> reflect Q b

```

Lemma `iffP` indeed relates two equivalences `(reflect P b)` and `(reflect Q b)` involving a 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 view is called `idP` and is seldom used in conjunction with `iffP`.

```
1 Lemma idP (b : bool) : reflect b b.
```

We can now come back to the proof of lemma `eqnP`, and start its proof script by applying `iffP`.

```
1 Lemma eqnP (n m : nat) :
2   reflect (n = m) (eqn n m).
3 Proof.
4 apply: (iffP idP).
```

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

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

The proof is now an easy induction, and is left as Exercise 13.

Let us now showcase the usage of the more general form of `iffP` by proving that a type equipped with an injection in type `nat` has a decidable equality:

```
1 Lemma nat_inj_eq T (f : T -> nat) x y :
2   injective f -> reflect (x = y) (eqn (f x) (f y)).
```

The equality decision procedure indeed just consists in pre-applying the injection `f` to the decision procedure `eqn` available on type `nat`. Since we already know that `eqn` is a decision procedure for equality, we just need to prove that `(x = y)` if and only if `(f x = f y)`, which follows directly from the injectivity of `f`. Using `iffP`, a single proof command splits the goal into two implications, replacing on the fly the evaluation `(eqn (f x) (f y))` by the `Prop` equality `(f x = f y)`:

```
1 Lemma nat_inj_eq T (f : T -> nat) x y :
2   injective f ->
3   reflect (x = y) (eqn (f x) (f y)).
4 Proof.
5 move=> f_inj.
6 apply: (iffP eqnP).
```

```
T : Type
f : T -> nat
f_inj : injective f
x, y : T
=====
x = y -> f x = f y

subgoal 2 is:
f x = f y -> x = y
```

Note that `eqn`, being completely specified by `eqnP`, is not anymore part of the picture. Finishing the proof is left as Exercise 14.

The latter example illustrates the convenience of combining an action on a goal, here breaking an equivalence into one subgoal per implication, with a change of viewpoint, here by the means of the `eqnP` view. This combination of atomic proof steps is pervasive in a library designed using the boolean reflection methodology: the `Ssreflect` tactic language lets one use view lemmas freely in the middle of intro-patterns.

4.1.3 Using views in intro patterns

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*:

<pre> 1 Lemma example n m k : k <= n -> 2 (n <= m) && (m <= k) -> n = k. 3 Proof. 4 move=> lekn /andP.</pre>	<pre> n, m, k : nat lekn : k <= n ===== n <= m /\ m <= k -> n = k</pre>
--	---

The view intro-pattern `/andP` has *applied* 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, view `andP` does not have the shape of an implication, that 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 adaptor.

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))`.

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

Term `(elimTF andP top)` hence has type

<pre> 1 if true then (n <= m) /\ (m <= k) else ~ ((n <= m) /\ (m <= k))</pre>

which reduces to `((n <= m) /\ (m <= k))` since `} 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:

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

As `(n <= m)` is by definition `(n - m == 0)`, we can use the view `eqnP` in order to transform this hypothesis into a proper equation. Observe the new shape of the `lekm` hypothesis:

<pre> 1 Lemma example n m k : k <= n -> 2 (n <= m) && (m <= k) -> n = k. 3 Proof. 4 move=> lekn /andP [/eqnP lekm lem].</pre>	<pre> n, m, k : nat lekn : k <= n lekm : n - m = 0 lemk : m <= k ===== n = k</pre>
---	--

Combining wisely the structured reasoning of inductive predicates in `Prop` with the ease to reason by equivalence via rewriting of boolean identities leads

to concise proofs.

Let us now move to a non artificial example to see how the `Ssreflect` tactic language supports the combination of views with the `apply`, `case` and `rewrite` tactics.

4.1.4 Using views with tactics

We dissect the proof that `<=` is a total relation. As usual the statement is expressed as a boolean formula:

```
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:

<pre>1 Lemma leq_total m n : (m <= n) (m >= n). 2 Proof. 3 rewrite -implNb.</pre>	<pre>m, n : nat ===== ~~ (m <= n) ==> (n <= m)</pre>
--	---

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

<pre>1 Lemma leq_total m n : (m <= n) (m >= n). 2 Proof. 3 rewrite -implNb -ltnNge.</pre>	<pre>m, n : nat ===== (n < m) ==> (n <= m)</pre>
--	---

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:

<pre>1 Lemma leq_total m n : (m <= n) (m >= n). 2 Proof. 3 rewrite -implNb -ltnNge; apply/implP.</pre>	<pre>m, n : nat ===== (n < m) -> (n <= m)</pre>
---	--

We can now conclude the proof:

```
1 Lemma leq_total m n : (m <= n) || (m >= n).
2 Proof. by rewrite -implNb -ltnNge; apply/implP; apply: ltnW. Qed.
```

The `case` tactic also combines well with the view mechanism, which eases reasoning by cases along a disjunction expressed with a boolean statement, like the just proved `leq_total`. For example we may want to start the proof of the following lemma by distinguishing two cases: `n1 <= n2` and `m2 <= 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 2 is:
(m <= maxn n1 n2) = (m <= n1) || (m <= n2)

```

Even if it is not displayed here, subgoal 2 has $(n1 \leq n2)$ in its context.

Finally, the `rewrite` tactic also handles views that relate a 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 $(\text{maxn_idP1 } le_n21)$ 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` is quite interesting and will be detailed in section 4.3.2

4.2 Advanced, practical, statements

As we have already hinted previously the shape of a lemma is very important. There are many ways to express the same concept, but, unsurprisingly, one can be easier to access. In particular there are classes of lemmas that *specify* a concept, and as a consequence shape the proofs involving such notion.

4.2.1 Inductive specs with indices

What we did for `reflect`, an ad hoc connective, to model a line of reasoning is a recurrent pattern in the Mathematical Components library. Such class of inductive predicates is called “spec”, for *specification*.

Spec predicates are inductive families with indexes, exactly as the `eq` predicate seen in section 3.1.2. In particular their elimination rule encapsulate the

notion of substitution, and that operation is performed automatically by the logic.

If we look at `reflect`, and its use, one is very likely to substitute `b` for its value in each branch of the case.

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

This alternative formulation makes the equation implicitly stated and also automatically substituted during case analysis.

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

Here the second argument of `reflect` is said to be an *index* and it is allowed to vary depending on the constructor: `ReflectT` always builds a term of type `(reflect P true)` while `ReflectF` builds a term of type `(reflect P false)`. When one reasons by cases on a term of type `(reflect P b)` he obtains two proof branches, in the first one `b` is replaced by `true`, since it corresponds to the `ReflectT` constructor. Conversely, `b` is replaced by `false` in the second branch.

Let's take an example where we reason by cases on the `andP` lemma, that states $\forall a b, (\text{reflect } (a \wedge b) (a \&\& b))$.

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

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

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

Remark how the automatic substitution trivializes the second goal. The first one can be solved by replacing both `a` and `b` by their truth values, once `ab` is destructed. Hence the full proof script is:

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

Note that we have not specified a value for the variables quantified in the statement of `andP`. Indeed such view is accompanied by an implicit argument declaration as follows:

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

We recall that this makes `a` and `b` implicit, hence writing `andP` is equivalent to `(@andP _ _)` whose type is `(reflect (_ /\ _) (_ && _))`. The value of the index of the inductive family to be replaced by `true` or `false` is here a pattern `(_ && _)` and the goal is searched for an instance of such pattern by the same matching algorithm the `rewrite` tactic uses for rewrite rules. Tuning of implicit arguments is key to obtain easy to use lemmas.

If one needs to override the pattern inferred for the index of `andP`, he can provide one 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 [10, 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 view worth mentioning is `leqP` that replaces, in one shot, both `(_ <= _)` and the converse `(_ < _)` by opposite truth values. Sometime a proof works best by splitting into three branches, i.e. separating the equality case. The `ltngtP` lemma is designed for that.

In practice lines of reasoning consisting in a specific branching of a proof can often be modelled by an appropriate spec lemma.

Advice

The structure of the proof shall not be driven by the syntax of the definition/predicate under study but by the view/spec used to reason about it



4.3 Real proofs, finally!

So far we’ve only tackled simple lemmas, most of them did admit a one line proof. When proofs get longer *structure* is the best ally in making them readable and maintainable. Structuring proofs means identifying intermediate results, factor similar lines of reasoning (e.g. symmetries), signal crucial steps to the reader, and so on. In short, a proof written in Coq should not look too different from a proof written on paper.

The first subsection introduces the `have` tactic, that is the key to structure proofs into intermediate steps. The second subsection deals with the “problem” of symmetries. The third ones uses most of the techniques and tactics seen so far to prove correct the Euclidean division algorithm. The three subsections contain material in increasing order of difficulty.

4.3.1 Primes, a never ending story

Saying that primes are infinite can be phrased as: for any natural number m , there exists a prime greater than m . The proof of this claim goes like that: every natural number greater than 1 has at least one prime divisor. If we take $m! + 1$, then such prime divisor p can be shown to be greater than m as follows. By contra position we assume $p \leq m$ and we show $p \nmid m! + 1$. Being smaller than m , $p \mid m!$, hence to divide $m! + 1$, p should divide 1, that is not possible since p is prime, hence greater than 1. \square

We first show that any positive number smaller than n divides $n!$.

```
1 Lemma dvdn_fact m n : 0 < m <= n -> m %| n'!.
2 Proof.
3   case: m => // = m; elim: n => // = n IHn; rewrite lt_nS leq_eqVlt.
4   by case/orP=> [/eqP-> | /IHn]; [apply: dvdn_mulr | apply: dvdn_mull].
5 Qed.
```

After the first line the proof state is the following one:

```
m, n : nat
IHn : m < n -> m.+1 %| n'!
=====
(m == n) || (m < n) -> m.+1 %| (n.+1)'
```

The case analysis rules out the case $(m = 0)$, and simplifies the hypothesis to $(m <= n)$. Recall that $(x <= y <= z)$ is a notation for $((x <= y) \&\& (y <= z))$, hence when the first inequality evaluates to true (e.g. when x is 0) the conjunction simplifies to the second conjunct. The `leq_eqVlt` rewrite rule rephrases `<=` as a disjunction. The capital `v` letter in the lemma name signals a disjunction.

When we reason by cases on the top assumption, line 4, we face two goals, both easy to solve if we look at the development of the factorial of $n.+1$, i.e. $(n.+1 * n'!)$. The former amounts to show that $(n.+1 \%| n.+1 * n'!)$, while the latter to show that $(m.+1 \%| n.+1 * n'!)$ under the (inductive) hypothesis that $(m.+1 \%| n'!)$.

What is paradigmatic in this little proof is the use of the *goal stack* as a

work space. In other words the proof script would be much more involved if we started by introducing in the proof context all assumptions.

We can now move to the proof of the main result. We state it using a “synonym” of the exists quantifier that is specialized to carry two properties. This way the statement is simpler to destruct: with just one case analysis we obtain the witness and the two properties. We also resort to the following lemmas.

Tools

```
1 Lemma fact_gt0 n : 0 < n'!.
2 Lemma pdivP n : 1 < n -> exists2 p, prime p & p %| n,
3 Lemma dvdn_addr m d n : d %| m -> (d %| m + n) = (d %| n).
4 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, that let 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. Since the proof is so short, we will put it on the same line, and remove the full stop.

The next step is to use the `pdivP` lemma to gather a prime divisor of $m'! + 1$. We end up with the following, rather unsatisfactory, script.

```
1 Lemma prime_above m : exists2 p, m < p & prime p.
2 Proof.
3 have m1_gt1: 1 < m'! + 1 by rewrite addn1 ltnS fact_gt0.
4 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 \Rightarrow operator, for example a view application. At 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 /pdvP[p pr_p p_dv_m1]: 1 < m'! + 1 by rewrite addn1 ltnS fact_gt0.
4 exists p => //; rewrite ltnNge; apply: contraL p_dv_m1 => p_le_m.
5 by rewrite dvdn_addr ?dvdn_fact ?prime_gt0 // gtnNdvd ?prime_gt1.
6 Qed.
```

Here the first line obtains a prime `p` as desired, the second one begins to shows it fits by contrapositive reasoning, and the third one, already commented

in section 2.3.2, concludes.

A motto accompanying the proof script style we propose that is that a full line should represent a meaningful reasoning step (for a human being).

4.3.2 Order and max, a matter of symmetry

It is quite widespread in paper proofs to appeal to the reader's intelligence pointing out that a missing part of the proof can be obtained by symmetry. The worst thing one can do when formalizing such an argument on a computer is to use the worst invention of computer science: copy-paste. The language of Coq is sufficiently expressive to model symmetries, and the Ssreflect proof language provides facilities to write symmetric arguments.

We prove the following characterization of the max of two natural numbers.

$$\forall n_1, n_2, m, \quad m \leq \max(n_1, n_2) \Leftrightarrow m \leq n_1 \text{ or } m \leq n_2$$

The proof goes like that: 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` that is an implication.

```
Tools
1 Lemma orb_idr (a b : bool) : (b -> a) -> a || b = a.
2 Lemma maxn_idPl {m n} : reflect (maxn m n = m) (n <= m).
3 Lemma leq_total m n : (m <= n) || (n <= m).
```

Our first attempt takes no advantage of the symmetry argument: we reason by cases on the order relation, we name the resulting fact on the same line (it eases tracking where things come from) and we solve the two goals independently.

```
1 Proof.
2 case/orP: (leq_total n2 n1) => [le_n21|le_n12].
3   rewrite (maxn_idPl le_n21) orb_idr // => le_mn2.
4   by apply: leq_trans le_mn2 le_n21.
5   rewrite maxnC orbC.
6   rewrite (maxn_idPl le_n12) orb_idr // => le_mn1.
7   by apply: leq_trans le_mn1 le_n12.
8   Qed.
```

After line 2, the proof status is the following one:

Output line 2

```

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

subgoal 2 is:
(m <= maxn n1 n2) = (m <= n1) || (m <= n2)

```

The first goal is simplified by rewriting with `maxn_idP1` (as we did in section 4.1.4). Then `orb_idr` trivializes the main goal and generates a side condition with an extra hypothesis we name `le_mn2`.

Output before line 3

```

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

subgoal 2 is: ...

```

Output after 3

```

2 subgoals
m, n1, n2 : nat
le_n21 : n2 <= n1
le_mn2 : m <= n2
=====
m <= n1

subgoal 2 is: ...

```

Line 4 combines by transitivity the two hypotheses to conclude. Since it closes the proof branch we use the prefix `by` to asserts the goal is solved and visually signal the end of the paragraph. Line 5 commutes `max` and `||`. We can then conclude by copy-paste.

To avoid copy-pasting, shrink the proof script and finally make the symmetry step visible we can resort to the `have` tactic. In this case the statement is a variation of what we need to prove. Remark that as for `Lemma`, we can place parameters, `x` and `y` here, before the `:` symbol.

```

1 Lemma leq_max m n1 n2 : (m <= maxn n1 n2) = (m <= n1) || (m <= n2).
2 Proof.
3 have th_sym x y: y <= x -> (m <= maxn x y) = (m <= x) || (m <= y).
4   move=> le_n21; rewrite (maxn_idP1 le_n21) orb_idr // => le_mn2.
5   by apply: leq_trans le_mn2 le_n21.
6 by case/orP: (leq_total n2 n1) => /th_sym; last rewrite maxnC orbC.
7 Qed.

```

The proof for the `th_sym` sub proof is the text we copy-paste in the previous script, while here it is factored out. Once we have such extra fact in our context we reason by cases on the order relation and we conclude. Remark that the last line instantiates `th_sym` in *each branch* using the corresponding hypothesis on `n1` and `n2` generated by the case analysis. As expected the two instances are symmetric.


```

2 subgoals
m, n1, n2 : nat
th_sym : ∀ x y : nat,
    y <= x -> (m <= maxn x y) = (m <= x) || (m <= y)
=====
(m <= maxn n1 n2) = (m <= n1) || (m <= n2) ->
(m <= maxn n1 n2) = (m <= n1) || (m <= n2)

subgoal 2 is:
(m <= maxn n2 n1) = (m <= n2) || (m <= n1) ->
(m <= maxn n1 n2) = (m <= n1) || (m <= n2)

```

This is exactly what is needed in the first branch of the case analysis. The last subgoal just requires commuting `max` and `||`.

We can further improve the script. For example we could rephrase the proof putting in front the justification of the symmetry, and then prove one case when we pick x to be smaller than y .

```

1 Lemma leq_max m n1 n2 : (m <= maxn n1 n2) = (m <= n1) || (m <= n2).
2 Proof.
3   suff th_sym x y: y <= x -> (m <= maxn x y) = (m <= x) || (m <= y).
4     by case/orP: (leq_total n2 n1) => /th_sym; last rewrite maxnC orbC.
5   move=> le_n21; rewrite (maxn_idP1 le_n21) orb_idr // => le_mn2.
6   by apply: leq_trans le_mn2 le_n21.
7   Qed.

```

The `suff` tactic (or `suffices`) is like `have` but swaps the two goals.

Note that here the sub proof is now the shortest paragraph. This is another recurrent characteristic of the proof script style we adopt in the Mathematical Components library.

There is still a good amount of repetition in the current script. In particular the main conjecture has been almost copy-pasted in order to invoke `have` or `suff`. When this repetition is a severe problem, i.e. the statement to copy is large, one can resort to the `wlog` tactic (or `without loss`).

```

1 Lemma leq_max m n1 n2 : (m <= maxn n1 n2) = (m <= n1) || (m <= n2).
2 Proof.
3   wlog le_n21: n1 n2 / n2 <= n1 => [th_sym|].
4     by case/orP: (leq_total n2 n1) => /th_sym; last rewrite maxnC orbC.
5   rewrite (maxn_idP1 le_n21) orb_idr // => le_mn2.
6   by apply: leq_trans le_mn2 le_n21.
7   Qed.

```

Remark how `wlog` only needs the statement of the extra assumption, and which portion of the context needs to be abstracted, here `n1` and `n2`. The subgoal to be proved is the following one.

```

2 subgoals
m, n1, n2 : nat
th_sym : ∀ n1 n2 : nat, n2 <= n1 ->
      (m <= maxn n1 n2) = (m <= n1) || (m <= n2)
=====
(m <= maxn n1 n2) = (m <= n1) || (m <= n2)

subgoal 2 is:
(m <= maxn n1 n2) = (m <= n1) || (m <= n2)

```

To keep the script similar to the previous one, we named explicitly `th_sym`, to better link the final script to the previous attempts. This is rarely the case in proof scripts of the library, since one typically uses the `/(_ ...)` intro pattern to specialize the top of the stack.

Shrinking proof scripts is a never ending game. The avid reader can see in the next section how intro patterns let one squeeze the last two lines into a single one. Indeed there are three steps in this script: remarking 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 finally proving, by transitivity, what is left when `max` is computed away thanks to the extra $(n2 \leq n1)$ assumption.

Partially applied views

A less important, but very widespread feature, of the `Ssreflect` proof language can be used to shrink even further the proof. Indeed we name the fact `le_mn2` just to apply transitivity to it.

```

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_idP1 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 $(m \leq n2)$. Note that the `leq_trans` expects a second proof argument, and that its type would fix `b`, that is otherwise unspecified. If one puts a full stop just before the terminating `->`, to see the output of the view application, he sees the following stack:

```

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

```

Rewriting the top assumption fixes `b` to `n1`, trivializes the goal $(m \leq n1)$ to `true` and opens a trivial side condition $(n2 \leq n1)$.

The very compact idiom `/leq_trans->` is quite frequent in the Mathematical Components library.

4.3.3 Euclidean division, simple and correct

Euclidean division is defined as one expects: iterating subtraction.

```
1 Definition edivn_rec d :=
2   fix loop m q := if m - d is m'.+1 then loop m' q.+1 else (q, m).
3
4 Definition edivn m d := if d > 0 then edivn_rec d.-1 m 0 else (0, m).
```

The `fix` keyword lets one write a recursive function locally, without providing a global name as `Fixpoint` does. This also means that `d` is a parameter of `edivn_rec` that does not change during recursion. The `edivn` program handles the case of a null divisor, producing the dummy pair $(0, m)$ for the quotient and the 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 elim: m. Qed.
```

It is often useful to state and prove unfolding equations like this one. Indeed the simplification tactic `/=` may unfold 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, being `edivn` a recursive program, its specification needs to be proved by induction. Given that the recursive call is on the subtraction of the input, we need to perform a strong induction, as we did for the postage example in section 3.3.1.

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

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

Line 1 handles the case of d being zero. The “`in E...`” suffix can be appended to each tactic in order to push on the stack the specified hypotheses before running the tactic and pulling them back afterwards (see [10, Section 6.5]). The `*` means that the goal is also affected by the tactic, and not just the hypotheses explicitly selected.

Lines 2 and 3 prepare the induction by unfolding the definition of `edivn` (to

expose the initial value of the accumulators of `edivn_rec`) and makes the invariant of the division loop explicit replacing `m` by `(0 * d.+1 + m)`. Recall the case `d` being 0 has already been handled.

Line 4 performs a strong induction, also generalizing the initial value of the accumulator 0, leading to the following goal:

```
d, n : nat
IHn : ∀ m n0 : nat, m <= n ->
  let ed := edivn_rec d m n0 in
  (ed.2 < d.+1) && (n0 * d.+1 + m == ed.1 * d.+1 + ed.2)
m, q : nat
le_mn : m < n.+1
=====
let ed := edivn_rec d m.+1 q in
(ed.2 < d.+1) && (q * d.+1 + m.+1 == ed.1 * d.+1 + ed.2)
```

Note that the induction touches variables used in `ed` that is hence pushed on the goal stack. The `@` modifier tells `Ssreflect` to keep the body of the `let-in`.

Line 5 unfolds the recursive function and uses the following lemma to push the subtraction into the branches of the if statement. Then it reasons by cases on the guard of the if-then-else statement.

```
1 Lemma subn_if_gt T m n F (E : T) :
2   (if m.+1 - n is m'.+1 then F m' else E) =
3   (if n <= m then F (m - n) else E).
```

The else branch corresponds to the non recursive case of the division algorithm and is trivially solved in line 6. The recursive call is done on `(m-d)`, hence the need for a strong induction. In order to satisfy the premise of the induction hypothesis, line 7 shows that `(m - d <= n)`. Line 8 concludes.

4.4 Notational aspects of specifications

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

A function automatically inserted by COQ to prevent a type error is called *coercion*. The most pervasive coercion in the Mathematical Components library is `is_true` one that lets one write statements using boolean predicates.

```
1 Lemma example : prime 17.
2
3
```

Goal fully printed

```
=====
is_true (prime 17)
```

The statement of the example is processed by COQ it is enforced to be a type, but `(prime 27)` is actually a term of type `bool`. Early in the library the function

`is_true` is declared as a coercion from `bool` to `Prop` and hence is it inserted by COQ automatically.

```
1 Definition is_true b := b = true.
2 Coercion is_true : bool -> Sortclass. (* Prop *)
```

Another coercion that is widely used injects booleans into naturals. Two examples follow.

```
1 Fixpoint count (a : pred nat) (s : seq nat) :=
2   if s is x :: s' then a x + count a s' else 0.
3 Lemma count_uniq_mem (s : seq nat) x :
4   uniq s -> count (pred1 x) s = has (pred1 x) s.
```

In line number 2 the term `(a x)` is a boolean. The `nat_of_bool` function is automatically inserted to turn `true` into 1 and `false` into 0. This notational trick is reminiscent of Kronecker's δ notation. Similarly, in the last line the membership test is turned into a number, that is shown to be equivalent to the count of any element in a list that is duplicate free.

Coercions are composed transitively.

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

For the convenience of the reader we list here the most widely used coercions. there are also a bunch on `Funclass` not listed and `elimT` surely deserves some explanation.

Coercions		
coercion	source	target
<code>Posz</code>	<code>nat</code>	<code>int</code>
<code>nat_of_bool</code>	<code>bool</code>	<code>nat</code>
<code>elimT</code>	<code>reflect</code>	<code>Funclass</code>
<code>isSome</code>	<code>option</code>	<code>bool</code>
<code>is_true</code>	<code>bool</code>	<code>Sortclass</code>

The reader already familiar with the concept of coercion may find the presentation of this chapter nonstandard. Indeed coercions are usually presented as a device to model subtyping in a theory that, like the Calculus of Inductive Constructions, does not feature subtyping. As we will see in chapter 7 the role played by coercions is in the modelling of the hierarchy of algebraic structure is minor. Indeed the job of coercions in that context is to forget some fields of a structure to obtain a simpler one, and that is easy. What is hard is to reconstruct the missing fields of a structure or compare two structures finding the minimum super structure. These tasks are mainly implemented programming type inference.

4.5 Exercises

Exercise 12. *reflect*

Prove the following lemmas. In particular prove the first one with and without using `iffP`.

```
1 Lemma iffP_lr (P : Prop) (b : bool) :
2   (P -> b) -> (b -> P) -> reflect P b.
3 Lemma iffP_rl (P : Prop) (b : bool) :
4   reflect P b -> ((P -> b) /\ (b -> P)).
```

Exercise 13. *eqnP*

Finish this proof.

```
1 Lemma eqnP n m : reflect (n = m) (eqn n m).
2 Proof.
3 apply: (iffP idP).
```

Exercise 14. *Injectivity to nat*

Finish this proof.

```
1 Lemma nat_inj_eq T (f : T -> nat) x y :
2   injective f -> reflect (x = y) (eqn (f x) (f y)).
3 Proof.
4 move=> f_inj; apply: (iffP eqnP).
```

Exercise 15. *Characterization of max*

Prove the following lemma.

```
1 Lemma maxn_idPl m n : reflect (maxn m n = m) (m >= n).
```

4.5.1 Solutions**Answer of Exercise 12**

```
1 Lemma iffP_lr (P : Prop) (b : bool) :
2   (P -> b) -> (b -> P) -> reflect P b.
3 Proof.
4 by case: b => [_ H|H _]; [left; apply H | right=> p; move: (H p)].
5 (* with iffP: by move=> *, apply: (iffP idP). *)
6 Qed.
7
8 Lemma iffP_rl (P : Prop) (b : bool) :
9   reflect P b -> ((P -> b) /\ (b -> P)).
10 Proof. by case: b; case=> p; split. Qed.
```

Answer of Exercise 13

```

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

```

Answer of Exercise 14

```

1 Lemma nat_inj_eq T (f : T -> nat) x y :
2   injective f -> reflect (x = y) (eqn (f x) (f y)).
3 Proof. by move=> f_inj; apply: (iffP eqnP) => [/f_inj|->]. Qed.

```

Answer of Exercise 15

```

1 Lemma maxn_idP1 m n : reflect (maxn m n = m) (m >= n).
2 Proof.
3 by rewrite -subn_eq0 -(eqn_add2l m) addn0 -maxnE; apply: eqP.
4 Qed.

```


Part II

Formalization techniques

Chapter 5

Type Inference

Teaching Coq how to read Math

The rules of the Calculus of Inductive Constructions, as the ones sketched in 3, are expressed on the syntax on 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.[20] 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 programs¹ that have access to the library of already formalized facts. In this way one can make the type inference algorithm aware of the

¹A 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.

contents of the library and make COQ behave as a trained reader that is able to guess the intended meaning of a mathematical expressions from the context thanks to his background knowledge.

This chapter also introduces the key concepts of *interface* and *instance*. An interface is essentially the signature of an algebraic structure: operations, properties and notations letting one reason abstractly about a family of objects sharing the interface. An instance is an example of an algebraic structure, an object that fits an interface. For example `eqType` is the interface of data types that come equipped with a comparison function, and the type `nat` forms, together with the `eqn` function, an example of `eqType`.

The programs we will write to extend type inference play two roles. On one hand they link instances to interfaces, like `nat` to `eqType`. On the other hand they build *derived instances* out of basic ones. For example we teach type inference how to synthesize an instance of `eqType` for a type like `(A * B)` whenever `A` and `B` are instances of `eqType`.

The concepts of interface and example are recurrent in both computer science and modern mathematics, but are not a primitive notion in COQ. Despite that, they can be encoded quite naturally, although not trivially, using inductive types and the dependent function space. This encoding is not completely orthogonal to the actual technology (the type inference and its extension mechanism). For this reason we shall need to dive, from time to time, into technical details, especially in sections labelled with two stars.

5.1 Type inference and Higher Order unification

The type inference algorithm is quite similar to the type checking one: it recursively traverses a term checking that each subterm has a type compatible with the type expected by its context. During type checking types are compared taking computation into account. Terms 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 indeed 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 assigning 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 presence of higher order constructs. For example to unify `(prime _)` with `true` one has to guess a prime number. Remember that `prime` is a boolean function that fed with a natural number returns either `true` or `false`. While assigning 2 to the implicit argument would be a perfectly valid solution, it is clear that it is not the only one. Enumerating all possible values until one finds a valid one is not a good strategy either, since the good value may not exist. Just think at the problem `(prime (4 * _))` versus `true`. An even harder class of problems is the one of synthesizing programs. Take for example the unification problem `(_ 17)` versus `[:: 17]`. Is the function we are looking for the list constructor? Or maybe, is it a factorization algorithm?

Given that there is no silver bullet for higher order unification COQ makes a sensible design choice: provide an (almost) heuristic-free algorithm and let the user extend it via an extension language. We refer to such language as the language of *Canonical Structures*. Despite being a very restrictive language, it is sufficient to program a wide panel of useful functionalities.

The concrete syntax for implicit arguments, an underscore character, does not let one name the missing piece of information. If an expression contains multiple occurrence of the placeholder “_” they are all considered as potentially different by the system, and hence hold (internally) unique names. For the sake of clarity we take the freedom to use the alternative syntax `?x` for implicit arguments (where *x* is a unique name).

5.2 Recap: type inference by examples

Lets start with the simplest example one could imagine: defining the polymorphic identity function and checking its application to 3.

Polymorphic identity	Response
<pre> 1 Definition id (A : Type) (a : A) : A := a. 2 Check (id nat 3). 3 Check (id _ 3).</pre>	<pre> id nat 3 : nat id nat 3 : nat</pre>

In the expression `(id nat 3)` no subterm was omitted, and indeed COQ accepted the term and printed its type. In the third line even if the sub term `nat` was omitted, COQ accepted the term. Type inference found a value for the place holder for us by proceeding in the following way: it traversed the term recursively from left to right, ensuring that the type of each argument of the application had the type expected by the function. In particular `id` takes two arguments. The former argument is expected to have type `Type` and the user left such argument implicit (we name it `?A`). Type inference imposes that `?A` has type `Type`, and this constraint is satisfiable. The algorithm continues checking the remaining argument. According to the definition of `id` the type of the second argument must be the value of the first argument. Hence type inference runs recursively on the argument 3 discovering it has type `nat` and imposes that it unifies with the value of the first argument (that is `?A`). For this to be true `?A`

has to be assigned the value `nat`. As a result the system prints the input term, where the place holder has been replaced by the value type inference assigned to it.

At the light of that we observe that every time we apply the identity function to a term we can omit to specify its first argument, since COQ is able to infer it and complete the input term for us. This phenomenon is so frequent that one can ask the system to insert the right number of `_` for him. For more details refer to [22, section 2.7]. Here we only provide a simple example.

Setting implicit arguments	Response
<pre> 1 Arguments id {A} a. 2 Check (id 3). 3 Check (@id nat 3).</pre>	<pre> id 3 : nat id 3 : nat</pre>

The `Arguments` directive “documents” the constant `id`. In this case it just marks then argument that has to be considered as implicit by surrounding it with curly braces. The declaration of implicit arguments can be locally disabled by prefixing the name of the constant with the `@` symbol.

Another piece of information that is often left implicit is the type of abstracted or quantified variables.

Omitting type annotations	Response
<pre> 1 Check (fun x => @id nat x). 2 3 Lemma prime_gt1 p : prime p -> 1 < p.</pre>	<pre> fun x : nat => id x : nat -> nat</pre>

In the first line the syntax `(fun x => ...)` is sugar for `(fun x : _ => ...)` where we leave the type of `x` open. Type inference fixes it to `nat` when it reaches the last argument of the identity function. It unifies the type of `x` with the value of the first argument given to `id` that in this case is `nat`. This last example is emblematic: most of the times the type of abstracted variables can be inferred by looking at how they are used. This is very common in lemma statements. For example the third line states a theorem on `p` without explicitly giving its type. Since the statement uses `p` as the argument of the `prime` predicate, it is automatically constrained to be of type `nat`.

The kind of information filled in by type inference can also be of another, more interesting, nature. So far all place holders were standing for types, but the user is also allowed to put `_` in place of a term.

Inferring a term	Goal after line 3
<pre> 1 Lemma example q : prime q -> 0 < q. 2 Proof. 3 move=> pr_q.</pre>	<pre> 1 subgoal q : nat pr_q : prime q ===== 0 < q</pre>

The proof begins by giving the name `pr_q` to the assumption `(prime q)`. Then it builds a proof term by hand using the lemma stated in the previous example and names it `q_gt1`. In the expression `(prime_gt1 _ pr_q)` the place holder, that

we name $?_p$, stands for a natural number. When type inference reaches $?_p$ it fixes its type to `nat`. What is more interesting is what happens when type inference reaches the `pr_q` term. Such term has its type fixed by the context: `(prime q)`. The type of the second argument expected by `prime_gt1` is `(prime ?_p)` (i.e. the type of `prime_gt1` were we substitute $?_p$ for p). Unifying `(prime ?_p)` with `(prime q)` is possible by assigning q to $?_p$. Hence the proof term just constructed is well typed, its type is `(1 < q)` and the place holder has been set to be q . As we did for the identity function we can declare the `p` argument of `prime_gt1` as implicit. Choosing a good declaration of implicit arguments for lemmas is tricky and requires one to think ahead how the lemma is used.

So far type inference and in particular unification has been used in its simplest form, and indeed a first order unification algorithm incapable of computing or synthesizing functions would have sufficed. In the next section we introduce the encoding of the relations that is at the base of the declarative programs we write to extend unification in the higher order case. As of today 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 [2]. The reader interested in a technical presentation of a simplified version of the unification algorithm implemented in COQ can read [21, 11].

5.3 Records as relations

In computer science a record is a very common data structure. It is a compound data type, a container with named fields. Records are represented faithfully in the Calculus of Inductive Constructions as inductive data types with just one constructor, recall 1.9. 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 precedes 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.

Computation of projections	Response
<pre> 1 Eval simpl in sort nat_eqType. 2 Eval simpl in @eq_op nat_eqType. </pre>	<pre> = nat : Type = eqn : sort nat_eqType -> sort nat_eqType -> bool </pre>

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 `(sort (Pack nat eqn))` and reduces the projection extracting `nat`. The obtained term literally matches the type of the last two arguments given to `eq_op`.

Now, why this complication? Why should one prefer `(eq_op nat_eqType 3 4)` to `(eqn 3 4)`? The answer is *overloading*. It is recurrent in mathematics and computer science to reuse a symbol, a notation, in two different contexts. A typical example coming from the mathematical practice is to use the same infix symbol `*` to denote any ring multiplication. A typical computer science example is the use of the same infix `==` symbol to denote the comparison over any data type. Of course the underlying operation one intends to use depends on the

values it is applied to, or better their type.² Using records lets us model these practices. Note that, thanks to its higher order nature, the term `eq_op` can always be the head symbol denoting a comparison. This makes it possible to recognize, hence print, comparisons in a uniform way as well as to input them. On the contrary, in the simpler expression `(eqn 3 4)` the name of the head symbol is very specific to the type of the objects we are comparing.

In the rest of this chapter we focus on the overloading of the `==` symbol and we start by defining another comparison function, this time for the `bool` data type.

```
1 Definition eqb (a b : bool) := if a then b else ~~ b.
2 Definition bool_eqType : eqType := @Pack bool eqb.
```

Now the idea is to define a notation that applies to any occurrence of the `eq_op` head constant and use such notation for both printing and parsing.

Overloaded notation	Response
<pre>1 Notation "x == y" := (@eq_op _ x y). 2 Check (@eq_op bool_eqType true false). 3 Check (@eq_op nat_eqType 3 4).</pre>	<pre>true == false : bool 3 == 4 : bool</pre>

As a printing rule, the place holder stands for a wild card: the notation is used no matter the value of the first argument of `eq_op`. As a result both occurrences of `eq_op`, line 2 and 3, are printed using the infix `==` syntax. Of course the two operations are different, they are specific to the type of the arguments and the typing discipline ensures the arguments match the type of the comparison function packaged in the record.

When the notation is used as a parsing rule, the place holder is interpreted as an implicit argument: type inference is expected to find a value for it. Unfortunately such notation does not work as a parsing rule yet.

```
1 Check (3 == 4).
2
```

Error: The term "3" has type "nat" while it is expected to have type "sort ?e".

If we unravel the notation the input term is really `(eq_op _ 3 4)`. We name the place holder $?_e$. If we replay the type inference steps seen before, the unification step is now failing. Instead of `(sort nat_eqType)` versus `nat`, now unification has to solve the problem `(sort $?_e$)` versus `nat`. This problem falls in one of the problematic classes we presented in section 5.1: the system has to synthesize a comparison function (or better a record instance containing a comparison function).

COQ gives up, leaving to the user the task of extending the unification algorithm with a declarative program that is able to solve unification problems of the form `(sort $?_e$)` versus τ for any τ . Given the current context it seems reasonable to write an extension that picks `nat_eqType` when τ is `nat` and `bool_eqType` when τ is `bool`. In the language of Canonical Structures such program is expressed as

²The meaning of a symbol in math 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 a proof. By programming type inference we model this practice in 5.5.

follows.

```

Declaring Canonical Structures
1 Canonical nat_eqType.
2 Canonical bool_eqType.

```

The keyword `Canonical` was chosen to stress that the program is deterministic: each type τ is related to (at most) one *canonical* comparison function.

```

1 Check (3 == 4).
2 Check (true == false).
3 Eval compute in (3 == 4).

```

```

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

```

The mechanics of the small program we wrote using the `Canonical` keyword can be explained using the global table of canonical solutions. Whenever a record instance is declared as canonical COQ adds to such table an entry for each field of the record type.

Canonical Structures Index		
projection	value	solution
sort	nat	nat_eqType
sort	bool	bool_eqType

Whenever a unification problem with the following shape is encountered, the table of canonical solution is consulted.

(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 and in the Mathematical Components library. The name of such projections is usually omitted to signal that fact.

What makes this 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.

Records are first class values in the Calculus of Inductive Constructions. As we have seen projections are no special, they are simple functions that pattern match on an inductive data type to access the record fields. Being first class citizens means that one can write a term that combines the fields of two records and builds a new record. Thanks to this fact the language of Canonical Struc-

tures is able to forge new record instances by combining the existing ones via a set of user definable combinators. This is the subject of the next section.

5.4 Records as first class relations

So far we have used the `==` symbol terms whose type is atomic, like `nat` or `bool`. If we try for example to use it on terms whose type was built using a type constructor like the one of pairs we encounter an error.

Error	Response
<pre>1 Check (3, true) == (4, false). 2</pre>	<pre>Error: The term "(3, true)" has type "(nat * bool)%type" while it is expected to have type "sort ?e".</pre>

The term `(3,true)` has type `(nat * bool)` and, so far, we only taught COQ how to compare booleans and natural numbers, not how to compare pairs. Intuitively the way to compare pairs is to compare their components *using the appropriate comparison function*. Let's write a comparison function for pairs.

Comparing pairs
<pre>1 Definition prod_cmp eqA eqB x y := 2 @eq_op eqA x.1 y.1 && @eq_op eqB x.2 y.2.</pre>

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 forces the type of `x` and `y` to be `(sort eqA * sort eqB)`. This means that the records `eqA` and `eqB` hold a sensible comparison function for, respectively, terms of type `(sort eqA)` and `(sort eqB)`.

It is now sufficient to pack together the pair data type constructor and this comparison function in an `eqType` instance to extend the canonical structures inference machinery with a new combinator.

Recursive canonical structure
<pre>1 Definition prod_eqType (eqA eqB : eqType) : eqType := 2 @Pack (sort eqA * sort eqB) (@prod_cmp eqA eqB). 3 Canonical prod_eqType.</pre>

The global table of canonical solutions is extended as follows.

Canonical Structures Index			
projection	value	solution	combines solutions for
<code>sort</code>	<code>nat</code>	<code>nat_eqType</code>	
<code>sort</code>	<code>bool</code>	<code>bool_eqType</code>	
<code>sort</code>	<code>T1 * T2</code>	<code>prod_eqType pA pB</code>	<code>pA ← (sort,T1), pB ← (sort,T2)</code>

The third column is empty for base instances while it contains the recursive calls for instance combinators. With the updated table when the unification problem

(sort ?_e) versus (T1 * T2)

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

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

```
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).
```

5.5 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. Indeed there is little one can do with a comparison function if that function is not “correct”.

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

The extra property turns the eqType relation into a proper *interface*, that fully specifies what op is.

The axiom says that the boolean comparison function is compatible with equality: two ground terms compare as equal if and only if they are syntactically equal. Note that this means that the comparison function is not allowed to quotient the type by identifying two syntactically different terms.

Advice

The infix notation == stands for a comparison function compatible with Leibniz equality (substitution in any context).



The `Equality` module enclosing the record acts a name space: `type`, `sort`, `eq` and `axiom`, three very generic words, are here made local to the `Equality` name space becoming, respectively, `Equality.type`, `Equality.sort`, `Equality.op` and `Equality.axiom`.

As in section 5.3, the record plays the role of a relation and its `sort` component is again the only field that drives canonical structure inference. Following a terminology typical of object oriented programming languages, the set of operations (and properties) that define an interface is called a *class*. In the next chapter we are going to re-use already defined classes in order to build new ones by mixing-in additional properties (typically called axioms). Hence the definition of `eqType` in the Mathematical Components library is closer to the following one:

The real definition of `eqType`

```

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`.

Said that, nothing really changes: the `eqType` structure relates a type with a signature.

Remark the use of `>` instead of `:` to type the field called `sort`. This tells COQ to declare the `Equality.sort` projection as a coercion, enabling one to write $(\forall T : \text{eqType}, \forall x y : T, P)$. Indeed `T` is not a type, only its `sort` projection is.

Warning

Being `Equality.sort` a coercion, it is not displayed by COQ, hence error messages about a missing canonical structure declaration typically look very confusing: “...has type `nat` but should have type `?e`”, instead of “...but should have type `(sort ?e)`”.



Given the new definition of `eqType`, when we write `(a == b)` type inference does not only infer a function to compare `a` with `b` but also a proof that such

function is correct. Indeed declaring the `eqType` instance for `nat` 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.

Making eqn 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.

5.6 Using a generic theory

The whole point of defining interfaces is to share a theory among all examples of each interface. In other words a theory proved starting from the properties (axioms) of an interface applies to all its instances, transparently. Every lemma part of an abstract theory is *generic*: the very same name can be used for each and every instance of the interface, exactly as the `==` notation.

The simplest lemma part of the theory of `eqType` is the `eqP` generic lemma that can be used in conjunction with any occurrence of the `==` notation.

The eqP lemma

```
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 `τ`. We can use it on an 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`.

The complete definition of `prod_eqType`

```

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.

```

Like for `nat` 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 is true (or better, the hypothesis is false) by computation.

Why one should always use `==` (EM)

```

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

```

Advice

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



5.7 The generic theory of sequences

Now that the `eqType` interface equips a type with a well specified comparison function we can use it build abstract theories, for example the one of sequences.

It is worth to remark that the concept of interface is crucial to the development of such theory. If we try to develop the theory of the type `(seq T)` for an arbitrary `T`, we can't go much far. For example we can express what belonging to a sequence means, but not write a program that tests if a value is indeed in the list. As a consequence we lose the form 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 type that parameterizes a generic container we define which operations are licit and which properties hold. So far the only interface we know is `eqType`, that is primordial to boolean reflection. In the next chapters more elaborate interfaces will enable us to organize knowledge in articulated ways.

Going back to the abstract theory of sequences over an `eqType`, we start by defining the membership operation.

```

Membership
1 Section SeqTheory.
2 Variable T : eqType.
3 Implicit Type s : seq T.
4
5 Fixpoint mem_seq s x :=
6   if s is y :: s' then (y == x) || mem_seq s' x else false.

```

Like we did for the overloaded `==` notation, we can define the `\in` (and `\notin`) infix notation. We can then easily define what a duplicate-free sequence is, and how to enforce such property.

```

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.

```

undup is correct (step 1)
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 convenience rewrite rule, while `mem_undup` says that the `undup` function does not drop any non-duplicate element. Note that in the proof we use both the decidability of membership (`Hy`) and the decidability

of equality (via `eqP`).

```
undup is correct (setp 2)

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`.

A last, very important step in the theory of sequences is to show that the container preserves the `eqType` interface: whenever we can compare the elements of a sequence, we can also compare sequences.

```
eqType for sequences

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.
```

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 && undup_uniq ss.
```

As we have anticipated in Chapter 1, functional programming and lists can model, definite, iterated operations like the “big” sum Σ . Next Section describes how the generic theory of iterated operations can be built and made practical thanks again to the programming of type inference via the canonical structures language.

5.8 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.

Lets 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 begin iterated and the neutral element for such operator. For example if A is empty, then the union of all $g(a)$ is \emptyset while if $n = 0$ then the sum of all $f(i)$ is 0. Then the range is an expression identifying a finite set of elements, some times expressing an order (relevant when the iterated operation is not commutative). Finally a general term describing the items being combined by the iterated operation.

As already mentioned in 1.7, 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 list and combining the results by the `fold` iterator.

Functional programming can also be used to describe the finite domain. For example the list of natural numbers $m, m+1, \dots, m+(n-m)$ corresponding to the range $m \leq i < n$ can be built using the `iota` function as follows:

Definition `index_iota m n := iota m (n - m).`

The only component of typical notations for iterated operations we have not discussed yet is the filter, used to iterate the operation only on a subset of the domain. For example to state that the sum of the first n odd numbers is n^2 one could write:

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

An alternative writing for the same summation exploits the general terms to rule out even numbers:

$$\sum_{i < n} (i * 2 - 1) = n^2$$

While this latter writing is elegant, it is harder to support generically, since the filtering condition is not explicit. For example the following equation clearly holds for any filter, range and general term. It would be hard to express such statement if the filter was mixed with the general term, and hence its negation was 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. En example is not being a prime number.

At 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 Σ . section 5.8.2 focuses on the generic theory for iterated operations.

5.8.1 The generic notation for `foldr`

The generic notation for iterated operations has to be attached to something more specific than `foldr` in order to clearly identify all components

```
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 states 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 notation is placed in `nat_scope` and is indeed specialized on `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. On the contrary 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_(1 <= i < 3.*2) i.*2 = 5 * 4
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 //.

```

Pulling out the last product

```

=====
\prod_(1 <= i < 4) i * 4 = 4'!

```

This is the reason why we can say that the bigop library is generic: it works uniformly on any iterated operator, and provided the operator has certain properties it give uniform access to a palette of lemmas.

5.8.2 Assumptions of a bigop lemma

As we anticipated, canonical structures can be indexed not only on types, but on any term. In particular we can index them on function symbols to relate, for example, `addn` and its monoid structure.

Here we only present the `Monoid` interface an operation has to satisfy in order to access a class of generic lemmas. Chapter 7 adds other interfaces to the picture and organizes the bigop library around them.

```

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 parameterize an entire theory on this interface.

```

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[operation 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 addn0n addn0.

```

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

Canonical Structures Index		
projection	value	solution
Monoid.operator	addn	addn_monoid

Whenever the lemma is applied to an expression about natural numbers as

```
1 Lemma test : \sum_(0 <= i < 6) i = \sum_(0 <= i < 5) i + 5.
2 Proof. by apply: big_nat_recr. Qed.
```

the following unification problem has to be solved: `addn` versus `(operator ?m)`. Inferring a value for `?m` mean inferring a proof that `addn` forms a monoid with 0: a prerequisite for the `big_nat_recr` lemma we don't have to provide by hand.

5.8.3 Searching the bigop library

Searching the bigop library for a lemma is slightly harder than searching the other libraries as explained in section 2.4. 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 begin folded is `nil` then the result is the initial value. Such lemma, called `big_nil`, indeed 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 get acquainted with the naming policy used in that library.

5.9 Stable notations for big operators (★★)

The definition of `bigop` 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.

```

5.10 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. Indeed 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 simplifies away, unless the 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 rewrite /=.
8 by rewrite addnC -big_cat.
9 Qed.
```

Response after line 6

```
F : nat -> nat
l1, l2 : seq nat
=====
addn_monoid
(\big[addn_monoid/0]_(i <- l1) F i)
(\big[addn_monoid/0]_(i <- l2) F i) =
\sum_(i <- (l2 ++ l1)) F i
```

It is not uncommon to see `/=` switch in purely algebraic proofs (where no computation is really involved) just to clean up the display of the current conjecture.

5.11 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.

5.11.1 Phantom types (★)

First of all, canonical structure resolution kicks in during unification that in turn is used to compare types. Types are compared whenever a function is applied to

an argument, and in particular the type expected by the function and the one of the argument are unified. What we need to craft is a mechanism that takes any input term (proper terms like `addn` but also types as `nat`) and puts it into a type. We will then wire things up so that such type is unified with another one containing the application of a projection to an unknown canonical structure instance.

```
1 Inductive phantom (T : Type) (p : Type) := Phantom.
```

The `Phantom` constructor expects two arguments. If we apply it to `nat` as in `(Phantom Type nat)` we obtain a term of type `(phantom Type nat)`. If we apply it to `addn` as in `(Phantom (nat -> nat -> nat) addn)` we obtain a term of type `(phantom (nat -> nat -> nat) addn)`. In both cases the input term (`nat` and `addn` respectively) is now part of a type.

The following example defined a notation `{set T}` that fails if `T` is not an `eqType` but is an alias of the type `(seq T)` that imposes no requirement on the type argument.

```
1 Definition set_of : (T : eqType) (_ : phantom Type (Equality.sort T)) :=
  seq T.
2 Notation "{ 'set' T }" := (set_of _ (Phantom _ T))
3 (at level 0, format "{ 'set' T }" : type_scope.
```

When type inference runs on `{set nat}` the underlying term being typed is `(set_of ?T (Phantom ?N nat))`. The unification problem arising for the last argument of `set_of` is `(phantom Type (Equality.sort ?T))` versus `(phantom Type nat)`, that in turn contains the sub problem we are interested in: `(Equality.sort ?T)` versus `nat`.

5.12 Exercises

Exercise 16. *Sum of $2n$ odd numbers*

Show the following lemma using the theory of big operators

```
1 Lemma sum_odd n : \sum_(0 <= i < n.*2 | odd i) i = n^2.
```

5.12.1 Solutions

Answer of Exercise ??

```
1 Lemma sum_odd n : \sum_(0 <= i < n.*2 | odd i) i = n^2.
2 Proof.
3 elim: n => [|n IHn]; first by rewrite unlock.
4 rewrite doubleS big_mkcond 2?big_nat_recr // -big_mkcond /.
5 by rewrite {}IHn odd_double /= addn0 -addnn -!mulnn; ring.
6 Qed.
```

Chapter 6

Sub-Types

Terms with properties

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 eventually show that the sum is still smaller than 7. Also, an informed reader knows which operations are compatible with a subset. E.g. $(i-1)$ stays in `'I_7`, as well as $(i+n \% 7)$. So in a sense, subsets also provide a linguistic construct to ask the reader to track an easy invariant and relieving the proof text from boring details.

The closest notion provided by the Calculus of Inductive Constructions is the one of Σ -types, that we have already seen in the previous chapter in their general form of records. For example, one can define the type `'I_7` as $\Sigma_{(n:\text{nat})} n \leq 7$. 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 $(p : 3 \leq 7)$. 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 to access a whole theory of results through type inference. Many such interfaces can be extended to sub-types, and we shall see that the Mathematical Components library provides facilities to automate this.

6.1 n -tuples, lists with an invariant on the length

We begin by defining the type of n -tuples: sequences of length n . In this section we focus on how tuples are defined, used as regular sequences and how to program type inference to track for us the invariant on tuples' length. Next section will complete the definition of the tuple sub-type by making the abstract theory attached to the `eqType` interface available on tuples whenever it is available on sequences.

A tuple is a sequence of values (of the same type) whose length is made explicit in the type.

Tuple sub-type of seq

```
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 map the function over the list and then reverse 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 make 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 of 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) 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` make 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 try to unify the left hand side of the `size_tuple` equation with the redex `(rev t)`, using the following toolkit:

Unification debugging 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:

Response

```

Error:
In environment
T : Type
n : nat
t : n.-tuple T
LHS := size (tval ?94 ?92 ?96) (*...*) : nat
RDX := size (rev (tval n T t))      : nat
The term "erefl ?95" has type "?95 = ?95" while
it is expected to have type "LHS = RDX".

```

Unifying $(\text{size } (\text{tval } ?_n ?_T ?_t))$ with $(\text{size } (\text{rev } (\text{tval } n \ T \ t)))$ is hard. Both term's head symbol is `size`, but then the projection `tval` applied to unification variables has to be unified with $(\text{rev } \dots)$, 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 $\text{rev } t$ (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.

```

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

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 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, unlike in the first, one is not required to know the names of the lemmas: it is the new concept of tuple that takes care of the size related reasoning steps.

6.2 *n*-tuples, a sub-type of sequences

We have seen that `(seq T)` is an `eqType` whenever `T` is. We now want to transport such `eqType` structure on tuples. We first do it manually, then we provide a toolkit to ease the declaration of sub-types.

The first step is to define a comparison function for tuples.

```

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, that is briefly discussed in 6.2.2.

Response after line 3	Response after line 5
<pre> 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 </pre>	<pre> 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 </pre>

We can then declare the canonical `eqType` instance for tuples.

```

1 Canonical tuple_eqType n T : eqType :=
2   Equality.Pack (Equality.Mixin (@eqtupleP n T)).

```

As a simple test we check that the notations and the theory that equips `eqType` is available on tuples.

```

1 Check ∀ t : 3.-tuple nat, [:: t] == [:::].
2 Check ∀ t : 3.-tuple bool, uniq [:: t; t].
3 Check ∀ t : 3.-tuple (7.-tuple nat), undup_uniq [:: t; t].

```

Although all these proofs and definitions are specific to `tuple`, it seems obvious that we are following a schema here, and that the parameters are three: the original type (`seq T`), the sub type (`n.-tuple T`) and the projection `tval`. The sub-type kit lets one write the following:

```

1 Canonical tuple_subType := Eval hnf in [subType for tval].
2 Definition tuple_eqMixin := Eval hnf in [eqMixin of n.-tuple T by <:].
3 Canonical tuple_eqType := Eval hnf in EqType (n.-tuple T) tuple_eqMixin.

```

Line 1 registers `tval` as a canonical projection to obtain a known type out of the newly defined type of tuple. 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.

6.2.1 The sub-type kit

(★)

When one has a base type `T` and a sub-type `ST` defined as a boolean sigma type, the Mathematical Components library provides facilities to build all the applicable canonical instances from just the name of the projection going from `ST` to `T`.

To register `tval` as the projection from tuples to sequences one writes:

```

1 Canonical tuple_subType := Eval hnf in [subType for tval].

```

As we will see in the next section, the `subType` structure provides a generic notation `val` for the projector of a sub-type (i.e., `tval` for `tuple`), with an over-

loaded injectivity lemma `val_inj` saying that two objects equal in ST are also equal in T.

In addition to the generic projection we get a generic static constructor `Sub`, which takes a value in 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* [4, Section 8.4]. The `insubd` constructor takes a default sub-type value which it returns if the tests fails, while `insub` takes only a base type value and returns an `option`; both are *locked* and will not evaluate the test, even for a ground base type value.¹

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

Here are 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.
4 Let s2 := 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 6.4 when an enumeration for sub-types is to be defined.

The `subType` structure describes a *boolean sigma-type* (a dependent pair whose second component is the proof of a boolean formula) in terms of its projector, constructor, and elimination rule:

```
1 Definition pred T := T -> bool.
2 Section SubTypeKit.
3 Variables (T : Type) (P : pred T).
4
5 Structure subType : Type := SubType {
6   sub_sort :> Type;
7   val : sub_sort -> T;
8   Sub : ∀ x, P x -> sub_sort;
9   (* elimination rule for sub_sort *)
10  _ : ∀ K ( _ : ∀ x Px, K (@Sub x Px)) u, K u;
11  _ : ∀ x Px, val (@Sub x Px) = x
12 }.
```

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)).
```

¹The equations describing the computation of `insub` are called `insubT` and `insubF`.

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 [22, section 1.2.13].

A useful variant of `[subType for tval]` is `[newType for Sval]`. Such specialized constructors forces 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 by inhabitants of `T`. Of course all the theory that equips `T` is also available on `ST`. Aliasing a type is useful to attach to it different notations or coercions.

6.2.2 A note on boolean Σ -types

The `eq_irrelevace` 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 [13] that proves such property for a wide class of predicates. In particular it show that any type with decidable identity has unique identity proofs. This result can be proved in its full generality in the Mathematical Components library, using to the `eqType` interface.

Hedberg

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

If we pick the concrete example of `bool`, then all proofs that `(b = true)` for a fixed `b` are identical.

Here we can see another crucial advantage of boolean reflection. Forming sub-types poses no complication from a logic perspective since proofs of boolean identities are very simple, canonical, objects.

In the Mathematical Components library, where *all predicates that can be expressed as a boolean function are expressed as a boolean function*, forming sub-types is extremely easy.

Advice

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



6.3 Finite types and their theory

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

Interface for finite types

```

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:

Declare a finType

```

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.

Declare a finType

```

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]`.

Some theory for finType

```

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 `[∀ x, P x]` is a boolean expression it enables reasoning by excluded middle and also combines well with other boolean connectives.

What makes this formulation of finite types handy is the explicit enumeration. It is hence trivial to iterate over the inhabitants of the finite type. This makes finite type easy to integrate in the library of iterated operations. In particular notations like `(\sum (i : T) F)` are used to express the iteration over the inhabitants of the finite type T .

6.4 The ordinal subtype

Apart from the aforementioned theory, finite types can serve as a powerful notational device for ranges. For example one may want to state that a matrix of size $m \times n$ is only accessed inside its bounds, i.e. that one cannot get the $m + 1$ row. The way this will be formulated in the Mathematical Components library is by saying that its row accessors accept only inhabitants of a finite type of size m . Accessing a matrix out of its bounds becomes a type error. Of course one wants to access a matrix using integer coordinates, but integers are infinite. Hence the first step is to define the sub-type of bounded integers.

```

Ordinals
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.

```

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, ... n.-1]`. Such 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 produces by `iota`) and `p` is a proof that `(m <= n)`. What we are left to show is that such 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. Indeed one can use ordinals to type the index making COQ statically check 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|.

```

6.5 Finite functions

In standard mathematics functions that are point wise equal are considered as equal. This principle, that we call *functional extensionality*, is compatible with the Calculus of Inductive Constructions but is not built-in. Indeed, at the time of writing, only very recent variations of CIC, as Cubical Type Theory [6], include such principle.

Still there is a class of functions for which such principle is provable in COQ. These are the functions with a finite domain: given their graph is a finite set of points we can represent them as regular lists, and lists with the same values are indeed equal.

```

1 Section FinFunDef.
2 Variables (aT : finType) (rT : Type).
3
4 Inductive finfun : Type := Finfun of #|aT|. -tuple rT.
5 Definition fgraph f := let: Finfun t := f in t.
6 Canonical finfun_subType := Eval hnf in [newType for fgraph].
7
8 End FinFunDef.
9 Notation "{ 'ffun' fT }" := (finfun fT).

```

The actual definition in the Mathematical Components library is slightly more complex to statically check that the domain is finite using the tricks explained in 5.11. I.e. COQ rejects `{ffun nat -> nat}` but accepts `{ffun 'I_7 -> 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. Recall the `all_words` function, solution of Exercise 7.

```

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, begin equal to $\#|rT| \wedge \#|aT|$.

```

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

```

Also, as expected, finite function 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 indexes i and j are independently chosen.

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

6.6 Finite sets

We have seen how sub-types let one easily define a new type by, typically, enriches 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.

```

Using the sub-type kit we can easily transport the `eqType` and `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 τ is a finite type whenever we write `{set T}`, exactly as we did for finite functions. Finite sets do validate extensionality and are equipped with subset-wise and point-wise operations:

```

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 `bool`.

Since `T` is finite, values of type `{set T}` admit a complement and `{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.

6.7 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).

```

Similarly to finite functions we can declare a coercion to let one write `(s x)` for `(s : {perm T})` to denote the result of applying the permutation `s` to `x`.

Thanks to the sub-type kit it is easy to transport to the type `{perm T}` the `eqType` and `finType` structures of `ffun T -> T`.

```

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 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| '!.

```

6.8 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 function 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 a known type $'M[R]_{(m,n)}$, then type inference 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 $\backslash\text{sum}$ 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 an overloaded `eqType` operation.

Via the sub-type kit we can transport `eqType` and `finType` from `{ffun 'I_m * 'I_n -> R}` to $'M[R]_{(m,n)}$. We omit the COQ code for brevity.

A useful accessory is the notation to define matrices in their extension. We provide a variant in which the matrix size is given and one in which it has to be inferred from the context.

```

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 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 $(-1) \wedge^+ s$ denotes signature of the permutation s : s can be used, thanks

to a coercion, as natural number that is 0 if s is an even permutation, 1 otherwise, and $\sim +$ 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^AT_EX:

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

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 statically rejects multiplication of matrices which sizes don't match.

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 matrix 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 will 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 6.8.3. At the same time type inference lets one omit size information most of the times, playing once again the role of a trained reader.

6.8.1 Example: matrix product commutes under trace

As an example let's take a 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_j \sum_i 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 `ssreflect` proof language provides the `gen` modifier tell `have` to abstract the given formula over 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)
trAB : \tr (A *m B) = \sum_i \sum_j A i j * B j i
trE : \forall m n (A : 'M_(m, n)) B, \tr (A *m B) = \sum_i \sum_j A i j * B j i
=====
\sum_j \sum_i 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.

6.8.2 Block operations

The size information stocked in the type of a matrix is also used to drive the decomposition of a matrix into sub-matrices, called blocks. For example when the size expression of a square matrix is like `(n1 + n2)`, then the upper left block is a square matrix of size `n1`.

Block destructors

```
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 blocs can be glued together. This time it is the size of the resulting matrix that shows a trace of the way it was built.

Block constructors

```

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 Atr : 'M_(m1 + m2, n1 + n2)

```

The interested reader can find in [7] a description of Cormen's LUP decomposition, an algorithm making use of these constructions. In particular, recursion on the size of a square matrix of size n naturally identifies an upper left square block of size 1, a row and a column of length $n - 1$, and a square block of size $n - 1$.

6.8.3 Size casts

(★)

Types containing values are a doubly sharp weapon. While we have seen that they make the writing of matrix expressions extremely succinct, in some case they need 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.

```

Indeed the left hand size has type $'M_(m, n1 + (n2 + n3))$ while the right hand size has type $'M_(m, (n1 + n2) + n3)$. The `castmx` operator, and all its companion lemmas, lets 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 $(n1 + (n2 + n3) = (n1 + n2) + n3)$.

Lemmas like the following two lets 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 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.

```

The $(m =P m1)$ notation simply hides $(@eqP \text{ nat_eqType } m \ m1)$, a proof of the `reflect`

inductive spec. Recall 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 4.2.1 differs from the one part of the Mathematical Components library in a tiny detail. Indeed the real declaration 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 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 proof accessible within programs.

However the precise difference between `Prop` and `Type` in the Calculus of Inductive Constructions is off topic for this text, so we will not detail further.

Chapter 7

Hierarchies

Organizing knowledge

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 organising the corresponding Calculus of Inductive Constructions structures in a hierarchy we can materialise 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.

Organising structures in a hierarchy does not require any new logical feature beyond those we have already seen: type inference with dependent types, coercions and canonical instances of structures. It is only a “simple matter of programming”, albeit one that involves some new formalisation idioms. This chapter describes the most important: telescopes, packed classes, and phantom parameters.

While some of these formalisation patterns are quite technical, casual users do not need to master them all: indeed the documented interface of structures suffices to use and declare instances of structures. We describe these interfaces first, so only those who wish to extend old or create new hierarchies need to read on.

7.1 Structure interface

Most of the documented interface to a structure concerns the operations and properties it provides. This will be obvious from the embedded documentation of the `ssralg` library, which provides structures for most of basic algebra (including rings, modules, fields). While these are of course important, they pertain to elements of the structure rather than the structure itself, and indeed are usually defined outside of the module introducing the structure.

The intrinsic interface of a structure is much smaller, and consists mostly of functions for creating instances to be typically declared *Canonical*. For structures like `eqType` that are packaged in a submodule (`Equality` for `eqType`), the interface coincides with the contents of the `Exports` submodule. For `eqType` the interface comprises:

- `eqType`: a short name for the structure type (here, `Equality.type`)
- `EqMixin`, `PcanEqMixin`, `[eqMixin of T by <:]`: mixin constructors that bundle the *new* operations and properties the structure provides.
- `EqType`: an instance constructor that creates an instance of the structure from a mixin.
- `[eqType of T]`, `[eqType of T for S]`: cloning constructors that specialize a canonical or given instance of the structure (to `T` here).
- canonical instances and coercions that link the structures to lower ones in the hierarchy or to its elements, e.g., `Equality.sort`.

Canonical instances and coercions are not mentioned directly in the documentation because they are only used indirectly, through type inference; a casual user of a structure only needs to be aware of which other structure it extends, in the hierarchy.

Let us see how the creation operations are used in practice, drawing examples from the `zmodp` library that puts a “mod p ” algebraic structure on the type `ordinal p` of integers less than p . The `ordinal` type is defined in library `fintype` as follows:

```
1 Inductive ordinal n := Ordinal m of m < n.
2 Notation "'I_n'" := (Ordinal n).
3 Coercion nat_of_ord n (i : 'I_n) := let: @Ordinal _ m _ := i in m.
```

Algebra only makes sense on non-empty types, so `zmodp` only defines arithmetic on `'I_p` when p is an explicit successor. This makes it easy to define `inZp`, a “mod p ” right inverse to the `ordinal` \rightarrow `nat` coercion, and a 0 value. With these the definition of arithmetic operations and the proof of the basic algebraic identities is straightforward.

```
1 Variable p' : nat.
2 Local Notation p := p'.+1.
3 Implicit Types x y : 'I_p.
4 Definition inZp i := Ordinal (ltn_pmod i (ltn0Sn p')).
```


The `inZp` construction injects any natural number `i` into `'I_p` by applying the modulus. Indeed the type of `ltm_pmod` is $(\forall m\ d : \text{nat}, 0 < d \rightarrow m \% d < d)$, and `(ltm0Sn p')` is a proof that $(0 < p)$.

We can now build the \mathbb{Z} -module operations and properties:

```

1 Definition Zp0 : 'I_p := ord0.
2 Definition Zp1 := inZp 1.
3 Definition Zp_opp x := inZp (p - x).
4 Definition Zp_add x y := inZp (x + y).
5 Definition Zp_mul x y := inZp (x * y).
6
7 Lemma Zp_add0z : left_id Zp0 Zp_add.
8 Lemma Zp_mulC : commutative Zp_mul.
9 ...

```

Creating an instance of the lowest `ssralg` structure, the \mathbb{Z} -module (i.e., additive group), requires two lines:

```

1 Definition Zp_zmodMixin := ZmodMixin Zp_addA Zp_addC Zp_add0z Zp_addNz.
2 Canonical Zp_zmodType := ZmodType 'I_p Zp_zmodMixin.

```

Line 1 bundles the additive operations $(0, +, -)$ and their properties in a *mixin*, which is then used in line 2 to create a canonical instance. After line 2 all the additive algebra provided in `ssralg` becomes applicable to `'I_p`; for example `0` denotes the zero element, and `i + 1` denotes the successor of `i` mod `p`.

The `ZmodMixin` constructor infers the operations `Zp_add`, ..., from the identities `Zp_add0z`, ... Providing an explicit definition for the mixin, rather than inlining it in line 2, is important as it speeds up type checking, which never need to open the mixin bundle.

The first argument to the instance constructor `ZmodType` is somewhat redundant, but documents precisely the type for which this instance will be used. This can be important as the value inferred by COQ could be “different”. Indeed line 2 does perform some nontrivial inference, because although `zmodType` is the first `ssralg` structure, it is not at the bottom of the hierarchy: in particular `zmodType` derives from `eqType`, so the `==` test can be used on all `zmodType` elements. The `ZmodType` constructor infers a parent structure instance from its first argument, then combines it with the mixin to create a full `zmodType` instance. This inference can have the side effect of unfolding constants occurring in the description of the type (though not in this case), that is the value on which the canonical solution is indexed. For this reason the type description, `'I_p` here, has to be provided explicitly. Section 7.4 gives the technical details of instance constructors.

Rings are the next step in the algebraic hierarchy. In order to simplify the theory of polynomials, see 1.4, `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 7 endows `'I_p` with a `ringType` structure, making it possible to multiply in `'I_p` or have polynomials over `'I_p`. Line 8 adds a `comRingType` commutative ring structure, which makes it possible to reorder products, or distribute evaluation over products of polynomials. Note that no mixin definition is needed for line 8 as only a single property is added.

Constraining the shape of the modulus p is a simple and robust way to enforce $p > 1$: it standardizes the proofs of $p > 0$ and $p > 1$, which avoid the unpleasantness of multiple interpretations of 0 stemming from different proofs of $p > 0$ — the latter tends to happen with ad hoc inference of such proofs using canonical structures or tactics. The shape constraint can however be inconvenient when the modulus is an abstract constant (say, `Variable p`), and `zmodp` provides some syntax to handle that case:

```

1 Definition Zp_trunc p := p.-2.
2 Notation "'Z_' p" := 'I_(Zp_trunc p).+2.

```

Although it is provably equal to `'I_p` when $p > 1$, `'Z_p` is the preferred way of referring to that type when using its ring structure. Note that the two types are identical when p is a `nat` literal such as 3 or 5.

Cloning constructors are mainly used to quickly create instances for defined types, such as

```

1 Definition Zmn := ('Z_m * 'Z_m)%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].
```

7.2 Telescopes

While using and populating a structure hierarchy is fairly straightforward, creating a robust and efficient hierarchy can be more difficult. In this section we explain *telescopes*, one of the simpler ways of implementing a structure hierarchy. Telescopes suffice for most simple — tree-like and shallow — hierarchies, so new users do not necessarily need expertise with the more sophisticated *packed class* organisation covered in the next section.

Because of their limitations (covered at the end of this section), telescopes were not suitable for the main type structure hierarchy of the Mathematical Components library, including `eqtype`, `choice`, `fintype` and `ssralg`. However, as we have seen in section 5.8.2, structures can be used to associate properties to any logical object, not just types, and the `Monoid.law` structure introduced in 5.8.2 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 facilitates 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` hierarchy is small there is no need to bundle the `Monoid.law` properties in a mixin. It thus takes only one line to declare an instance for the boolean “and” operator `andb`

```
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`,¹ as it informs type inference how to solve the unification problem (`operator ?L`) versus `andb`, by setting `?L` to `andb_monoid`.

Now many of the more interesting `bigop` properties permute the operands of the iterated operator; for example

```
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. After

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

one can state `big_pairA`, declare

```
1 Canonical andb_comoid := ComLaw andbC.
```

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

```
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 `big_pairA` really stands for `operator (com_operator op)`. Thus applying `big_pairA` to the term above leads to the unification of `operator (com_operator ?C)` with `andb`. This first resolves the outer operator, as above, reducing the problem to unifying `com_operator ?C` with `andb_monoid`, which is finally solved using `andb_comoid`. Hence, `andb_comoid` associates commutativity with the `law andb_monoid` rather than the operator `andb`, and the invisible chain of coercions guides the instance resolution.

The telescope idiom works recursively for arbitrarily deep hierarchies, though the `Monoid` one only has one more level for a distributivity property

```
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.
```

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

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 three-stage resolution.

The coercion chains that support the ease of use of telescope hierarchies have unfortunately two major drawbacks: they limit the shape of the hierarchy to a tree (with linear ancestry) and trigger crippling inefficiencies in the type inference and type checking heuristics for deep hierarchies.

7.3 Packed classes

(★)

The type structure hierarchy for the Mathematical Components library is both deep (up to 11 levels) and non-linear. It is not uncommon for an algebraic structure to combine the properties of two unrelated structures: for example an algebra is both a ring and a module, neither of which is an instance of the other. Thus the Mathematical Components type structures are organised along a different pattern, *packed classes*, which is more flexible and efficient than the telescope pattern, but requires more work to follow.

The packed class design calls for three layers of records for each structure: a *mixin* record holding the new operations and properties the structure adds to the structures it extends (as in section 7.1), a *class* record holding all the primitive operations and properties in the structure, including those in substructures, and finally a *packed class* record that associates the class to a type, and is used to define instances of the structure. Crucially, in this organization the type “key” that directs inference is always a direct field of a structure’s instance record, so all coercion chains have length one.

This arrangement was already hinted at in section 5.5 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 5.5, regroups all the declarations in Equality that should have global scope, such as the `Coercion` declaration for Equality.sort.

The roles of mixin and class are conflated for `eqType` because it sits at the bottom of the type structure hierarchy. To clarify the picture we need to move one level up, to the `choiceType` structure that provides effective choice for decidable predicates:

```

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 main operation provided by the `choiceType` structure `T` is a choice function for decidable predicates (`choose : pred T -> T`) satisfying:

```

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 mixin actually specifies a specific search depth based choice strategy: `choose` can be defined using `find` and countable choice, which is derivable in Calculus of Inductive Constructions².

²behind the `{m|..}` notation lies the `sig` dependent pair. Such inductive data type is a copy of the existential quantifier but is declared as a datatype rather than a proposition [22, section 3.1.4]. COQ checks that the `m` is built without using the existential witness `n`. In other words the proof in input is only used to justify the termination of the search for `m`.

```

1 Lemma find_ex_minn (P : pred nat) :
2   (exists n, P n) -> {m | P m & ∀ n, P n -> n >= m}.

```

The stronger requirement makes it possible to compose `choiceTypes`, so that pairs or sequences of `choiceTypes` are also `choiceTypes`. This subtlety is detailed in the comments of the `choice` file.

An important difference to telescopes is that the definition of `Choice.type` does not link it directly to `eqType`: a `choiceType` structure contains an `Equality.class` 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` \equiv `sort cT` and `class` equal to `base class`. The actual link between `choiceType` and `eqType` is established by the following two lines in `Choice.Exports`:

```

1 Coercion eqType : type >-> Equality.type.
2 Canonical eqType.

```

Line 1 merely lets us explicitly use a `choiceType` where an `eqType` is expected, which is rare as structures are almost 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 $?E = \text{Choice.eqType } T$.

The bottom structure in the Mathematical Components algebraic hierarchy introduced by the `ssralg` library is `zmodType` (`GRing.Zmodule.type`); it encapsulates additive groups, and directly extends the `choiceType` structure.

```

1 Module GRing.
2
3 Module Zmodule.
4
5 Record mixin_of (V : Type) : Type := Mixin {
6   zero : V; opp : V -> V; add : V -> V -> V;
7   _ : associative add;
8   ...}.
9
10 Record class_of T := Class { base: Choice.class_of T; mixin: mixin_of T }.
11 Structure type := Pack {sort; _ : class_of sort}.

```

Strictly speaking, the Mathematical Components algebraic structures don't really *have* to extend `choiceType`, but it is very convenient that they do. We

can use `eqType` and `choiceType` operations to test for 0 in fields, or choose a basis of a subspace, for example. Furthermore, this is essentially a free assumption, because the Mathematical Components algebra mixins specify *strict* identities, such as `associative add` on line 7 above. In the pure Calculus of Inductive Constructions, these can only be realised for concrete data types with a binary representation, which are both discrete and countable, hence are `choiceTypes`. On the other hand, the “classical Calculus of Inductive Constructions” axioms needed to construct, e.g., real numbers, imply that all types are `choiceTypes`.

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 that encloses most of `ssralg`, and also given the usual arithmetic syntax (`0`, `- x`, `x + y`) in the `%R` scope. Only the notations are exported from `GRing`, as these definitions are intended to remain private.

Warning

If you see an undocumented `GRing.something`, then you have broken an abstraction barrier



The next structure in the hierarchy encapsulates nontrivial rings. Imposing the nontriviality 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}.
```


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: 2×2 matrices over \mathbb{Q} have computable inverses but do not commute, while polynomials over \mathbb{Z}_p commute but do not have easily computable inverses. The definition of `comRingType` and `unitRingType` follow exactly the pattern we have seen, except there is no need for a `ComRing.mixin_of` record.

Since there are also rings such as \mathbb{Z}_p that commute *and* have computable inverses, and properties such as $(x/y)^n = x^n/y^n$ that hold only for such rings, `ssralg` provides a `comUnitRingType` structure for them. Although this structure simultaneously extends two unrelated structures, it is easy to define using the packed class pattern: we just reuse the `UnitRing` mixin.

```
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}.
```

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.
```

Lines 1–6 below `comUnitRingType` to each of the 6 structures it extends, just as before. Line 7 is needed because `comUnitRingType` has several direct ancestors in the hierarchy. Making `ComUnitRing.com_unitRingType` a canonical `unitRingType`

instance tells type inference that it can unify `UnitRing.sort ?U` with `ComRing.sort ?C`, by unifying `?U` with `ComUnitRing.com_unitRingType ?R` and `?C` with `ComUnitRing.comRingType ?R`, where `?R : comUnitRingType` is a fresh unification variable. In other words, the `ComUnitRing.com_unitRingType` instance says that `comUnitRingType` is the join of `comRingType` and `unitRingType` in the structure hierarchy (see also [16, Section 5]).

If a new structure S extends structures that are further apart in the hierarchy more than one such additional link may be needed: precisely one for each pair of structures whose join is S . For example, `unitRingAlgebraType` requires three such links, while `finFieldType` in library `finalg` requires 11. It is highly advisable to map out the hierarchy when simultaneously extending multiple structures.

Finally, the telescope and packed class design patterns are not at all incompatible: it is possible to extend a packed class hierarchy with telescopes (library `ringgroup` does this), or to add explicit “join” links to a telescope hierarchy (`ssralg` does this for its algebraic predicate hierarchy).

7.4 Parameters and constructors (★★)

We have noted already that structure instances are often hard to provide explicitly because it is intended they always be inferred. For example the explicit `ringType` structure for `int * rat` is

```
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’ve see in section 5.11 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)` and let type inference fill in the unsightly expression `pair_ringType int_ringType rat_ringType` for `R`.

Inference for constructors is more involved, because it has to produce bespoke classes and mixins subject to dependent typing constraints. While it is in principle possible to program this using dependent matching and transport, the complexity of doing so can be daunting.

Instead, we propose a simpler, static solution using a combination of phantom and function types:

```

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, following the definition of the introduction of section variables `τ` and `cτ`, and the definition of `class`:

```

1 Definition clone c & phant_id class c := @Pack T c.

```

Recall that with the `SSREFLECT` extension to `COQ`, `& τ` introduces an anonymous parameter of type `τ`. As for `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.

Now type checking `(@clone NN _ _ id)` will try to give `id` \equiv `(fun x => x)` the type `(phant_id (Choice.class ?cT) ?c)`. It will assign `x` the type `(phantom (Choice.sort ?cT) (Choice.class ?cT))`, which it will then unify with `(phantom NN ?c)`. To do so `Choice.sort ?cT` will first be unified with `NN`, by setting `?cT` to the canonical instance `nat_choiceType` found by unfolding the definition of `NN`, then setting `?c` to `Choice.class nat_choiceType`.

The code for the instance constructor for `choiceType` is almost identical, be-

cause it only extends `eqType` with a mixin that does not depend on `eqType`. Note that this definition allows COQ to infer τ from m .

```
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 mT m & phant_id m0 m => Pack (@Class T b m).
4 Notation "[ 'ringType' 'of' T ]" := (@pack T _ m _ _ id _ _ id)
```

Type-checking the second `id` will set m to m_0 after checking that the inferred base class $b \equiv \text{Zmodule.class } bT$ coincides with the actual base class b_0 in the structure parameter of the type of m_0 . Forcing the sort of that parameter to be equal to τ allows COQ to infer τ 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 τ . 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).
```

The interested reader can also refer to [16, Section 7] for a description of this technique.

7.5 Linking a custom data type to the library

The sub-type kit of chapter 6 is not the only way to easily add instances to the library. For example imagine we are interested to define the type of a wind rose and attach to it the theory of finite types.

```
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`. Lets 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.
```

Remark how the `inord` constructor lets us postpone the (trivial by computation) proofs that 0, 1, 2, 3 are smaller than 4.

The type of ordinals is larger, hence we provide only a partial function.

```
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 is left on the side. To use `windrose` in conjunction with `\in` and `#|...|`, the type declaration has to be tagged as a `predArgType` as follows.

```
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).
```

More in general a data type can be equipped with a `eqType`, `choiceType`, and `countType` structure by providing a correspondence with the generic tree data type (`GenTree.tree T`): an n -ary tree with nodes labelled with natural numbers and leafs carrying a value in T .

Part III

**Mathematics in
Mathematical Components**

Chapter 8

Numbers

Chapter 9

Polynomials, Linear algebra

Chapter 10

Finite group theory

The part of the library covering finite group theory is composed of the `finGroup` and `solvable` directories. It covers a substantial part of [12] and [1] ranging from basic notions like morphisms, quotients, actions, cyclic and p-groups to more substantial topics as Sylow and Hall groups, the Abelian structure theorem and the Jordan Holder theorem. The paper [15] describes the constructions and formalization techniques needed in order to prove the Jordan Holder theorem.

Basic formalization choices To ease the study of sub-groups, the type of groups is indexed over a container group called `finGroupType`, fixing the group law and imposing finiteness by inheriting from `finType`. For example two groups $(G, H : \{\text{group } gT\})$, where $(gT : \text{finGroupType})$, share the same law and unit. Groups naturally coerce to finite sets, and all set-wise operations are available.

Type inference is programmed to infer the group structure whenever possible. For example the type of $(G : \& : H)$ is $\{\text{set } gT\}$ but such expression is automatically promoted to a group when needed, similarly to what was done in section 6.1 for tuples.

Notations The same infix $*$ notation can be used to multiply both sets (or groups) and their points. For example $(g * h \ \backslash \text{in } G * H)$ is a valid writing, meaning that $g \cdot h \in \{x \cdot y \mid x \in G, y \in H\}$. Given the `finGroupType` container many open notations are also supported, for example the normalizer of A , written $'N(A)$ for A being a $\{\text{set } gT\}$, is a sensible writing.

```
1 Definition normaliser A := [set x | A :^ x \subset A].
2 Notation "'N' ( A )" := (normaliser A) : group_scope.
3 Lemma normP x A : reflect (A :^ x = A) (x \in 'N(A)).
```

Here $^{\cdot}$ is the notation for conjugating a set by a point: $\{x^{-1} * a * x \mid a \in A\}$.

Formalization trick: totality of operations In standard math one can write A/H only if the normality condition $(H <| A)$ holds. Such construction is made total by defining A/H as $((N(H) : \& : A) * H) / H$, i.e. A is intersected

with the normalizer of H , the biggest group that can be quotiented by H . This means that a part of A may be discarded before computing the quotient; as a consequence lemmas about quotients require normality conditions, for example:

```
1 Lemma quotientM1 A B : A \subset 'N(H) -> A * B / H = (A / H) * (B / H).
```

Remark that the equational form lets you require only one precondition: if B exceeds the $'N(H)$ then the equation still holds, since the intersection with the normalizer of H occurs on both sides.

A similar choice is used for the (semi) direct and central product. This time the operation is made total by using a default value, the empty set, when the preconditions are not satisfied.

```
1 Definition partial_product A B :=
2   if A == 1 then B else if B == 1 then A else
3   if [&& group_set A, group_set B & B \subset 'N(A)] then A * B else set0.
4 Definition sdprod A B :=
5   if A :&: B \subsetset 1%G then partial_product A B else set0.
6 Notation "G ><| H" := (sdprod G H)%g (at level 40, left associativity).
```

Note that by equating an expression $(A ><| B)$ with a group one imposes that the pre conditions hold, since a group is never the empty set. Here `group_set` is a predicate asserting that a set is closed under multiplication and contains the unit; `1%G` is the trivial group.

Formalization trick: presentations One cannot decide if a group presented via generators and relations is finite, hence the integration of such concept in a library of finite groups is tricky. The notation $(G \text{ \homg Grp } (x : (x \text{ } ^+ n)))$ states that the finite group G is generated by a single point x of order n (as in the standard mathematical notation, $(x \text{ } ^+ n)$ really means $(x \text{ } ^+ n = 1)$). The shrewdness is that the notation hides an existential quantification postulating the existence of a finite tuple of generators satisfying the equations and building g : the potentially infinite object is never built as well the `\homg` relation, standing for homomorphic image, suggests the fact that G is the image of the presented group, not the group itself. Notation $(G \text{ \homg Grp } (x_1 : .. x_n : (s_1 = t_1, .., s_m = t_m)))$ for a $(G : \{group\} gT)$ unfolds to:

```
1 [exists t : gT * .. * gT, let: (x_1, .., x_n) := t in
2   (<[x_1]> <*> .. <*> <[x_n]>, (s_1, .. (s_m-1, s_m) ..))
3   == (G, (t_1, .. (t_m-1, t_m) ..))]
```

Such formula is generated reflexively, i.e. by a COQ program that takes in input the syntax of the presentation and produces that statement. Remark the $m + 1$ components compared by $(_ == _)$. It first compares the group generated by the generators $x_1 \dots x_n$ with G ; then it compares all the expressions being related. Here $\langle [x] \rangle$ denotes the group generated by x and $(A <*> B)$ is the group generated by the sets A and B (also called join). An example is the standard presentation of the dihedral group ($'D_m$ has order $m = q.*2 >= 4$):

```
1 Lemma Grp_dihedral: 'D_m \isog Grp(x: y: (x ^+ q, y ^+ 2, x ^ y = x^-1)).
```

Chapter 11

Representation and Character theory

[14]

As an example of application, in particular of the linear algebra theory.

Chapter 12

Galois Theory

Chapter 13

Real closed fields

1. real closure of countable fields
2. direct construction of algebraic numbers
3. real closure of archimedean fields
4. construction of the algebraic closure of a Rcf
5. decidability of first order theory of real closed fields

Part IV

Indexes

Index: Concepts

- abbreviation, 33
- binder, 14
- case analysis, 45
 - naming, 46
- coercion, 100
- computation, 16, 22, 44
 - symbolic, 31
- consistency, 78
- convertibility, 69
- currying of functions, 17

- dependent function space, 69
- dependent type, 81

- equality, 39, 71

- formal proof, 43
- forward reasoning, 93, 95
- function application, 14
- functional extensionality, 141

- general term, *see also* higher order, 58
- goal stack model, 73
- ground equality, 39

- higher order, 16, 58

- identity, 41
- implicit argument, 27
- improving **by** \square , 54
- induction, 57
 - curse of values, 79
 - generalizing, 59
 - strong, 79
- inductive type, 17
 - constructor, 17
 - destructor, 19

- keyed matching, 63

- list comprehension, 29

- machine checked proof, 44
- matching algorithm, 62

- natural number, 18
- notation, 33
- notation scope, 29

- partial application, 17
- pattern, 50
- pattern matching, 20
 - exhaustiveness, 21
 - irrefutable, 35
- polymorphism, 26
- positivity, 79
- proof irrelevance, 135, 138
- proposition, 39
- provide choiceType structure, 164
- provide eqType structure, 164
- provide finType structure, 164

- record, 34
- recursion, 21
 - termination, 22
- reordering goals, 48
- rewrite rule, 50

rewriting, [50](#)

search, [60](#)

section, [30](#)

simplifying equation, [99](#)

symmetric argument, [95](#)

termination, [22](#), [78](#)

type, [15](#)

type error, [15](#)

typing an application, [17](#)

unfolding equation, [99](#)

well typed, [15](#)

Index: Ssreflect tactics

`apply`., 54
`by` [], 44
`case`: name => [m], 47
`case`: name, 45
`do` n! (iteration), 80
`elim`: name => [m IHm], 58
`gen have` name : name / type, 147
`have` name : type `by` tactic, 94
`have` name : type, 94
`last first`, 48
`rewrite`, 50
 // (close trivial goals), 57
 /= (simplification), 62
 ! (iteration), 51
 -/ (folding), 64
 -[term]/(term) (changing), 64
 - (right-to-left), 51
 / (unfolding), 62
 ? (optional iteration), 62
 [in RHS] (focusing), 62
 {name} (clear), 64
`set` name := term, 79
`suff` also suffices, 97
`tactic` : ..., 76
 : {name} (disposal), 77
 : term (generalization), 77
 name: term (equation), 78
`tactic` => ..., 73
 => -> (rewriting L2R), 76
 => /(_ arg) (specialization), 76
 => // (close trivial goals), 74
 => /= (simplification), 74
 => /view/view (many views), 75
 => /view (view application), 74
 => <- (rewriting R2L), 76
 => [..| ..] (case), 74
 => {name} (disposal), 74
`tactic in` name, 99
`wlog` also without loss, 97

Index: Coq terms

($_ \ * \ _$) (pairs), 37
 ($_ \ ++ \ _$), 29
 ($_ \ , \ _$), 37
 ($_ \ . * 2$), 24
 ($_ \ . + 1$), *see also* s
 ($_ \ . - 1$), 24
 ($_ \ :: \ _$), 27
 ($_ \ = 1 \ _$), 143
 ($_ \ = P \ _$), *see also* eqP , 148
 AddLaw, 156
 ComLaw, 156
 EqMixin, 152
 EqType, 152
 Equality.sort, 152
 Equality.type, 152
 False, 71
 GenTree.tree, 165
 I, 71
 Law, 155
 0, 19
 PcanEqMixin, 152
 S, 19
 True, 71
 [$\&\& \ \dots \ , \ \dots \ \& \ \dots$], 27
 [$:: \ \dots \ , \ \dots \ \& \ \dots$], 27
 [$=> \ \dots \ , \ \dots \ => \ \dots$], 27
 [eqMixin of \dots by $<:$], 152
 [eqType of \dots], 152
 [seq \dots ; \dots], 27
 [seq \dots <- \dots | \dots], 29
 [seq \dots | \dots <- \dots], 29
 [| \dots , \dots | \dots], 27
 \big[\dots / \dots](\dots | \dots) \dots , 123
 \sum, 32, 123
 addSn, 45
 addn, 22
 add, 31
 andP, 86
 andb, 18
 associative, 53
 block_mx, 148
 cancel, 54
 castmxKV, 148
 castmx_id, 148
 castmx, 148
 choiceType, 158
 classically_EM, 80
 classically_bind, 80
 classically, 80
 col_mx, 148
 comRingType, 161
 com_unitRingType, 161
 commutative, 53
 conform_mx, 149
 conj, 70
 contraL, 54
 elimTF, 88
 eqP, 118
 eqType, 152, 158
 eqnP, 86
 erefl, 71
 ex_intro, 70
 fix, 99
 foldr, 30
 foralll, 26
 fun \dots => \dots , 14
 idP, 87
 if \dots is \dots then \dots else \dots , 20

`if .. then .. else ..`, 18
`ifP`, 92
`iffP`, 87
`implyP`, 86
`injective`, 54
`iota`, 32
`isT`, 69
`iter`, 30
`left_distributive`, 53
`left_id`, 53
`leqP`, 92
`leqn0`, 46
`leq`, 24
`let .. := .. in ..`, 34
`let: .. := .. in ..`, 35
`lsubmx`, 147
`ltngtP`, 92
`map`, 28
`match .. with .. end`, 19
`muln_eq0`, 47
`muln`, 47
`nat_ind`, 78

`nat`, 19
`negbK`, 45
`not`, 71
`option`, 29
`orP`, 86
`or_introl`, 71
`or_intror`, 71
`pcancel`, 54
`preArgType`, 165
`pred` (predecessor), *see also* `.-1`
`pred` (predicate), 137
`reflect`, 85, 91
`rel`, 53
`ringType`, 160
`row_mx`, 148
`size_map`, 53
`size`, 28
`subn`, 23
`ulsubmx`, 147
`unitRingType`, 161
`usubmx`, 147
`zmodType`, 159

Index: Coq commands

About, 15
Admitted, 41
Arguments, 91
Check, 15
Definition, 14
Eval compute, 15
Fixpoint, 22
Hint Resolve, 54
Implicit Type, 30
Inductive, 17, 70
Lemma, 41
Locate, 19
Notation, 33
Print, 15
Record, 34
Section, 30
Theorem, 41
Variable, 30

Bibliography

- [1] M. Aschbacher. *Finite Group Theory*. Cambridge University Press, second edition, 2000. Cambridge Books Online.
- [2] Andrea Asperti, Wilmer Ricciotti, Claudio Sacerdoti Coen, and Enrico Tassi. A bi-directional refinement algorithm for the Calculus of (Co)Inductive Constructions. *Logical Methods in Computer Science*, 8(1), 2012.
- [3] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Springer, 2004.
- [4] Adam Chlipala. *Certified Programming with Dependent Types*. The MIT Press, 2014.
- [5] Cyril Cohen. Construction of real algebraic numbers in coq. In *Interactive Theorem Proving - Third International Conference, ITP 2012, Princeton, NJ, USA, August 13-15, 2012. Proceedings*, pages 67–82, 2012.
- [6] Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. Cubical Type Theory: a constructive interpretation of the univalence axiom, 2015. Preprint.
- [7] François Garillot, Georges Gonthier, Assia Mahboubi, and Laurence Rideau. Packaging Mathematical Structures. In Tobias Nipkow and Christian Urban, editors, *Theorem Proving in Higher Order Logics*, volume 5674 of *Lecture Notes in Computer Science*, Munich, Germany, 2009. Springer.
- [8] Georges Gonthier. Formal Proof – The Four-Color Theorem. *Notices of the American Mathematical Society*, 55(11):1382–1393, December 2008.
- [9] Georges Gonthier, Andrea Asperti, Jeremy Avigad, Yves Bertot, Cyril Cohen, François Garillot, Stéphane Le Roux, Assia Mahboubi, Russell O'Connor, Sidi Ould Biha, Ioana Pasca, Laurence Rideau, Alexey Solovyev, Enrico Tassi, and Laurent Théry. A Machine-Checked Proof of the Odd Order Theorem. In Sandrine Blazy, Christine Paulin, and David Pichardie, editors, *ITP 2013, 4th Conference on Interactive Theorem Proving*, volume 7998 of *LNCS*, pages 163–179, Rennes, France, July 2013. Springer.

- [10] 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.
- [11] Georges Gonthier, Beta Ziliani, Aleksandar Nanovski, and Derek Dreyer. How to make ad hoc proof automation less ad hoc. *J. Funct. Program.*, 23(4):357–401, 2013.
- [12] D. Gorenstein. *Finite Groups*. AMS Chelsea Publishing Series. 2007.
- [13] Michael Hedberg. A coherence theorem for Martin-Löf’s Type Theory. *J. Funct. Program.*, 8(4):413–436, July 1998.
- [14] I.M. Isaacs. *Character Theory of Finite Groups*. AMS Chelsea Pub. Series. 1976.
- [15] Assia Mahboubi. The rooster and the butterflies. In *Intelligent Computer Mathematics - MKM, Calculemus, DML, and Systems and Projects 2013, Held as Part of CICM 2013, Bath, UK, July 8-12, 2013. Proceedings*, pages 1–18, 2013.
- [16] Assia Mahboubi and Enrico Tassi. Canonical Structures for the working Coq user. In Sandrine Blazy, Christine Paulin, and David Pichardie, editors, *ITP 2013, 4th Conference on Interactive Theorem Proving*, volume 7998 of *LNCS*, pages 19–34, Rennes, France, July 2013. Springer.
- [17] Rob Nederpelt, Herman Geuvers, and Roel de Vrijer, editors. *Selected Papers on Automath*, volume 133 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, 1994.
- [18] Christine Paulin-Mohring. Introduction to the Calculus of Inductive Constructions. In Bruno Woltzenlogel Paleo and David Delahaye, editors, *All about Proofs, Proofs for All*, volume 55 of *Studies in Logic (Mathematical logic and foundations)*. College Publications, January 2015.
- [19] Benjamin C. Pierce, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hrițcu, Vilhelm Sjöberg, and Brent Yorgey. *Software Foundations*. Electronic textbook, 2015.
- [20] Robert Pollack. Implicit syntax. In *Informal Proceedings of First Workshop on Logical Frameworks*, page pages, 1992.
- [21] Matthieu Sozeau and Beta Ziliani. Towards a better behaved unification algorithm for Coq. In *Proceedings of The 28th International Workshop on Unification*, 2014.
- [22] The Coq development team. *The Coq proof assistant reference manual*. Inria.

- [23] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. <https://homotopytypetheory.org/book>, Institute for Advanced Study, 2013.