Programs and Proofs

Mechanizing Mathematics with Dependent Types

Lecture Notes

Ilya Sergey

Draft of April 27, 2020

Contents

1 Introduction

These lecture notes are the result of the author's personal experience of learning how to structure formal reasoning using the Coq proof assistant and employ Coq in large-scale research projects. The present manuscript offers a brief and practically-oriented introduction to the basic concepts of mechanized reasoning and interactive theorem proving.

The primary audience of this text are the readers with expertise in software development and programming and knowledge of discrete mathematic disciplines on the level of an undergraduate university program. The high-level goal of the course is, therefore, to demonstrate how much the rigorous mathematical reasoning and development of robust and intellectually manageable programs have in common, and how understanding of common programming language concepts provides a solid background for building mathematical abstractions and proving theorems formally. The low-level goal of this course is to provide an overview of the Coq proof assistant, taken in its both incarnations: as an expressive functional programming language with dependent types and as a proof assistant providing support for mechanized interactive theorem proving.

By aiming for these two goals, this manuscript is, thus, intended to provide a demonstration how the concepts familiar from the mainstream programming languages and serving as parts of good programming practices can provide illuminating insights about the nature of reasoning in Coq's logical foundations and make it possible to reduce the burden of mechanical theorem proving. These insights will eventually give the reader a freedom to focus solely on the *essential* part of her formal development instead of fighting with a proof assistant in futile attempts to encode the "obvious" mathematical intuition—a reason that made many of the new-comers abandon their attempts to apply the machine-assisted approach for formal reasoning as an everyday practice.

1.1 Why yet another course on Coq?

The Coq proof assistant [10] has been in development since 1983, and by now there is a number of courses that provide excellent introductions into Coq-powered interactive theorem proving and software development. Among the other publicly available manuscripts, the author finds the following three to be the most suitable for teaching purposes.

- The classical book *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions* by Yves Bertot and Pierre Castéran [3] is a great and exhaustive overview of Coq as a formal system and a tool, covering both logical foundations, reasoning methodologies, automation tools and offering large number of examples and exercises (from which this course borrows some).

- Benjamin Pierce et al.'s *Software Foundations* electronic book [53] introduces Coq development from an angle of the basic research in programming languages, focusing primarily on formalization of program language semantics and type systems, which serve both as main motivating examples of Coq usage and a source of intuition for explaining Coq's logical foundations.
- The most recently published book, *Certified Programming with Dependent Types* by Adam Chlipala [7] provides a gentle introduction to Coq from the perspective of writing programs that manipulate *certificates*, i.e., first-class proofs of the program's correctness. The idea of certified programming is a natural fit for a programming language with dependent types, which Coq offers, and the book is structured as a series of examples that make the dependently-typed aspect of Coq shine, along with the intuition behind these examples and a detailed overview of state-of-the-art *proof automation* techniques.

Although all the three books have been used in numerous introductory courses for Coq with a large success, it is the author's opinion that there are still some topics essential for grasping the intuition behind rigorous and boilerplate-free mathematical reasoning via a proof assistant that are left underrepresented. This course is targeted to fill these gaps, while giving the reader enough background to proceed as a Coq hacker on her own. In particular, this manuscript describes in detail the following aspects of proof engineering, most of which are enabled or empowered by Gonthier et al.'s *small-scale reflection* extension (Ssreflect) to Coq [23] and its accompanying library called Mathematical Components:

- Special treatment is given to the *computational* nature of inductive reasoning about *decidable* propositions, which makes it possible to compute a result of the vast majority of them (as opposed to prove them constructively) as a boolean value, given that they are formulated as computable recursive Coq functions, rather than inductive predicates (which is more in the spirit of the traditional Coq school).
- Instead of supplying the reader with a large vocabulary of tactics necessary for everyday Coq hacking, this course focuses on a *very small* but expressive set of proof constructing primitives (of about a seven in total), offered by Ssreflect or inherited from the vanilla Coq with notable enhancements.
- This course advocates inductive types' *parameters* as an alternative to *indices* as a way of reasoning about explicit equalities in datatypes with constraints.
- The reasoning by rewriting is first presented from the perspective of Coq's definition of propositional equality and followed by elaboration on the idea of using *datatype indices* as a tool to define client-specific conditional *rewriting rules*.
- This manuscript explains the essentials of Ssreflect's *boolean reflection* between the sort `Prop` and the datatype *bool* as a particular case of conditional rewriting, following the spirit of the computational approach to the proofs of decidable propositions.
- Formal encoding of familiar mathematical structures (e.g., monoids and lattices) is presented by means of Coq's *dependent records* and overloading mathematical operations using the mechanism of *canonical instances*.

- A novel (from a teaching perspective) case study is considered, introducing the readers to the concepts of Hoare Type Theory and describing the basics of type-based reasoning about *imperative programs* by means of *shallow embedding*.

1.1.1 What this course is about

Besides the enumerated above list of topics, which are described in detail and supported by a number of examples, this course supplies some amount of "standard" material required to introduce a reader with a background in programming and classical mathematical disciplines to proof engineering and program development in Coq. It starts from explaining how simple functional programs and datatypes can be defined and executed in the programming environment of Coq, proceeding to the definition of propositional logic connectives and elements of interactive proof construction. Building further on the programming intuitions about algebraic datatypes, this manuscript introduces a definition of propositional equality and the way to encode custom rewriting rules, which then culminates with a discussion on the boolean reflection and reasoning by means of computation. This discussion is continued by revising important principles of proofs by induction in Coq and providing pointers to the standard Ssreflect libraries, which should be used as a main component for everyday mathematical reasoning. The course concludes by reconciling all of the described concepts and Coq/Ssreflect reasoning principles by tackling a large case study—verifying imperative programs within the framework of Nanevski et al.'s Hoare Type Theory [41, 42].

1.1.2 What this course is not about

There is a range of topics that this course does not cover, although it is the author's belief that the provided material should be sufficient for the reader to proceed to these more advanced subjects on her own. Some of the exciting topics, which are certainly worth studying but lie beyond the scope of this manuscript, are listed below together with pointers to the relevant bibliographic references.

- Reasoning about infinite objects in Coq via co-induction (see Chapters 5 and 7 of the book [7] as well as the research papers [32, 34]).
- Proof automation by means of tactic engineering (see [7, Chapters 13–15] and the papers [61, 62, 70]) or lemma overloading [25].
- Using a proof assistant in the verification of program calculi [2, 53] and optimizing compilers [1] as well as employing Coq to specify and verify low-level and concurrent programs [5, 8, 17, 44].

1.1.3 Why Ssreflect?

A significant part of this course's material is presented using the Ssreflect extension of Coq [23] and its accompanying libraries, developed as a part of the Mathematical Components project[1] in order to facilitate the automated reasoning in very large mathematical

[1] https://math-comp.github.io/math-comp/

developments, in particular, the fully formal machine-checked proofs of the *four color theorem* [22] and *Feit-Thompson (odd order) theorem* [24].

Ssreflect includes a small set of powerful novel primitives for interactive proof construction (tactics), different from the traditional set provided by Coq. It also comes with a large library of various algebraic structures, ranging from natural numbers to graphs, finite sets and algebras, formalized and shipped with exhaustive toolkits of lemmas and facts about them. Finally, Ssreflect introduces some mild modifications to Coq's native syntax and the semantics of the proof script interpreter, which makes the produced proofs significantly more concise.

Using Ssreflect for the current development is not the goal by itself: a large part of the manuscript could be presented using traditional Coq without any loss in the insights but, perhaps, some loss in brevity. However, what is more important, using Ssreflect's libraries and tactics makes it much easier to stress the main points of this course, namely, that (a) the proof construction process should rely on Coq's native computational machinery as much as possible and (b) rewriting (in particular, by equality) is one of the most important proof techniques, which should be mastered and leveraged in the proofs. Luckily, the way most of the lemmas in Ssreflect and Mathematical Components libraries are implemented makes them immediately suitable to use for rewritings, which directly follows the natural mathematical intuition. The enhancements Ssreflect brings over the standard Coq rewriting machinery also come in handy.

Last, but not least, Ssreflect comes with a much improved `Search` tool (comparing to the standard one of Coq). Given that a fair part of time spent for development (either programs and proofs) is typically dedicated to reading and understanding the code (or, at least, specifications) written by other implementors, the `Search` tool turns out to be invaluable when it comes to looking for necessary third-party facts to employ in one's own implementation.

In the further chapters of this course, we will not be making distinction between native Coq and Ssreflect-introduced commands, tactics and tacticals, and will keep the combined lists of them in the Index section at the end of the manuscript.

1.2 Prerequisites

The reader is expected to have some experience with mainstream object-oriented and functional programming languages, such as Scala [46], Haskell [31], OCaml [35] or Standard ML [39]. While strong knowledge of any of the mentioned languages is not mandatory, it might be useful, as many of the Coq's concepts making appearance in the course are explained using the analogies with constructs adopted in practical programming, such as algebraic datatypes, higher-order functions, records and monads.

While this manuscript is aiming to be self-contained in its presentation of a subset of Coq, it would be naïve to expect it to be the *only* Coq reference used for setting-up a formal development. That said, we encourage the reader to use the standard Coq manual [10] as well as Ssreflect documentation [23] whenever an unknown tactic, piece of syntax or obscure notation is encountered. Coq's `Search`, `Locate` and `Print` tools, explained in Chapter 2 are usually of great help when it comes to investigating what someone's Coq code does, so don't hesitate to use them.

Finally, we assume that the Emacs text editor with a Proof General mode installed (as explained further in this chapter) will be used as the environment for writing code scripts, and the GNU `make` machinery is available on the reader's machine in order to build the necessary libraries and tools.

1.3 Setup

In order to be able to follow the manuscript and execute the examples provided, the reader is expected to have Coq with Ssreflect installed on her machine. This section contains some general instructions on the installation and set-up. Most of the mentioned below sources can be downloaded from the following URL, accompanying these notes:

http://ilyasergey.net/pnp

Alternatively, you can clone the sources of these lecture notes, along with the exercises and the solution from the following public GitHub repository:

https://github.com/ilyasergey/pnp

1.3.1 Installing Coq, Ssreflect and Mathematical Components

The sources of this manuscript have been compiled and tested with Coq version 8.11.1, Ssreflect/Mathematical Components version 1.10.0, and FCSL PCM version 1.2.0. It is not guaranteed that the same examples will work seamlessly with different versions.

The easiest way to obtain the necessary versions of Coq and the libraries is to install them via the OPAM package manager (https://opam.ocaml.org):

```
opam install coq.8.11.1
```

In order to install Ssreflect/Mathematical Components and FCSL PCM, you will need to register the corresponding repository and then install the packages as follows:

```
opam repo add coq-released https://coq.inria.fr/opam/released
opam install coq-mathcomp-ssreflect.1.10.0 coq-fcsl-pcm.1.2.0
```

1.3.2 Emacs set-up

The Emacs[2] (or Aquamacs[3] for macOS users) text editor provides a convenient environment for Coq development, thanks to the Proof General mode. After downloading and installing Emacs, clone the Git repository of Proof General,[4] and Ssreflect/Mathematical Components[5] following the instructions below. Upon cloning both repositories, for instance, into the folders `~/misc/PG/` and `~/misc/math-comp-1.8.0/`, add the following lines into the `.emacs` configuration file located in the home directory in Unix and in the `C:\` root in Windows (possibly replacing the `~/misc/` part with the path where the Proof General and Ssreflect/Mathematical Components repositories were).

[2] http://www.gnu.org/software/emacs/
[3] http://aquamacs.org
[4] https://github.com/ProofGeneral/PG
[5] https://github.com/math-comp/math-comp

```
;; Proof General support
(load-file "~/misc/PG/generic/proof-site.el")

;; Ssreflect support
(load-file "~/misc/math-comp-1.8.0/mathcomp/ssreflect/pg-ssr.el")
```

Linux users who are more used to the Windows-style Copy/Paste/Undo keystrokes can also find it convenient to enable the Cua mode in Emacs, which can be done by adding the following lines into the `.emacs` file:

```
(cua-mode t)
(setq cua-auto-tabify-rectangles nil)
(transient-mark-mode 1)
(setq cua-keep-region-after-copy t)
```

Every Coq file has the extension `.v`. Opening any `.v` file will automatically trigger the Proof General mode.

Finally, the optional Company-Coq[6] collection of extensions to Proof General adds many modern IDE features such as auto-completion of tactics and names, refactoring, and inline help.

1.3.3 Getting the lecture files and solutions

The reader is encouraged to download the additional material for this course in the form of Coq files with all examples from the manuscript plus some additional exercises. The sources can be obtained from the GitHub repository. The Coq files accompanying lectures (with solutions omitted) are contained in the `lectures` folder. For the examples of Chapter 8 and the corresponding lecture source file, the sources of the Hoare Type Theory (HTT) development will be required. The current version of the notes includes the ready-to-use up-to-date sources of HTT in the folder `htt`. Solutions for all of the exercises can be found in the folder `solutions` of the GitHub project accessible by the link above.

After the sources are cloned, run `make` from the root folder. This will build all necessary libraries, lectures, solutions for the exercises, and the lecture notes. The resulting PDF file is `latex/pnp.pdf`.

The table below describes the correspondence between the chapters of the manuscript and the accompanying files.

№	Chapter title	Coq file
2	Functional Programming in Coq	`FunProg.v`
3	Propositional Logic	`LogicPrimer.v`
4	Equality and Rewriting Principles	`Rewriting.v`
5	Views and Boolean Reflection	`BoolReflect.v`
6	Inductive Reasoning in Ssreflect	`SsrStyle.v`
7	Encoding Mathematical Structures	`DepRecords.v`
8	Case Study: Program Verification in Hoare Type Theory	`HTT.v`

[6] https://github.com/cpitclaudel/company-coq

1.4 Naming conventions

Coq as a tool and environment for interactive theorem proving incorporates a number of entities in itself. As a programming and specification language, Coq implements a dependently-typed *calculus* (i.e., a small formal programming language) *Gallina*, which is an extension of the *Calculus of Inductive Constructions* (CIC) explained in Chapter 3. Therefore, all the expressions and programs in Coq, including standard connectives (e.g., `if-then-else` or `let-in`) are usually referred to as *Gallina terms*. In the listing, keywords of Gallina terms will be usually spelled using `typewriter monospace font`. The defined entities, such as functions, datatypes theorems and local variables will be usually spelled in the *italic* or sans serif fonts.

On top of the language of programs in Coq there is a language of *commands* and *tactics*, which help to manage the proof scripts, define functions and datatypes, and perform queries, such as searching and printing. The language of Coq commands, such as `Search` and `Print`, is called *Vernacular*. Commands and tactics, similarly to the keywords, are spelled in `typewriter monospace font`.

In the rest of the manuscript, though, we will be abusing the terminology and blur the distinction between entities that belong to Gallina, Vernacular or Coq as a framework, and will be referring to them simply as "Coq terms", "Coq tactics" and "Coq commands".

In the program displays, interleaving with the text, some mathematical symbols, such as $\forall$, $\exists$ and $\rightarrow$, will be displayed in Unicode, whereas in the actual program code they are still spelled in ASCII, e.g., `forall`, `exists` and `->`, correspondingly.

1.5 Acknowledgements

This course was inspired by the fantastic experience of working with Aleks Nanevski on verification of imperative and concurrent programs during the author's stay at IMDEA Software Institute. Aleks' inimitable sense of beauty when it comes to formal proofs has been one of the main principles guiding the design of these lecture notes.

I'm grateful to Michael D. Adams, Amal Ahmed, Jim Apple, Daniil Berezun, Giovanni Bernardi, Dmitri Boulytchev, William J. Bowman, Kirill Bryantsev, Santiago Cuellar, Andrea Cerone, Olivier Danvy, Rémy Haemmerle, José Francisco Morales, Phillip Mates, Gleb Mazovetskiy, Anton V. Nikishaev, Karl Palmskog, Daniel Patterson, Anton Podkopaev, Leonid Shalupov, Kartik Singhal, Jan Stolarek, Anton Trunov and James R. Wilcox who provided a lot of valuable feedback and found countless typos in earlier versions of the notes.

The mascot picture *Le Coq Mécanisé* on the front page is created by Lilia Anisimova.

2 Functional Programming in Coq

Our journey to the land of mechanized reasoning and interactive theorem proving starts from observing the capabilities of Coq as a programming language.

Coq's programming component is often described as a *functional* programming language, since its programs are always pure (i.e., not producing any sort of side effects), possibly higher-order functions, which means that they might take other functions as parameters and return functions as results. Similarly to other functional programming languages, such as Haskell, OCaml or Scala, Coq makes heavy use of algebraic datatypes, represented by a number of possibly recursive constructors. Very soon, we will see how *programming* with inductive algebraic datatypes incorporates *reasoning* about them, but for now let us take a short tour of Coq's syntax and define a number of simple programs.

2.1 Enumeration datatypes

Let us create an empty `.v` file—a standard extension for Coq files, recognized, in particular, by Proof General, and define our first Coq datatype. The simplest datatype one can imagine is **unit**, a type inhabited by exactly one element. In Coq, one can define such a type in the following manner:[1]

```
Inductive unit : Set := tt.
```

The definition above postulates that the type **unit** has *exactly* one constructor, namely, `tt`. In the type theory jargon, which we will adopt, it is said that the expression `tt` *inhabits* the **unit** type. Naturally, it is the only inhabitant of the set, corresponding to the **unit** type. We can now check the `tt`'s affiliation via the `Check` command:

```
Check tt.
```

```
tt
     : unit
```

Moreover, we can make sure that the **unit** datatype itself defines a set:

```
Check unit.
```

```
unit
     : Set
```

In fact, since Coq makes the analogy between sets and types so transparent, it is not difficult to define a type describing the *empty* set:

[1] Use the `Ctrl-C Ctrl-Enter` keyboard shortcut to initiate the interactive programming/proof mode in Proof General and gradually compile the file.

```
Inductive empty : Set := .
```

That is, the empty set is precisely described by the type, values of which we simply *cannot construct*, as the type itself does *not* provide any constructors! In fact, this observation about inhabitance of types/sets and the definition of an empty type will come in quite handy very soon when we will be talking about the truth and falsehood in the setting of the Curry-Howard correspondence in Chapter 3. Unfortunately, at this moment there is not so much we can do with such simple types as **unit** or **empty**, so we proceed by defining some more interesting datatypes.

The type **bool** is familiar to every programmer. In Coq, it is unsurprisingly defined by providing exactly two constructors: true and false. Since **bool** is already provided by the standard Coq library, we do not need to define it ourselves. Instead, we include the following modules into our file using the `From ... Require Import` command:[2]

```
From mathcomp
Require Import ssreflect ssrbool.
```

Now, we can inspect the definition of the **bool** type by simply printing it:

```
Print bool.
```

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

Let us now try to define some functions that operate with the bool datatype ignoring for a moment the fact that most of them, if not all, are already defined in the standard Coq/Ssreflect library. Our first function will simply negate the boolean value and return its opposite:

```
Definition negate b :=
  match b with
  | true ⇒ false
  | false ⇒ true
  end.
```

The syntax of Coq as programming language is very similar to Standard ML. The keyword `Definition` is used to define non-recursive values, including functions. In the example above, we defined a function with one argument b, which is being scrutinized against two possible value patterns (true and false), respectively, and the corresponding results are returned. Notice that, thanks to its very powerful type inference algorithm, Coq didn't require us to annotate neither the argument b with its type, nor the function itself with its result type: these types were soundly inferred, which might be confirmed by checking the overall type of negate, stating that it is a function from **bool** to **bool**:

```
Check negate.
  negate : bool → bool
```

[2] The From ... premise is optional, and in this particular case it allows to include libraries from *mathcomp* without additional qualifiers.

2.2 Simple recursive datatypes and programs

At this point we have seen only very simple forms of inductive types, such that all their inhabitants are explicitly enumerated (e.g., **unit** and **bool**). The next type used ubiquitously in the computations and mathematical reasoning are natural numbers, the first *truly* inductive datatype. Following the Peano axioms, the type **nat** of natural numbers is defined by induction, i.e., via the following two constructors:

```
Print nat.
```

```
Inductive nat : Set := O : nat | S : nat → nat
```

The definition of the type **nat** is *recursive*. It postulates that O is a natural number (hence, the first constructor), and, if n is a natural number then $S\ n$ is a natural number as well (hence, the name S, which is a shortcut for *successor*). At this point, the reader can recall the notion of *mathematical induction*, usually introduced in school and postulating that if a statement P has to be proven to hold over *all* natural numbers, it should be proven to hold on zero *and* if it holds for n, then it should hold for $n + 1$. The very same principle is put into the definition of the natural numbers themselves. In the future, we will see many other interesting data structures going far beyond natural numbers and each equipped with its own *induction principle*. Moreover, quite soon we will see that in Coq recursive definitions/computations and inductive proofs are in fact two sides of the same coin.

For now, let us write some functions dealing with natural numbers. In order to work conveniently with the elements of type **nat**, we will import yet another Ssreflect library:

```
From mathcomp
Require Import ssrnat.
```

Probably, the most basic function working on natural numbers is their addition. Even though such function is already implemented in the vast majority of the programming languages (including Coq), let us do it from scratch using the definition of **nat** from above. Since **nat** is a recursive type, the addition of two natural numbers n and m should be defined recursively as well. In Coq, recursive functions are defined via the keyword `Fixpoint`. In the following definition of the my_plus function, we will make use of Ssreflect's postfix notation .+1 (with no spaces between the characters) as an alternative to the standard **nat**'s recursive constructor S.[3] Also, Coq provides a convenient notation 0 for the *zero* constructor O.

```
Fixpoint my_plus n m :=
 match n with
 | 0 ⇒ m
 | n'.+1 ⇒ let: tmp := my_plus n' m in tmp.+1
 end.
```

Here, we deliberately used less concise notation in order to demonstrate the syntax `let:` *x* := *e1* `in` *e2* construct, which, similarly to Haskell and OCaml, allows one to bind

[3] It is important to bear in mind that `.+1` is not just a function for incrementation, but also is a datatype constructor, allowing one to obtain the Peano successor of a number `n` by taking `n.+1`.

intermediate computations within expressions.[4] The function my_plus is recursive on its *first* argument, which is being decreased in the body, so *n'* is a *predecessor* of *n*, which is passed as an argument to the recursive call. We can now check the result of evaluation of my_plus via Coq's `Eval compute in` command:[5]

```
Eval compute in my_plus 5 7.
  = 12 : nat
```

The same function could be written quite a bit shorter via Ssreflect's pattern-matching `if-is`-notation, which is a convenient alternative to pattern matching with only two alternatives:

```
Fixpoint my_plus' n m := if n is n'.+1 then (my_plus' n' m).+1 else m.
```

At this point, the reader might have an impression that the computational language of Coq is the same as of OCaml and Haskell, so all usual tricks from the functional programming might be directly applicable. Unfortunately, it is not so, and the main difference between Coq and other general-purpose programming languages stems from the way it treats recursion. For instance, let us try to define the following "buggy" addition function, which goes into an infinite recursion instead of producing the value, due to the fact that the recursion argument is not decreasing and remains to be *n*:

```
Fixpoint my_plus_buggy n m :=
    if n is n'.+1 then (my_plus_buggy n m).+1 else m.
```

we immediately get the following error out of the Coq interpreter:

```
Error: Cannot guess decreasing argument of fix.
```

This is due to the fact that the recursion in *my_plus_buggy* is not *primitive*: that is, there is a recursive call, whose argument is not "smaller" comparing to the initial function's arguments *n* or *m*, which makes this procedure to fall into a larger class of *generally recursive* programs. Unlike primitively-recursive programs, generally-recursive programs may not terminate or terminate only on a subset of their inputs, and checking termination statically in general is an undecidable problem (that is, such checking will not terminate by itself, which is known under the name of Turing's *halting problem*).[6]

The check for primitive recursion, which implies termination, is performed by Coq *syntactically*, and the system makes sure that there is at least one argument of an inductively-defined datatype, which is being consistently decreased at each function call.[7] This criteria is sufficient to ensure the termination of all functions in Coq. Of course, such termination check is a severe restriction to the computational power of Coq, which there-

[4] The same example also demonstrates the use of Ssreflect alternative to Coq's standard `let` command, not trailed with a colon. We will be making use of Ssreflect's `let:` consistently, as it provides additional benefits with respect to in-place pattern matching, which we will see later.

[5] The command in evaluation might look a bit verbose in this form, but it is only because of its great flexibility, as it allows for different evaluation strategies. In this case we employed `compute`, as it performs all possible reductions.

[6] The computability properties of primitively and generally recursive functions is a large topic, which is essentially orthogonal to our development, so we omit a detailed discussion on the theory of recursion.

[7] Sometimes, it is possible to "help" Coq to guess such argument using the explicit annotation `struct` right after the function parameter list, e.g., {`struct` *n*} in the case of my_plus.

fore is not Turing-complete as a programming language (as it supports only primitive recursion).

Although Coq is equipped with an amount of machinery to *reason* about potentially non-terminating programs and prove some useful facts about them[8] (for example, Chapter 7 of the book [7] provides a broad overview of methods to encode potentially non-terminating programs in Coq and reason about them), it usually requires some ingenuity to execute generally-recursive computations within Coq. Fortunately, even without the possibility to *execute* any possible program in the system, Coq provides a rich tool-set to *encode* such programs, so a number of statements could be proved about them (as we will see in Chapter 8), and the encoded programs themselves could be later *extracted* into a general-purpose language, such as Haskell or OCaml in order to be executed (see [3, Chapter 10] for detailed description of the extraction).

So, why is ensuring termination in Coq so important? The reason for this will be better understood once we introduce the way Coq works with logical statements and propositions. For now, it should be enough to accept the fact that in order to ensure the logical calculus underlying Coq sound, the results of all functions in it (even operating with infinite values, e.g., streams defined co-inductively) should be computable in a finite number of steps. A bit further we will see that the proofs of propositions in Coq are just ordinary values in its computational language, and the construction of the proofs naturally should terminate, hence computation of *any* value in Coq should terminate, since each value can be involved into a proof of some statement.

Postponing the discussion on the nature of propositions and proofs in Coq, we will continue our overview of programming principles in Coq.

With the example of the addition function, we have already seen how the recursive functions are defined. However, using the `Fixpoint` command is not the only way to provide definitions to functions similar to my_plus. When defining the types **unit** or **empty**, we could have noticed the following output produced by the interactive interpreter:

unit `is` *defined*
unit_rect `is` *defined*
unit_ind `is` *defined*
unit_rec `is` *defined*

These three lines indicate that along with the new datatype (**unit** in this case) three additional entities have been generated by the system. These are the companion *induction* and *recursion* principles, which are named using the simple convention basing on the name of the datatype. For example, the **nat** datatype comes accompanied by nat_rect, *nat_ind* and nat_rec, correspondingly.

Continuing playing with natural numbers and leaving the nat_rect and *nat_ind* aside for a moment, we focus on the recursion primitive nat_rec, which is a *higher-order* function with the following type:

`Check` *nat_rec*.

nat_rec : $\forall$ P : **nat** $\to$ Set,
$P\ 0 \to (\forall\ n$: **nat**, $P\ n \to P\ n.{+}1) \to \forall\ n$: **nat**, $P\ n$

[8] Typically, this is done by supplying a user-specific termination argument, which "strictly reduces" at each function call, or defining a function, so it would take a *co-inductive* datatype as its argument.

The type of nat_rec requires a bit of explanation. It is polymorphic in the sense of Haskell and OCaml (i.e., it is parametrized over another type). More precisely, its first parameter, bound by the ∀ quantifier is a function, which maps natural numbers to types (hence the type of this parameter is **nat** → Set). The second parameter is a result of type described by application of the function P to zero. The third parameter is a *family* of functions, indexed by a natural number n. Each function from such a family takes an argument of type $P\ n$ and returns a result of type $P\ n.+1$. The default recursion principle for natural numbers is therefore a higher-order function (i.e., a combinator). If the three discussed arguments are provided, the result of nat_rec will be a function, mapping a natural number n to a value of type $P\ n$.

To see how nat_rec is implemented, let us explore its generalized version, nat_rect:

```
Print nat_rect.
```

```
nat_rect =
 fun (P : nat → Type) (f : P 0) (f0 : ∀ n : nat, P n → P n.+1) ⇒
 fix F (n : nat) : P n :=
   match n as n0 return (P n0) with
   | 0 ⇒ f
   | n0.+1 ⇒ f0 n0 (F n0)
   end
      : ∀ P : nat → Type,
        P 0 → (∀ n : nat, P n → P n.+1) → ∀ n : nat, P n
```

Abstracting away from the details, we can see that nat_rect is indeed a function with three parameters (the keyword fun is similar to the lambda notation and is common in the family of ML-like languages). The body of nat_rect is implemented as a recursive function (defined via the keyword fix) taking an argument n of type **nat**. Internally, it proceeds similarly to our implementation of my_plus: if the argument n is zero, then the "default" value f of type $P\ 0$ is returned. Otherwise, the function proceeds recursively with a smaller argument *n0* by applying the "step" function *f0* to the *n0* and the result of recursive call *F n0*.

Therefore, the summing function can be implemented via the **nat**'s recursion combinator as follows:

```
Definition my_plus'' n m := nat_rec (fun _ ⇒ nat) m (fun n' m' ⇒ m'.+1) n.
Eval compute in my_plus'' 16 12.
```

```
= 28 : (fun _ : nat ⇒ nat) 16
```

The result of invoking my_plus'' is expectable. Notice, however, that when defining it we didn't have to use the keyword Fixpoint (or, equivalently, fix), since all recursion has been "sealed" within the definition of the combinator nat_rect.

2.2.1 Dependent function types and pattern matching

An important thing to notice is the fact that the type of P in the definition of nat_rec is a function that maps *values* of type **nat** into arbitrary types. This gives us a possibility to define *dependently-typed* functions, whose return type depends on their input argument value. A simple example of such a function is below:

```
Check nat_rec.

Definition sum_no_zero n :=
 let: P := (fun n ⇒ if n is 0 then unit else nat) in
 nat_rec P tt (fun n' m ⇒
 match n' return P n' → _ with
 | 0 ⇒ fun _ ⇒ 1
 | n''.+1 ⇒ fun m ⇒ my_plus m (n'.+1)
 end m) n.

Eval compute in sum_no_zero 0.
  = tt : (fun n : nat ⇒ match n with | 0 ⇒ unit | _.+1 ⇒ nat end) 0
Eval compute in sum_no_zero 5.

     = 15
     : (fun n : nat ⇒ match n with
                      | 0 ⇒ unit
                      | _.+1 ⇒ nat
                      end) 5
```

The toy function sum_no_zero maps every natural number n to a sum of numbers 1 ... n, except for 0, which is being mapped into the value tt of type **unit**. We define it via the nat_rec combinator by providing it a function P, which defines the type contract described just above. Importantly, as the first parameter to nat_rec, we pass a type-level function P, which maps 0 to the **unit** type and all other values to the type **nat**. The "step" function, which is a third parameter, of this nat_rec call, makes use of the *dependent* pattern matching, which now explicitly *refines* the return type $P\ n' \to \_$ of the whole `match` e `with` ps `end` expression. This small addition allows the Coq type checker to relate the expected type of my_plus' first argument in the second branch to the type of the pattern matching scrutinee n'. Without the explicit `return` in the pattern matching, in some cases when its result type depends on the value of the scrutinee, the Coq type checking engine will fail to unify the type of the branch and the overall type. In particular, had we omitted the `return` clauses in the pattern matching, we would get the following type-checking error, indicating that Coq cannot infer that the type of my_plus' argument is always **nat**, so it complains:

```
Definition sum_no_zero' n :=
 let: P := (fun n ⇒ if n is 0 then unit else nat) in
 nat_rec P tt (fun n' m ⇒
match n' with
   | 0 ⇒ fun _ ⇒ 1
   | n''.+1 ⇒ fun m ⇒ my_plus m (n'.+1)
end m) n.

Error:
In environment
```

```
n : ?37
P := fun n : nat ⇒ match n with
                   | 0 ⇒ unit
                   | _.+1 ⇒ nat
                   end : nat → Set
n' : nat
m : P n'
The term "m" has type "P n'" while it is expected to have type "nat".
```

In general, dependent pattern matching is a quite powerful tool, which, however, should be used with a great caution, as it makes assisting the Coq type checker a rather non-trivial task. In the vast majority of the cases dependent pattern matching can be avoided. We address the curious reader to the Chapter 8 of the book [7] for more examples on the subject.

Dependent function types, akin to those of nat_rec and our sum_no_zero, which allow the type of the result to vary depending on the value of a function's argument, are a powerful way to *specify the behaviour* of functions, and therefore, are often used to "enforce" the dependently-typed programs to work in a particular expected way. In Coq, dependent function types are omnipresent, and are syntactically specified using the ∀-binder, similarly to the way *parametric* types are specified in Haskell or typed calculi like polymorphic lambda calculus (also known as System F [21, 56]).[9] The crucial difference between Coq's core calculus and System F is that in Coq the types can be parametrised not just by *types* but also by *values*. While the utility of this language "feature" can be already demonstrated for constructing and type-checking *programs* (for example, sum_no_zero), its true strength is best demonstrated when using Coq as a system to construct *proofs*, which is the topic of the subsequent chapters.

2.2.2 Recursion principle and non-inhabited types

Automatically-generated recursion principles for inductively-defined datatypes provide a generic (although not universal) scheme to define recursive functions for the corresponding values. But what if a type is not inhabited, i.e., there are no values in it? We have already seen such a type—it's **empty**, which corresponds to the empty set. As any inductive datatype in Coq, it comes with an automatically generated generalized recursion principle, so let us check its type:

```
Check empty_rect.
```

```
empty_rect
     : ∀ (P : empty → Type) (e : empty), P e
```

Very curiously, the type signature of empty_rect postulates that it is sufficient to provide a function from **empty** to any type (which can very well be just a constant type, e.g., **nat**), and an argument e of type **empty**, so the result of the call to empty_rect will be of type $P\ e$. More concisely, empty_rect allows us to produce a result of *any* type, given that

[9] Although, generally speaking, Coq abuses the ∀-notation using it for what is denoted in other typed calculi by means of quantifiers Λ (terms parametrized by types), ∀ (types parametrized by types) and Π (types parametrized by terms) [52].

we can provide an argument of type **empty**. While it might sound very surprising at the first moment, upon some reflection it seems like a perfectly valid principle, since we will *never* be able to construct the required value of type **empty** in the first place. In more fancy words, such recursion principle can be reformulated as the following postulate:

> Assuming existence of a value, which *cannot be constructed*,
> we will be able to construct *anything*.

This is a very important insight, which will become illuminating when we will be discussing the reasoning with negation in the next chapter.

To conclude this section, we only mention that defining a datatype with no constructors is not the only way to get a type, which is not inhabited. For example, the following type **strange** [3] has a constructor, which, however, can never be invoked, as it requires a value of it type itself in order to return a value:

```
Inductive strange : Set := cs : strange → strange.
```

Therefore, an attempt to create a value of type **strange** by invoking its single constructor will inevitably lead to an infinite, non-terminating, series of constructor calls, and such programs cannon be encoded in Coq. It is interesting to take a look at the recursion principle of **strange**:

```
Check strange_rect.
```

```
strange_rect
     : ∀ P : strange → Type,
       (∀ s : strange, P s → P (cs s)) → ∀ s : strange, P s
```

That is, if we pose the argument P to be a constant type function `fun _ ⇒ empty`, and the second argument to be just an identity function (`fun _ x ⇒ x`) that maps its second argument to itself, we will get a function that, upon receiving argument of type **strange** will construct an argument of type **empty**! More precisely, the existence of a value of type **strange** would allow us to create a value of type **empty** and, therefore a value of *any* type, as was previously demonstrated. The following definition of the function strange_to_empty substantiates this observation:

```
Definition strange_to_empty (s: strange): empty :=
  strange_rect (fun _ ⇒ empty) (fun _ e ⇒ e) s.
```

To summarize, designing a datatype, which is not inhabited, while not trivial, is not impossible, and it is a task of a designer of a particular type to make sure that its values in fact can be constructed.

2.3 More datatypes

While programming with natural numbers is fun, it is time for us to take a brief look at other datatypes familiar from functional programming, as they appear in Coq.

The type of pairs is parametrized by two arbitrary types A and B (by now let us think of its sort `Type` as a generalization of `Set`, which we have seen before). As in Haskell or

OCaml, **prod** can also be seen as a type-level constructor with two parameters that can be possibly curried:

`Check` *prod*.

prod : `Type` → `Type` → `Type`

Pairs in Coq are defined as a higher-order datatype **prod** with just one constructor:
`Print` *prod*.

`Inductive` **prod** (*A B* : `Type`) : `Type` := pair : $A \to B \to A \times B$

For pair: *Arguments A, B are implicit and maximally inserted*
For **prod**: *Argument scopes are* [*type_scope type_scope*]
For pair: *Argument scopes are* [*type_scope type_scope* _ _]

The display above, besides showing how **prod** is defined, specifies that the type arguments of **prod** are *implicit*, in the sense that they will be inferred by the type-checker when enough information is provided, e.g., the arguments of the constructor pair are instantiated with particular values. For instance, type arguments can be omitted in the following expression:

`Check` *pair* 1 *tt*.

(1, tt) : **nat** × **unit**

If one wants to explicitly specify the type arguments of a constructor, the @-prefixed notation can be used:

`Check` @*pair nat unit* 1 *tt*.

(1, tt) : **nat** × **unit**

Notice that the parameters of the datatype come first in the order they are declared, followed by the arguments of the constructor.

The last two lines following the definition of **prod** specify that the notation for pairs is overloaded (in particular, the "_ × _" notation is also used by Coq to denote the multiplication of natural numbers), so it is given a specific *interpretation scope*. That is, when the expression **nat** × **unit** will appear in the type position, it will be interpreted as a type pair **nat unit** rather than like an (erroneous) attempt to "multiply" two types as if they were integers.

Coq comes with a number of functions for manipulating datatypes, such as pair. For instance, the first and second components of a pair:

`Check` *fst*.

fst : ∀ *A B* : `Type`, $A \times B \to A$

`Check` *snd*.

snd : $\forall\ A\ B$: Type, $A \times B \to B$

Curiously, the notation "_ × _" is not hard-coded into Coq, but rather is defined as a lightweight syntactic sugar on top of standard Coq syntax. Very soon we will see how one can easily extend Coq's syntax by defining their own notations. We will also see how is it possible to find what a particular notation means.

The arsenal of a functional programmer in Coq would be incomplete without proper sum and list datatypes:[10]

```
Print sum.
```

```
Inductive sum (A B : Type) : Type := inl : A → A + B | inr : B → A + B
```

```
From mathcomp
Require Import seq.
Print seq.
```

```
Notation seq := list
```

```
Print list.
```

```
Inductive list (A : Type) : Type := nil : list A | cons : A → list A → list A
```

Exercise 2.1 (Fun with lists in Coq)**.** Implement the recursive function alternate of type seq **nat** → seq **nat** → seq **nat**, so it would construct the alternation of two sequences according to the following "test cases".

```
Eval compute in alternate [:: 1;2;3] [:: 4;5;6].
```

```
     = [:: 1; 4; 2; 5; 3; 6]
     : seq nat
```

```
Eval compute in alternate [:: 1] [:: 4;5;6].
```

```
     = [:: 1; 4; 5; 6]
     : seq nat
```

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

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

Hint: The reason why the "obvious" elegant solution might fail is that the argument is not strictly decreasing.

[10] In Ssreflect's enhanced library lists are paraphrased as the seq datatype, which is imported from the module seq.

2.4 Searching for definitions and notations

Of course, we could keep enumerating datatypes and operations on them from the standard Coq/Ssreflect library (which is quite large), but it's always better for a starting Coq hacker to have a way to find necessary definitions on her own. Fortunately, Coq provides a very powerful search tool, whose capabilities are greatly amplified by Ssreflect. Its use is better demonstrated by examples.

```
Search "filt".
```

```
List.filter ∀ A : Type, (A → bool) → list A → list A
List.filter_In
  ∀ (A : Type) (f : A → bool) (x : A) (l : list A),
  List.In x (List.filter f l) ↔ List.In x l ∧ f x = true
```

```
Search "filt" (_ → list _).
```

```
List.filter ∀ A : Type, (A → bool) → list A → list A
```

That is, the first `Search` query just takes a string and looks for definitions of functions and propositions that have it as a part of their name. The second pattern elaborates the first by adding a requirement that the type of the function should include (_ → **list** _) as a part of its return type, which narrows the search scope. As usual the underscores _ denote a wildcard in the pattern and can be used both in the name or type component. Moreover, one can use named patterns of the form ?*id* to bind free identifiers in the sub-types of a sought expression. For instance, the next query will list all functions with map-like types (notice how the higher-order constructor types are abstracted over using wildcards):

```
Search _ ((?X → ?Y) → _ ?X → _ ?Y).
```

```
option_map ∀ A B : Type, (A → B) → option A → option B
List.map ∀ A B : Type, (A → B) → list A → list B
...
```

If necessary, the type patterns in the query can have their types explicitly specified in order to avoid ambiguities due to notation overloading. For instance, the following search will return all functions and propositions that make use of the _ × _ notation and operate with natural numbers:

```
Search _ (_ × _ : nat).
```

In contrast, the next query will only list the functions/propositions, where _ × _ is treated as a notation for the pair datatype (including fst and snd, which we have already seen):

```
Search _ (_ × _: Type).
```

A detailed explanation of the syntax of `Search` tool as well as additional examples can be found in Chapter 10 of Ssreflect documentation [23].

When working with someone's Coq development, sometimes it might be not entirely obvious what particular notation means: Coq's extensible parser is very simple to abuse

by defining completely intractable abbreviations, which might say a lot to the library developer, but not to its client. Coq provides the `Locate` command to help in demystifying notations as well as locating the position of particular definitions. For example, the following query will show all the definitions of the notation "_ + _" as well as the scopes they defined in.

```
Locate "_+ _".
```

```
Notation Scope
"x + y" := sum x y : type_scope

"m + n" := addn m n : nat_scope
```

We can see now that the plus-notation is used in particular for the addition of natural numbers (in *nat_scope*) and the declaration of a sum type (in *type_scope*). Similarly to the notations, the `Locate` command can help finding the definition in the source modules they defined:[11]

```
Locate map.
```

```
Constant Coq.Lists.List.map
  (shorter name to refer to it in current context is List.map)
Constant Ssreflect.ssrfun.Option.map
  (shorter name to refer to it in current context is ssrfun.Option.map)
...
```

2.5 An alternative syntax to define inductive datatypes

In the previous sections of this chapter we have already seen the way inductive datatypes are defined in the setting "traditional" Coq. These are the definitions that will be displayed when using the `Print` utility. However, in the rest of the development in this book, we will be using a version of Coq, enhanced with the Ssreflect tool, which, in particular, provides more concise notation for defining constructors. For instance, as an alternative to the standard definition of the product datatype, we can define our own product in the following way:

```
Inductive my_prod (A B : Type) : Type := my_pair of A & B.
```

Notice that A and B are type parameters of the whole datatype as well as of its single constructor my_pair, which *additionally* required two value arguments, *whose* types are A and B, respectively.

Next, let us try to create a value of type **my_prod nat unit** and check its type.

```
Check my_pair 1 tt.
```

```
Error: The term "1" has type "nat" while it is expected to have type "Type".
```

[11] The module system of Coq is similar to OCaml and will be discussed further in this chapter.

The error message is caused by the fact that the constructor has expected the type parameters to be provided *explicitly* first, so the value above should in fact have been created by calling my_pair **nat unit** 1 tt. Since mentioning types every time is tedious, we can now take advantage of Coq's elaboration algorithm, which is capable to infer them from the values of actual arguments (e.g., 1 and tt), and declare my_pair's type arguments as implicit:

Arguments my_pair [*A B*].

We have already witnessed standard Coq's datatypes making use of specific user-defined notations. Let us define such notation for the type **my_prod** and its my_pair constructor.

```
Notation "X ** Y" := (my_prod X Y) (at level 2).
Notation "( X „ Y )" := (my_pair X Y).
```

The `level` part in the first notation definition is mandatory for potentially left-recursive notations, which is the case here, in order to set up parsing priorities with respect to other notations.

With these freshly defined notations we are now free to write the following expressions:

```
Check (1 „ 3).
```

```
(1„ 3)
     : nat ** nat
```

```
Check nat ** unit ** nat.
```

```
(nat ** unit) ** nat
     : Set
```

Notice that the notation "_ ** _" for my_pair by default is set to be left-associative. The other associativity should be declared explicitly, and we address the reader to the Chapter 12 of Coq manual [10] for the details of the `Notation` command syntax.

2.6 Sections and modules

We conclude this chapter by a very brief overview of Coq's module system.

Sections are the simplest way to structure the programs in Coq. In particular, sections allow the programmer to limit the scope of modules imported to the current file (each compiled `.v` file in the scope of the interpreter is considered as a module), as well as to defined *locally-scoped* variables. To see how it works, let us construct a section containing a utility function for natural numbers. Declaring a section starts from the keyword `Section`, followed by the name of the section:

```
Section NatUtilSection.
```

We now define a *variable* n of type n, whose scope is lexically limited by the section NatUtilSection (including its internal sections). One can think of variables declared this way as of unspecified values, which we assume to be available outside of the section.

```
Variable n: nat.
```

We can now define a function, implementing multiplication of natural numbers by means of addition. To do this, we assume the variable n to be fixed, so the multiplication can be formulated just as a function of *one* parameter:

```
Fixpoint my_mult m := match (n, m) with
 | (0, _) ⇒ 0
 | (_, 0) ⇒ 0
 | (_, m'.+1) ⇒ my_plus (my_mult m') n
 end.
```

We now close the section by using the `End` keyword.

```
End NatUtilSection.
```

Unlike Haskell or Java's modules, sections in Coq are transparent: their internal definitions are visible outside of their bodies, and the definitions' names need not be qualified. The same *does not* apply to sections' variables. Instead, they become *parameters* of definitions they happened to be used in. This can be seen by printing the implementation of my_mult outside of the section NatUtilSection.

```
Print my_mult.
```

```
my_mult =
fun n : nat ⇒
fix my_mult (m : nat) : nat :=
  let (n0, y) := (n, m) in
  match n0 with
  | 0 ⇒ 0
  | _.+1 ⇒ match y with
              | 0 ⇒ 0
              | m'.+1 ⇒ my_plus (my_mult m') n
              end
  end
     : nat → nat → nat
```

We can see now that the variable n became an actual parameter of my_mult, so the function now takes *two* parameters, just as expected.

An alternative to sections in Coq, which provides better encapsulation, are *modules*. A module, similarly to a section, can contain locally-declared variables, sections and modules (but not modules within sections!). However, the internals of a module are not implicitly exposed to the outside, instead they should be either referred to by *qualified* names or exported explicitly by means of putting them into a submodule and via the command `Export`, just as demonstrated below:

```
Module NatUtilModule.

Fixpoint my_fact n :=
  if n is n'.+1 then my_mult n (my_fact n') else 1.

Module Exports.
Definition fact := my_fact.
End Exports.
```

```
End NatUtilModule.
```

The submodule EXPORTS creates a synonym fact for the function my_fact, defined outside of it. The following command explicitly exports all internals of the module NATUTILMODULE.EXPORTS, therefore making fact visible outside of NATUTILMODULE.

```
Export NatUtilModule.Exports.
```

```
Check my_fact.
```

Error: *The reference* my_fact *was not found* `in` *the current environment.*

```
Check fact.
```

```
fact
     : nat → nat
```

3 Propositional Logic

In the previous chapter we had an opportunity to explore Coq as a functional programming language and learn how to define inductive datatypes and programs that operate with them, implementing the latter ones directly or using the automatically-generated recursion combinators. Importantly, most of the values that we met until this moment, inhabited the types, which were defined as elements of the sort `Set`. The types *unit*, *empty*, *nat*, *nat* × *unit* etc. are good examples of *first-order* types inhabiting the sort `Set` and, therefore, contributing to the analogy between sets and first-order types, which we explored previously. In this chapter, we will be working with a new kind of entities, incorporated by Coq: *propositions*.

3.1 Propositions and the `Prop` sort

In Coq, propositions bear a lot of similarities with types, demonstrated in Chapter 2, and inhabit a separate sort `Prop`, similarly to how first-order types inhabit `Set`.[1] The "values" that have elements of `Prop` as their types are usually referred to as *proofs* or *proof terms*, the naming convention which stems out of the idea of *Curry-Howard Correspondence* [14, 30].[2] Sometimes, the Curry-Howard Correspondence is paraphrased as *proofs-as-programs*, which is truly illuminating when it comes to the intuition behind the formal proof construction in Coq, which, in fact, is just programming in disguise.

The *Calculus of Inductive Constructions* (CIC) [3, 13] a logical foundation of Coq, similarly to its close relative, Martin-Löf's *Intuitionistic Type Theory* [38], considers proofs to be just regular values of the "programming" language it defines. Therefore, the process of constructing proofs in Coq is very similar to the process of writing programs. Intuitively, when one asks a question "Whether the proposition P is *true*?", what is meant in fact is "Whether the *proof* of P can be constructed?". This is an unusual twist, which is crucial for understanding the concept of the "truth" and proving propositions in CIC (and, equivalently, in Coq), so we specifically outline it here in the form of a motto:

> Only those propositions are considered to be *true*, which are provable *constructively*, i.e., by providing an *explicit* proof term, that inhabits them.

This formulation of "truth" is somewhat surprising at the first encounter, comparing to classical propositional logic, where the propositions are considered to be true simply if they are tautologies (i.e., reduce to the boolean value *true* for all possible combinations of their free variables' values), therefore leading to the common proof method in classical propositional logic: truth tables. While the truth table methodology immediately delivers the recipe to prove propositions without quantifiers *automatically* (that is, just

[1] In the Coq community, the datatypes of `Prop` sort are usually referred to as *inductive predicates*.
[2] http://en.wikipedia.org/wiki/Curry-Howard_correspondence

by checking the corresponding truth tables), it does not quite scale when it comes to the higher-order propositions (i.e., quantifying over predicates) as well as of propositions quantifying over elements of arbitrary domains. For instance, the following proposition, in which the reader can recognize the induction principle over natural numbers, cannot be formulated in the zeroth- or first-order propositional logic (and, in fact, in *any* propositional logic):

For any predicate P, if $P(0)$ holds, and for any m, $P(m)$ implies $P(m+1)$,
then for any n, $P(n)$ holds.

The statement above is *second-order* as it binds a first-order predicate by means of universal quantification, which makes it belong to the corresponding second-order logic (which is not even propositional, as it quantifies over arbitrary natural values, not just propositions). Higher-order logics [9] are known to be undecidable in general, and, therefore, there is no automatic way to reduce an arbitrary second-order formula to one of the two values: *true* or *false*.

CIC as a logic is expressive enough to accommodate propositions with quantifications of an arbitrary order and over arbitrary values. On one hand, it makes it an extremely powerful tool to state almost any proposition of interest in modern mathematics or computer science. On the other hand, proving such statements (i.e., constructing their proof terms), will require human assistance, in the same way the "paper-and-pencil" proofs are constructed in classical mathematics. However, unlike the paper-and-pencil proofs, proofs constructed in Coq are a subject of immediate *automated* check, since they are just programs to be verified for well-typedness. Therefore, the process of proof construction in Coq is *interactive* and assumes the constant interoperation between a human prover, who constructs a proof term for a proposition (i.e., writes a program), and Coq, the proof assistant, which carries out the task of *verifying* the proof (i.e., type-checking the program). This largely defines our agenda for the rest of this course: we are going to see how to *prove* logical statements by means of writing *programs*, that have the types corresponding to these statements.

In the rest of this chapter we will focus only on the capability of Coq as a formal system allowing one to reason about propositions, leaving reasoning about values aside till the next chapter. It is worth noticing that a fragment of Coq, which deals with the sort `Prop`, accommodating all the propositions, and allows the programmer to make statements with propositions, corresponds to the logical calculus, known as System F_ω (see Chapter 30 of [52]) extending System F [21, 56], mentioned in Chapter 2. Unlike System F, which introduces polymorphic types, and, equivalently, first-order propositions that quantify over other propositions, System F_ω allows one to quantify as well over *type operators*, which can be also thought of as higher-order propositions.

3.2 The truth and the falsehood in Coq

We start our acquaintance with propositional logic in Coq by demonstrating how the two simplest propositions, the truth and the falsehood, are encoded. Once again, let us remember that, unlike in propositional logic, in Coq these two are *not* the only possible propositional *values*, and soon we will see how a wide range of propositions different from

mere truth or falsehood are implemented. From now on, we will be always including to the development the standard Ssreflect's module ssreflect, which imports some necessary machinery for dealing with propositions and proofs.

```
From mathcomp Require Import ssreflect.
```

The truth is represented in Coq as a datatype of sort `Prop` with just one constructor, taking no arguments:

```
Print True.
```

```
Inductive True : Prop := I : True
```

Such simplicity makes it trivial to construct an instance of the **True** proposition:[3] Now we can prove the following proposition in Coq's embedded propositional logic, essentially meaning that **True** is provable.

```
Theorem true_is_true: True.
```

```
1 subgoals, subgoal 1 (ID 1)

  ============================
  True
```

The command `Theorem` serves two purposes. First, similarly to the command `Definition`, it defines a named entity, which is not necessarily a proposition. In this case the name is true_is_true. Next, similarly to `Definition`, there might follow a list of parameters, which is empty in this example. Finally, after the colon : there is a type of the defined value, which in this case it **True**. With this respect there is no difference between `Theorem` and `Definition`. However, unlike `Definition`, `Theorem` doesn't require one to provide the expression of the corresponding type right away. Instead, the *interactive proof mode* is activated, so the proof term could be constructed incrementally. The process of the gradual proof construction is what makes Coq to be a *interactive proof assistant*, in addition to being already a programming language with dependent types.

Although not necessary, it is considered a good programming practice in Coq to start any interactive proof with the Coq's command `Proof`, which makes the final scripts easier to read and improves the general proof layout.

```
Proof.
```

In the interactive proof mode, the *goals* display shows a *goal* of the proof—the type of the value to be constructed (**True** in this case), which is located below the double line. Above the line one can usually see the context of *assumptions*, which can be used in the process of constructing the proof. Currently, the assumption context is empty, as the theorem we stated does not make any and ventures to prove **True** out of thin air. Fortunately, this is quite easy to do, as from the formulation of the **True** type we already know that it is inhabited by its only constructor I. The next line proved the *exact* value of the type of the goal.

[3] In the context of propositional logic, we will be using the words "type" and "proposition" interchangeably without additional specification when it's clear from the context.

`exact`: *I*.

This completes the proof, as indicated by the Proof General's `*response*` display:

No more `subgoals`.
(`dependent` *evars*:)

The only thing left to complete the proof is to inform Coq that now the theorem `true_is_true` is proved, which is achieved by typing the command `Qed`.

`Qed`.

In fact, typing `Qed` invokes a series of additional checks, which ensure the well-formedness of the constructed proof term. Although the proof of `true_is_true` is obviously valid, in general, there is a number of proof term properties to be checked *a posteriori* and particularly essential in the case of proofs about infinite objects, which we do not cover in these course (see Chapter 13 of [3] for a detailed discussion on such proofs).

So, our first theorem is proved. As it was hinted previously, it could have been stated even more concisely, formulated as a mere definition, and proved by means of providing a corresponding value, without the need to enter the proof mode:

`Definition` *true_is_true'*: *True* := *I*.

Although this is a valid way to prove statements in Coq, it is not as convenient as the interactive proof mode, when it comes to construction of large proofs, arising from complicated statements. This is why, when it comes to proving propositions, we will prefer the interactive proof mode to the "vanilla" program definition. It is worth noticing, though, that even though the process of proof construction in Coq usually looks more like writing a *script*, consisting from a number of commands (which are called *tactics* in Coq jargon), the result of such script, given that it eliminates all of the goals, is a valid well-typed Coq program. In comparison, in some other dependently-typed frameworks (e.g., in Agda), the construction of proof terms does not obscure the fact that what is being constructed is a program, so the resulting interactive proof process is formulated as "filling the holes" in a program (i.e., a proof-term), which is being gradually refined. We step away from the discussion on which of these two views to the proof term construction is more appropriate.

There is one more important difference between values defined as `Definition`s and `Theorem`s. While both define what in fact is a proof terms for the declared type, the value bound by `Definition` is *transparent*: it can be executed by means of unfolding and subsequent evaluation of its body. In contrast, a proof term bound by means of `Theorem` is *opaque*, which means that its body cannot be evaluated and serves only one purpose: establish the fact that the corresponding type (the theorem's statement) is inhabited, and, therefore is true. This distinction between definitions and theorems arises from the notion of *proof irrelevance*, which, informally, states that (ideally) one shouldn't be able to distinguish between two proofs of the same statement as long as they both are valid.[4] Conversely, the programs (that is, what is created using the `Definition` command) are typically of interest by themselves, not only because of the type they return.

The difference between the two definitions of the truth's validity, which we have just constructed, can be demonstrated by means of the `Eval` command.

[4]Although, in fact, proof terms in Coq can be very well distinguished.

`Eval compute in` *true_is_true.*

= true_is_true : **True**

`Eval compute in` *true_is_true'.*

= I : **True**

As we can see now, the theorem is evaluated to itself, whereas the definition evaluates to it body, i.e., the value of the constructor I.

A more practical analogy for the above distinction can be drawn if one will think of `Definition`s as of mere functions, packaged into libraries and intended to be used by third-party clients. In the same spirit, one can think of `Theorem`s as of facts that need to be checked only once when established, so no one would bother to re-prove them again, knowing that they are valid, and just appeal to their types (statement) without exploring the proof.[5] This is similar to what is happening during the oral examinations on mathematical disciplines: a student is supposed to remember the statements of theorems from the *previous* courses and semesters, but is not expected to reproduce their proofs.

At this point, an attentive reader can notice that the definition of **True** in Coq is strikingly similar to the definition of the type *unit* from Chapter 2. This is a fair observation, which brings us again to the Curry-Howard analogy, and makes it possible to claim that the trivial truth proposition is isomorphic to the *unit* type from functional programming. Indeed, both have just one way to be constructed and can be constructed in any context, as their single constructor does not require any arguments.

Thinking by analogy, one can now guess how the falsehood can be encoded.

`Print` *False.*

`Inductive` **False** : `Prop` :=

Unsurprisingly, the proposition **False** in Coq is just a Curry-Howard counterpart of the type *empty*, which we have constructed in Chapter 2. Moreover, the same intuition that was applicable to *empty*'s recursion principle ("anything can be produced given an element of an empty set"), is applicable to reasoning by induction with the **False** proposition:

`Check` *False_ind.*

False_ind

: ∀ *P* : `Prop`, **False** → *P*

That is, *any* proposition can be derived from the falsehood by means of implication.[6] For instance, we can prove now that **False** implies the equality $1 = 2$.[7]

[5] While we consider this to be a valid analogy to the mathematical community functions, it is only true in spirit. In the real life, the statements proved once, are usually re-proved by students for didactical reasons, in order to understand the proof principles and be able to produce other proofs. Furthermore, the history of mathematics witnessed a number of proofs that have been later invalidated. Luckily, the mechanically-checked proofs are usually not a subject of this problem.

[6] In light of the Curry-Howard analogy, at this moment it shouldn't come as a surprise that Coq uses the arrow notation `->` both for function types and for propositional implication: after all, they both are just particular cases of functional abstraction, in sorts `Set` or `Prop`, correspondingly.

[7] We postpone the definition of the equality till the next chapter, and for now let us consider it to be just an arbitrary proposition.

`Theorem` *one_eq_two*: *False* → 1 = 2.
`Proof.`

One way to prove this statement is to use the **False** induction principle, i.e., the theorem False_ind, directly by instantiating it with the right predicate *P*:

`exact:` (*False_ind* (1 = 2)).

This indeed proves the theorem, but for now, let us explore a couple of other ways to prove the same statement. For this we first `Undo` the last command of the already succeeded but not yet completed proof.

`Undo.`

Instead of supplying the argument (1 = 2) to False_ind manually, we can leave it to Coq to figure out, what it should be, by using the Ssreflect `apply:` tactic.

`apply:` *False_ind*.

The following thing just happened: the tactic `apply:` supplied with an argument False_ind, tried to figure out whether our goal **False** → (1 = 2) matches any *head* type of the theorem False_ind. By *head type* we mean a component of type (in this case, ∀ *P* : `Prop`, **False** → *P*), which is a type by itself and possibly contains free variables. For instance, recalling that → is right-associative, head-types of False_ind would be *P*, **False** → *P* and ∀ *P* : `Prop`, **False** → *P* itself.

So, in our example, the call to the tactics `apply:` False_ind makes Coq realize that the goal we are trying to prove matches the type **False** → *P*, where *P* is taken to be (1 = 2). Since in this case there is no restrictions on what *P* can be (as it is universally-quantified in the type of False_ind), Coq assigns *P* to be (1 = 2), which, after such specialization, turns the type of False_ind to be exactly the goal we're after, and the proof is done.

There are many more ways to prove this rather trivial statement, but at this moment we will demonstrate just yet another one, which does not appeal to the False_ind induction principle, but instead proceeds by *case analysis*.

`Undo.`

`case.`

The tactic `case` makes Coq to perform the case analysis. In particular, it *deconstructs* the *top assumption* of the goal. The top assumption in the goal is such that it comes first before any arrows, and in this case it is a value of type **False**. Then, for all constructors of the type, whose value is being case-analysed, the tactic `case` constructs *subgoals* to be proved. Informally, in mathematical reasoning, the invocation of the `case` tactic would correspond to the statement "let us consider all possible cases, which amount to the construction of the top assumption". Naturally, since **False** has *no* constructors (as it corresponds to the *empty* type), the case analysis on it produces *zero* subgoals, which completes the proof immediately. Since the result of the proof is just some program, again, we can demonstrate the effect of `case` tactic by proving the same theorem with an exact proof term:

`Undo.`

`exact:` (`fun` (*f*: *False*) ⇒ `match` *f* `with end`).

As we can see, one valid proof term of one_eq_two is just a function, which case-analyses on the value of type **False**, and such case-analysis has no branches.

```
Qed.
```

3.3 Implication and universal quantification

By this moment we have already seen how implication is represented in Coq: it is just a functional type, represented by the "arrow" notation $\rightarrow$ and familiar to all functional programmers. Indeed, if a function of type $A \rightarrow B$ is a program that takes an argument value of type A and returns a result value of type B, then the propositional implication $P \rightarrow Q$ is, ... a program that takes an argument proof term of type P and returns a proof of the proposition Q.

Unlike most of the value-level functions we have seen so far, propositions are usually parametrized by other propositions, which makes them instances of *polymorphic* types, as they appear in System F and System F_ω. Similarly to these systems, in Coq the universal quantifier $\forall$ (spelled `forall`) binds a variable immediately following it in the scope of the subsequent type.[8] For instance, the transitivity of implication in Coq can be expressed via the following proposition:

$$\forall\ P\ Q\ R\text{: } \texttt{Prop},\ (P \rightarrow Q) \rightarrow (Q \rightarrow R) \rightarrow P \rightarrow R$$

The proposition is therefore *parametrized* over three propositional variables, P, Q and R, and states that from a proof term of type $P \rightarrow Q$ and a proof term of type $Q \rightarrow R$ one can build a proof term of type $P \rightarrow R$.[9] Let us now prove this statement in the form of a theorem.

```
Theorem imp_trans: ∀ P Q R: Prop, (P → Q) → (Q → R) → P → R.
Proof.
```

Our goal is the statement of the theorem, its type. The first thing we are going to do is to "peel off" some of the goal assumptions—the $\forall$-bound variables—and move them from the goal to the assumption context (i.e., from below to above the double line). This step in the proof script is usually referred to as *bookkeeping*, since it does not directly contribute to reducing the goal, but instead moves some of the values from the goal to assumption, as a preparatory step for the future reasoning.

Ssreflect offers a tactic and a small but powerful toolkit of *tacticals* (i.e., higher-order tactics) for bookkeeping. In particular, for moving the bound variables from "bottom to the top", one should use a combination of the "no-op" tactic `move` and the tactical $\Rightarrow$ (spelled `=>`). The following command moves the next three assumptions from the goal, P, Q and R to the assumption context, simultaneously renaming them to A, B and C. The renaming is optional, so we just show it here to demonstrate the possibility to give arbitrary (and, preferably, more meaningful) names to the assumption variables "on the fly" while constructing the proof via a script.

```
move⇒ A B C.
```

[8] As it has been noticed in Chapter 2 the $\forall$-quantifier is Coq's syntactic notation for dependent function types, sometimes also referred to a Π-*types* or *dependent product types*.

[9] Recall that the arrows have right associativity, just like function types in Haskell and OCaml, which allows one to apply functions partially, specifying their arguments one by one

A : `Prop`
B : `Prop`
C : `Prop`
==========
$(A \to B) \to (B \to C) \to A \to C$

We can now move the three other arguments to the top using the same command: the `move⇒` combination works uniformly for ∀-bound variables as well as for the propositions on the left of the arrow.

`move⇒` *H1 H2 a*.

H1 : $A \to B$
H2 : $B \to C$
a : A
==========
C

Again, there are multiple ways to proceed now. For example, we can recall the functional programming and get the result of type C just by two subsequent applications of *H1* and *H2* to the value *a* of type A:

`exact:` (*H2* (*H1 a*)).

Alternatively, we can replace the direct application of the hypotheses *H1* and *H2* by the reversed sequence of calls to the `apply:` tactics.

`Undo`.

The first use of `apply:` will replace the goal C by the goal B, since this is what it takes to get C by using *H2*:

`apply:` *H2*.

H1 : $A \to B$
a : A
==========
B

The second use of `apply:` reduces the proof of B to the proof of A, demanding an appropriate argument for *H1*.

`apply:` *H1*.

a : A
==========
A

Notice that both calls to `apply:` removed the appropriate hypotheses, *H1* and *H2* from the assumption context. If one needs a hypothesis to stay in the context (to use it twice, for example), then the occurrence of the tactic argument hypothesis should be parenthesised: `apply:` (*H1*).

Finally, we can see that the only goal left to prove is to provide a proof term of type A. Luckily, this is exactly what we have in the assumption by the name a, so the following demonstration of the exact a finishes the proof:

```
exact: a.
Qed.
```

In the future, we will replace the use of trivial tactics, such as `exact:` by Ssreflect's much more powerful tactics `done`, which combines a number of standard Coq's tactics in an attempt to finish the proof of the current goal and reports an error if it fails to do so.

Exercise 3.1 (∀-distributivity)**.** Formulate and prove the following theorem in Coq, which states the distributivity of universal quantification with respect to implication:

$$\forall P\ Q, [(\forall x, P(x) \implies Q(x)) \implies ((\forall y, P(y)) \implies \forall z, Q(z))]$$

Hint: Be careful with the scoping of universally-quantified variables and use parentheses to resolve ambiguities!

3.3.1 On forward and backward reasoning

Let us check now the actual value of the proof term of theorem imp_trans.

```
Print imp_trans.
```

```
imp_trans =
  fun (A B C : Prop) (H1 : A → B) (H2 : B → C) (a : A) ⇒
  (fun _evar_0_ : B ⇒ H2 _evar_0_) ((fun _evar_0_ : A ⇒ H1 _evar_0_) a)
       : ∀ P Q R : Prop, (P → Q) → (Q → R) → P → R

Argument scopes are [type_scope type_scope type_scope _ _ _]
```

Even though the proof term looks somewhat hairy, this is almost exactly our initial proof term from the first proof attempt: *H2 (H1 a)*. The only difference is that the hypotheses *H1* and *H2* are *eta-expanded*, that is instead of simply *H1* the proof terms features its operational equivalent `fun` *b*: $B \Rightarrow$ *H2 b*. Otherwise, the printed program term indicates that the proof obtained by means of direct application of *H1* and *H2* is the same (modulo eta-expansion) as the proof obtained by means of using the `apply:` tactic.

These two styles of proving: by providing a direct proof to the goal or some part of it, and by first reducing the goal via tactics, are usually referred in the mechanized proof community as *forward* and *backward* proof styles.

- The *backward* proof style assumes that the goal is being gradually transformed by means of applying some tactics, until its proof becomes trivial and can be completed by means of basic tactics, like `exact:` or `done`.
- The *forward* proof style assumes that the human prover has some "foresight" with respect to the goal she is going to prove, so she can define some "helper" entities as

well as to adapt the available assumptions, which will then be used to solve the goal. Typical example of the forward proofs are the proofs from the classical mathematic textbooks: first a number of "supporting" lemmas is formulated, proving some partial results, and finally all these lemmas are applied in concert in order to prove an important theorem.

While the standard Coq is very well supplied with a large number of tactics that support reasoning in the backward style, it is less convenient for the forward-style reasoning. This aspect of the tool is significantly enhanced by Ssreflect, which introduces a small number of helping tactics, drastically simplifying the forward proofs, as we will see in the subsequent chapters.

3.3.2 Refining and bookkeeping assumptions

Suppose, we have the following theorem to prove, which is just a simple reformulation of the previously proved imp_trans:

```
Theorem imp_trans' (P Q R: Prop) : (Q → R) → (P → Q) → P → R.
Proof.
move⇒ H1 H2.
```

Notice that we made the propositional variables P, Q and R to be parameters of the theorem, rather than $\forall$-quantified values. This relieved us from the necessity to lift them using `move⇒` in the beginning of the proof.

In is natural to expect that the original imp_trans will be of some use. We are now in the position to apply it directly, as the current goal matches its conclusion. However, let us do something slightly different: *move* the statement of imp_trans into the goal, simultaneously with specifying it (or, equivalently, partially applying) to the assumptions *H1* and *H2*. Such move "to the bottom part" in Ssreflect is implemented by means of the : tactical, following the `move` command:

```
move: (imp_trans P Q R)=> H.
```

```
H1 : Q → R
H2 : P → Q
H : (P → Q) → (Q → R) → P → R
============================
 P → R
```

What has happened now is a good example of the forward reasoning: the specialized version of (imp_trans P Q R), namely, $(P \to Q) \to (Q \to R) \to P \to R$, has been moved to the goal, so it became $((P \to Q) \to (Q \to R) \to P \to R) \to P \to R$. Immediately after that, the top assumption (that is, what has been just "pushed" to the goal stack) was moved to the top and given the name *H*. Now we have the assumption *H* that can be applied in order to reduce the goal.

```
apply: H.
```

```
H1 : Q → R
```

```
  H2 : P → Q
  ============================
   P → Q
```

```
subgoal 2 (ID 142) is:
 Q → R
```

The proof forked into two goals, since H had two arguments, which we can now fulfill separately, as they trivially are our assumptions.

```
done.
done.
```

The proof is complete, although the last step is somewhat repetitive, since we know that for two generated sub-goals the proofs are the same. In fact, applications of tactics can be *chained* using the ; connective, so the following complete proof of imp_trans' runs `done` for *all* subgoals generated by `apply`: H:

```
Restart.
move: (imp_trans P Q R)=> H H1 H2.
apply: H; done.
```

Also, notice that the sequence in which the hypotheses were moved to the top has changed: in order to make the proof more concise, we first created the "massaged" version of imp_trans, and then moved it as H to the top, following by *H1* and *H2*, which were in the goal from the very beginning.

To conclude this section, let us demonstrate even shorter way to prove this theorem once again.

```
Restart.
move⇒H1 H2; apply: (imp_trans P Q R)=>//.
Qed.
```

After traditional move of the two hypotheses to the top, we applied the specialised version of imp_trans, where its three first arguments were explicitly instantiated with the local P, Q and R. This application generated two subgoals, each of which has been then automatically solved by the trailing tactical ⇒ //, which is equivalent to ;`try done` and, informally speaking, "tries to kill all the newly created goals".[10]

3.4 Conjunction and disjunction

Two main logical connectives, conjunction and disjunction, are implemented in Coq as simple inductive predicates in the sort `Prop`. In order to avoid some clutter, from this moment and till the end of the chapter let us start a new module *Connectives* and assume a number of propositional variables in it (as we remember, those will be abstracted over outside of the module in the statements they happen to occur).

```
Module Connectives.
```

[10]The Coq's `try` tactical tries to execute its tactic argument in a "soft way", that is, not reporting an error if the argument fails.

```
Variables P Q R: Prop.
```

The propositional conjunction of P and Q, denoted by $P \wedge Q$, is a straightforward Curry-Howard counterpart of the *pair* datatype that we have already seen in Chapter 2, and is defined by means of the predicate **and**.

```
Locate "_/\ _".
```

"$A \wedge B$" := **and** A B : *type_scope*

```
Print and.
```

```
Inductive and (A B : Prop) : Prop := conj : A → B → A ∧ B

For conj: Arguments A, B are implicit
For and: Argument scopes are [type_scope type_scope]
For conj: Argument scopes are [type_scope type_scope _ _]
```

Proving a conjunction of P and Q therefore amounts to constructing a pair by invoking the constructor conj and providing values of P and Q as its arguments:[11]

```
Goal P → R → P ∧ R.
move⇒ p r.
```

The proof can be completed in several ways. The most familiar one is to apply the constructor conj directly. It will create two subgoals, P and Q (which are the constructor arguments), that can be immediately discharged.

```
apply: conj=>//.
```

Alternatively, since we now know that **and** has just one constructor, we can use the generic Coq's `constructor` n tactic, where n is an (optional) number of a constructor to be applied (and in this case it's 1)

```
Undo.
constructor 1=>//.
```

Finally, for propositions that have exactly one constructor, Coq provides a specialized tactic `split`, which is a synonym for `constructor` 1:

```
Undo. split=>//.
Qed.
```

In order to prove something out of a conjunction, one needs to *destruct* it to get its constructor's arguments, and the simplest way to do so is by the `case`-analysis on a single constructor.

```
Goal P ∧ Q → Q.
case.
```

Again, the tactic `case` replaced the top assumption $P \wedge Q$ of the goal with the arguments of its only constructor, P and Q making the rest of the proof trivial.

```
done.
Qed.
```

[11] The command `Goal` creates an anonymous theorem and initiates the interactive proof mode.

The datatype of disjunction of P and Q, denoted by $P \lor Q$, is isomorphic to the *sum* datatype from Chapter 2 and can be constructed by using one of its two constructors: *or_introl* or *or_intror*.

```
Locate "_\/ _".
```

"$A \lor B$" := **or** A B : *type_scope*

```
Print or.
```

```
Inductive or (A B : Prop) : Prop :=
    or_introl : A → A ∨ B | or_intror : B → A ∨ B
```

For or_introl, when applied to less than 1 *argument*:
Arguments A, B are implicit
...

In order to prove disjunction of P and Q, it is sufficient to provide a proof of just P or Q, therefore appealing to the appropriate constructor.

```
Goal Q → P ∨ Q ∨ R.
move⇒ q.
```

Similarly to the case of conjunction, this proof can be completed either by applying a constructor directly, by using `constructor` 2 tactic or by a specialised Coq's tactic for disjunction: `left` or `right`. The notation "_ ∨ _" is right-associative, hence the following proof script, which first reduces the goal to the proof of $Q \lor R$, and then to the proof of Q, which is trivial by assumption.

```
by right; left.
Qed.
```

The use of Ssreflect's tactical `by` makes sure that its argument tactic (`right`; `left` in this case) succeeds and the proof of the goal completes, similarly to the trailing `done`. If the sequence of tactics `left`; `right` wouldn't prove the goal, a proof script error would be reported.

The statements that have a disjunction as their assumption are usually proved by case analysis on the two possible disjunction's constructors:

```
Goal P ∨ Q → Q ∨ P.
case⇒x.
```

Notice how the case analysis via the Ssreflect's `case` tactic was combined here with the trailing ⇒. It resulted in moving the constructor parameter in *each* of the subgoals from the goal assumptions to the assumption context. The types of x are different in the two branches of the proof, though. In the first branch, x has type P, as it names the argument of the *or_introl* constructor.

```
  P : Prop
  Q : Prop
  R : Prop
  x : P
```

```
  ============================
   Q ∨ P

subgoal 2 (ID 248) is:
 Q ∨ P
```

```
by right.
```

```
  P : Prop
  Q : Prop
  R : Prop
  x : Q
  ============================
   Q ∨ P
```

In the second branch the type of x is Q, as it accounts for the case of the *or_intror* constructor.

```
by left.
Qed.
```

It is worth noticing that the definition of disjunction in Coq is *constructive*, whereas the disjunction in classical propositional logic is not. More precisely, in classical logic the proof of the proposition $P \vee \sim P$ is true by the axiom of excluded middle (see Section 3.7 for a more detailed discussion), whereas in Coq, proving $P \vee \sim P$ would amount to *constructing* the proof of either P or $\sim P$. Let us illustrate it with a specific example. If P is a proposition stating that $P = NP$, then in classical logic tautology $P \vee \sim P$ holds, although it does not contribute to the proof of either of the disjuncts. In constructive logic, which Coq is an implementation of, in the trivial assumptions given the proof of $P \vee \sim P$, we would be able to extract the proof of either P or $\sim P$.[12]

3.5 Proofs with negation

In Coq's constructive approach proving the negation of $\sim P$ of a proposition P literally means that one can derive the falsehood from P.

```
Locate "~ _".
```

```
  "~ x" := not x : type_scope
```

```
Print not.
```

```
  not = fun A : Prop ⇒ A → False
      : Prop → Prop
```

Therefore, the negation not on propositions from Prop is just a function, which maps a proposition A to the implication $A \rightarrow$ **False**. With this respect the intuition of negation from classical logic might be misleading: as it will be discussed in Section 3.7, the Calculus

[12] Therefore, winning us the Clay Institute's award.

of Constructions lacks the double negation elimination axiom, which means that the proof of $\sim\sim A$ will not deliver the proof of A, as such derivation would be non-constructive, as one cannot get a value of type A out of a function of type $(A \to B) \to B$, where B is taken to be **False**.

However, reasoning out of negation helps to derive the familiar proofs by contradiction, given that we managed to construct P *and* $\sim P$, as demonstrated by the following theorem, which states that from any Q can be derived from P and $\sim P$.

```
Theorem absurd: P → ~P → Q.
Proof. by move⇒p H; move : (H p). Qed.
```

One extremely useful theorem from propositional logic involving negation is *contraposition*. It states that in an implication, the assumption and the goal can be flipped if inverted.

```
Theorem contraP: (P → Q) → ~Q → ~P.
```

Let us see how it can be proved in Coq

```
Proof.
move⇒ H Hq.
move /H.
```

```
  H : P → Q
  Hq : ~Q
  ============================
   Q → False
```

The syntax `move / H` (spaces in between are optional) stands for one of the most powerful features of Ssreflect, called *views* (see Section 9 of [23]), which allows one to *weaken* the assumption in the goal part of the proof on the fly by applying a hypothesis H to the top assumption in the goal. In the script above the first command `move⇒ H Hq` simply popped two assumptions from the goal to the context. What is left is $\sim P$, or, equivalently $P \to$ **False**. The view mechanism then "interpreted" P in the goal via H and changing it to Q, since H was of type $P \to Q$, which results in the modified goal $Q \to$ **False**. Next, we apply the view Hq to the goal, which switches Q to **False**, which makes the rest of the proof trivial.

```
move /Hq.
done.
Qed.
```

3.6 Existential quantification

Existential quantification in Coq, which is denoted by the notation "`exists` x, P x" is just yet another inductive predicate with exactly one constructor:

```
Locate "exists".
```

```
"'exists' x .. y , p" := ex (fun x ⇒ .. (ex (fun y ⇒ p)) ..)
```

: *type_scope*

`Print` *ex*.

```
Inductive ex (A : Type) (P : A → Prop) : Prop :=
    ex_intro : ∀ x : A, P x → ex A P
```

The notation for existentially quantified predicates conveniently allows one to existentially quantify over several variables, therefore, leading to a chain of enclosed calls of the constructor *ex_intro*.

The inductive predicate **ex** is parametrized with a type A, over elements of which we quantify, and a predicate function of type $A \to$ `Prop`. What is very important is that the scope of the variable x in the constructor captures **ex** A P as well. That is, the constructor type could be written as $\forall\ x,\ (P\ x \to$ **ex** $A\ P)$ to emphasize that each particular instance of **ex** A P carries is defined by a *particular* value of x. The actual value of x, which satisfies the predicate P is, however, not exposed to the client, providing the *data abstraction* and information hiding, similarly to the traditional existential types (see Chapter 24 of [52]), which would serve as a good analogy. Each inhabitant of the type **ex** is therefore an instance of a *dependent pair*,[13] whose first element is a *witness* for the following predicate P, and the second one is a result of application of P to x, yielding a particular proposition.

The proofs of propositions that assume existential quantification are simply the proofs by case analysis: destructing the only constructor of **ex**, immediately provides its arguments: a witness x and the predicate P, satisfied by x. The proofs, where the existential quantification is a goal, can be completed by applying the constructor *ex_intro* directly or by using a specialized Coq's tactic `exists` z, which does exactly the same, instantiating the first parameter of the constructor with the provided value z. Let us demonstrate it on a simple example [3, §5.2.6], accounting for the weakening of the predicate, satisfying the existentially quantified variable.

```
Theorem ex_imp_ex A (S T: A → Prop):
  (exists a: A, S a) → (∀ x: A, S x → T x) → exists b: A, T b.
```

The parentheses are important here, otherwise, for instance, the scope of the first existentially-quantified variable a would be the whole subsequent statement, not just the proposition $S\ a$.

`Proof`.

First, we decompose the first existential product into the witness a and the proposition *Hst*, and also store the universally-quantified implication assumption with the name *Hst*.

`case`⇒*a Hs Hst*.

```
  A : Type
  S : A → Prop
  T : A → Prop
  a : A
```

[13] In the literature, dependent pairs are often referred to as *dependent sums* or Σ-types.

Hs : S *a*
Hst : ∀ *x* : *A*, S *x* → *T x*
==========
`exists` *b* : *A*, *T b*

Next, we apply the **ex**'s constructor by means of the `exists` tactic with an explicit witness value *a*:

`exists` *a*.

We finish the proof by applying the weakening hypothesis *Hst*.

`by apply`: *Hst*.

`Qed`.

Exercise 3.2. Let us define our own version **my_ex** of the existential quantifier using the Ssreflect notation for constructors:

`Inductive` *my_ex A* (*S*: *A* → `Prop`) : `Prop` := *my_ex_intro x* `of` *S x*.

The reader is invited to prove the following goal, establishing the equivalence of the two propositions

`Goal` ∀ *A* (*S*: *A* → `Prop`), *my_ex A S* `<->` `exists` *y*: *A*, *S y*.

Hint: the propositional equivalence `<->` is just a conjunction of two implications, so proving it can be reduced to two separate goals by means of `split` tactics.

3.6.1 A conjunction and disjunction analogy

Sometimes, the universal and the existential quantifications are paraphrased as "infinitary" conjunction and disjunction correspondingly. This analogy comes in handy when understanding the properties of both quantifications, so let us elaborate on it for a bit.

In order to prove the conjunction *P1* ∧ ... ∧ *Pn*, one needs to establish that *all* propositions *P1* ... *Pn* hold, which in the finite case can be done by proving *n* goals, for each statement separately (and this is what the `split` tactic helps to do). Similarly, in order to prove the propositions ∀ *x*: *A*, *P x*, one needs to prove that *P x* holds for *any x* of type *A*. Since the type *A* itself can define an infinite set, there is no way to enumerate all conjuncts, however, an explicit handle *x* gives a way to effectively *index* them, so proving *P x* for an arbitrary *x* would establish the validity of the universal quantification itself. Another useful insight is that in Coq ∀ *x*: *A*, *P x* is a type of a dependent function that maps *x* of type *A* to a value of type *P x*. The proof of the quantification would be, therefore, a function with a suitable "plot". Similarly, in the case of *n*-ary conjunction, the function has finite domain of exactly *n* points, for each of which an appropriate proof term should be found.

In order to prove the *n*-ary disjunction *P1* ∨ ... ∨ *Pn* in Coq, it is sufficient to provide a proof for just one of the disjunct *as well as* a "tag" — an indicator, which disjunct exactly is being proven (this is what tactics `left` and `right` help to achieve). In the case of infinitary disjunction, the existential quantification "`exists` *x*, *P x*", the existentially quantified variable plays role of the tag indexing all possible propositions *P x*. Therefore, in order to prove such a proposition, one needs first to deliver a witness *x* (usually, by means of calling the tactics `exists`), and then prove that for this witness/tag the

proposition $P\ x$ holds. Continuing the same analogy, the disjunction in the assumption of a goal usually leads to the proof by `case` analysis assuming that one of the disjuncts holds at a time. Similarly, the way to destruct the existential quantification is by case-analysing on its constructor, which results in acquiring the witness (i.e., the "tag") and the corresponding "disjunct".

Finally, the folklore alias "dependent product type" for dependent function types (i.e., $\forall$-quantified types) indicates its relation to products, which are Curry-Howard counterparts of conjunctions. In the same spirit, the term "dependent sum type" for the dependent types, of which existential quantification is a particular case, hints to the relation to the usual sum types, and, in particular *sum* (discussed in Chapter 2), which is a Curry-Howard dual of a disjunction.

`End` *Connectives*.

3.7 Missing axioms from classical logic

In the previous sections of this chapter, we have seen how a fair amount of propositions from the higher-order propositional logics can be encoded and proved in Coq. However, some reasoning principles, employed in *classical* propositional logic, cannot be encoded in Coq in a form of provable statements, and, hence, should be encoded as *axioms*.

In this section, we provide a brief and by all means incomplete overview of the classical propositional logic axioms that are missing in Coq, but can be added by means of importing the appropriate libraries. Chapter 12 of the book [7] contains a detailed survey of useful axioms that can be added into Coq development on top of CIC.

To explore some of some of the axioms, we first import that classical logic module Classical_Prop.

`Import` *Classical_Prop*.

The most often missed axiom is the axiom of *excluded middle*, which postulates that the disjunction of P and $\sim P$ is provable. Adding this axiom circumvents the fact that the reasoning out of the excluded middle principle is *non-constructive*, as discussed in Section 3.4.

`Check` *classic*.

classic
: $\forall\ P$: `Prop`, $P \vee \sim P$

Another axiom from the classical logic, which coincides with the type of Scheme's *call/cc* operator[14] (pronounced as *call with current continuation*) modulo Curry-Howard isomorphism is *Peirce's law*:

`Definition` *peirce_law* := $\forall\ P\ Q$: `Prop`, $((P \to Q) \to P) \to P$.

In Scheme-like languages, the *call/cc* operator allows one to invoke the undelimited continuation, which aborts the computation. Similarly to the fact that *call/cc* cannot be

[14] http://community.schemewiki.org/?call-with-current-continuation

implemented in terms of polymorphically-typed lambda calculus as a function and should be added as an external operator, the Peirce's law is an axiom in the constructive logic.

The classical double negation principle is easily derivable from Peirce's law, and corresponds to the type of *call/cc*, which always invokes its continuation parameter, aborting the current computation.

```
Check NNPP.
```

```
NNPP
     : ∀ P : Prop, ~~P → P
```

Finally, the classical formulation of the implication through the disjunction is again an axiom in the constructive logic, as otherwise from the function of type $P \to Q$ one would be able to construct the proof of $\sim P \vee Q$, which would make the law of excluded middle trivial to derive.

```
Check imply_to_or.
```

```
imply_to_or
     : ∀ P Q : Prop, (P → Q) → ~P ∨ Q
```

Curiously, none of these axioms, if added to Coq, makes its logic unsound: it has been rigorously proven (although, not within Coq, due to Gödel's incompleteness result) that all classical logic axioms are consistent with CIC, and, therefore, don't make it possible to derive the falsehood [66].

The following exercise reconciles most of the familiar axioms of classical logic.

Exercise 3.3 (Equivalence of classical logic axioms (from § 5.2.4 of [3]))**.** Prove that the following five axioms of the classical are equivalent.

```
Definition peirce := peirce_law.
Definition double_neg := ∀ P: Prop, ~~P → P.
Definition excluded_middle := ∀ P: Prop, P ∨ ~P.
Definition de_morgan_not_and_not := ∀ P Q: Prop, ~( ~P ∧ ~Q) → P ∨ Q.
Definition implies_to_or := ∀ P Q: Prop, (P → Q) → (~P ∨ Q).
```

Hint: Use `rewrite /d` tactics to unfold the definition of a value d and replace its name by its body. You can chain several unfoldings by writing `rewrite /d1 /d2` ... etc.
Hint: To facilitate the forward reasoning by contradiction, you can use the Ssreflect tactic `suff`: P, where P is an arbitrary proposition. The system will then require you to prove that P implies the goal *and* P itself.
Hint: Stuck with a tricky proof? Use the Coq *Admitted* keyword as a "stub" for an unfinished proof of a goal, which, nevertheless will be considered completed by Coq. You can always get back to an admitted proof later.

3.8 Universes and Prop impredicativity

While solving Exercise 3.3 from the previous section, the reader could notice an interesting detail about the propositions in Coq and the sort `Prop`: the propositions that quantify

over propositions still remain to be propositions, i.e., they still belong to the sort `Prop`. This property of propositions in Coq (and, in general, the ability of entities of some class to abstract over the entities of the same class) is called *impredicativity*. The opposite characteristic (i.e., the inability to refer to the elements of the same class) is called *predicativity*.

One of the main challenges when designing the Calculus of Constructions was to implement its logical component (i.e., the fragment responsible for constructing and operating with elements of the `Prop` sort), so it would subsume the existing impredicative propositional calculi [12], and, in particular, System F (which is impredicative), allowing for the expressive reasoning in higher-order propositional logic.

Impredicativity as a property of definitions allows one to define domains that are *self-recursive*—a feature of `Prop` that we recently observed. Unfortunately, when restated in the classical set theory, impredicativity immediately leads to the famous paradox by Russell, which arises from the attempt to define the set of all sets that do not belong to themselves. In terms of programming, Russell's paradox provides a recipe to encode a fixpoint combinator in the calculus itself and write generally-recursive programs.

System F is not a dependently-typed calculus and it has been proven to contain no paradoxes [21], as it reasons only about *types* (or, *propositions*), which do not depend on values. However, adding dependent types to the mix (which Coq requires to make propositions quantify over arbitrary values, not just other propositions, serving as a general-purpose logic) makes the design of a calculus more complicated, in order to avoid paradoxes akin to the Russell's, which arise from mixing values and sets of values. This necessity to "cut the knot" inevitably requires to have a sort of a higher level, which contains all sets and propositions (i.e., the whole sorts `Set` and `Prop`), but does not contain itself. Let us call such sort `Type`. It turns out that the self-inclusion `Type : Type` leads to another class of paradoxes [11], and in order to avoid them, the hierarchy of higher-order sorts should be made infinite and *stratified*. Stratification means that each sort has a level number, and is contained in a sort of a higher level but not in itself. The described approach is suggested by Martin-Löf [38] and adopted literally, in particular, by Agda [45]. The stratified type sorts, following Martin-Löf's tradition, are usually referred to as *universes*.

A similar stratification is also implemented in Coq, which has its own universe hierarchy, although with an important twist. The two universes, `Set` and `Prop` cohabit at the first level of the universe hierarchy with `Prop` being impredicative. The universe containing both `Set` and `Prop` is called `Type@{Set+1}`, and it is *predicative*, as well as all universes that are above it in the hierarchy. CIC therefore remains consistent as a calculus, only because of the fact that all impredicativity in it is contained at the very bottom of the hierarchy.

3.8.1 Exploring and debugging the universe hierarchy

In the light of Martin-Löf's stratification, the Coq' non-polymorphic types, such as *nat*, **bool**, *unit* or **list** *nat* "live" at the 0th level of universe hierarchy, namely, in the sort `Set`. The polymorphic types, quantifying over the elements of the `Set` universe are, therefore located at the higher level, which in Coq is denoted as `Type@{Set+1}`, but in the displays is usually presented simply as `Type`, as well as all the higher universes. We can enable

the explicit printing of the universe levels to see how they are assigned:

```
Set Printing Universes.
Check bool.
```

```
bool
     : Set
```

```
Check Set.
```

```
Set
     : Type@{Set+1}
```

```
Check Prop.
```

```
Prop
     : Type@{Set+1}
```

The following type is polymorphic over the elements of the Set universe, and this is why its own universe is "bumped up" by one, so it now belongs to Set+1.

```
Definition S := ∀ T: Set, list T.
Check S.
```

```
S
     : Type@{Set+1}
```

Until version 8.5, Coq used to provide a very limited version of *universe polymorphism*. To illustrate the idea, let us consider the next definition R, which is polymorphic with respect to the universe of its parameter A, so its result is assumed to "live" in the universe, whose level is taken to be the level of A.

```
Definition R (A: Type) (x: A): A := x.
Arguments R [A].
Check R tt.
```

```
R tt
     : unit
  (*  |= Set <= Top.93  *)
```

The part in comments show the inequality, generated by the Coq unification algorithm that had to be solved in order to determine the universe level of the value R tt with *Top*.93 being the level, assigned to R itself.

If the argument of R is itself a universe, it means that A's level is higher than x's level, and so is the level of R's result.

```
Check R Type.
```

R Type@{Top.94}
: *Type@{Top.95}*

```
(*  Top.94
    Top.95 |= Top.94 < Top.95
              Top.95 < Top.93  *)
```

The Coq's unifier algorithm in this case looks for a universe levels *Top*.95, which can be larger than the level of *R*'s argument level *Top*.94, but smaller than the one of *R* itself (i.e., *Top*.93). In the absence of other constraints, such system of equalities is easily satisfiable.

However, the attempt to apply *R* to *itself* immediately leads to an error reported, as the system cannot infer the level of the result, by means of solving a system of universe level inequations, therefore, preventing meta-circular paradoxes.

`Check` *R R*.

The term "R" *has* `type` "forall A : Type@{Top.93}, A -> A"
while it `is` *expected to have* `type` "?A"
(*unable to find a well-typed instantiation* `for` "?A": *cannot ensure that* "Type@{Top.93+1}" `is` *a subtype* ***`of`*** "Type@{Top.93}").

The solution to this problem is to think of the two occurrences *R* in the last example as of inhabitants of *different* universes, so that the *R*-"function" belongs to the universe with a higher level number than *R*-"argument". Checking this scenario requires supporting a more flexible form of universe polymorphism, which can assign different universe levels to different occurrences of the same definition in a common expression, and in this sense reminds *let-polymorphism* [52, §22.7]. This feature was introduced in Coq since version 8.5 [60], and it allows us to redefine *R* as universe-polymorphic via the new *Polymorphic* keyword.[15]

Polymorphic `Definition` *RPoly* {*A* : `Type`} (*a* : *A*) : *A* := *a*.

`About` *RPoly*.

RPoly : ∀ *A* : *Type@{Top.96}*, *A* → *A*

```
(* Top.96 |=  *)
```

RPoly `is` *universe polymorphic*
Argument A `is` *implicit* **and** *maximally inserted*
...

We can now apply RPoly to itself using the following syntax.

`Definition` *selfRPoly* := *RPoly* (@*RPoly*).

selfRPoly `is` *defined*

[15] More documentation on universe polymorphism is available at https://coq.inria.fr/distrib/V8.5beta2/refman/Reference-Manual032.html.

Let us now check the details of universes participating in selfRPoly's typing.

`Print` *selfRPoly.*

```
selfRPoly =
RPoly@{Top.97} (@RPoly@{Top.98})
     : ∀ A : Type@{Top.98}, A → A

  (*  Top.97
      Top.98 |= Top.98 < Top.97
      *)
```

The display above demonstrates that the two occurrences of RPoly were assumed by the typing algorithm to belong to different universes: *Top*.97 for the function and *Top*.98 for the argument, correspondingly. In the absence of additional additional constraints, the inequality between them can be trivially satisfied by assuming $Top.98 < Top.97$.

4 Equality and Rewriting Principles

In the previous chapter we have seen how main connectives from propositional logic are encoded in Coq. However, the mathematical reasoning only by means of propositional logic is still quite limited. In particular, by this moment we are still unable to state what does it mean for two objects to be *equal*. In this chapter we are going to see how equality can be implemented in Coq. Moreover, the statement "x is equal to y" automatically gives us a way to replace y by x and vice versa in the process of reasoning, therefore implementing a discipline of *rewriting*—one of the key ingredients of the mathematical proof.[1] Later in the chapter, we will see how rewriting by equality is just a particular case of a general proof pattern, which allows one to define arbitrary *rewriting rules* by exploiting Coq's mechanism of *indexed type families*.

```
From mathcomp
Require Import ssreflect ssrnat ssrbool eqtype.
Unset Strict Implicit.
Unset Printing Implicit Defensive.
```

4.1 Propositional equality in Coq

Let us begin by exploring the definition of the equality predicate "_ = _".

```
Locate "_= _".
```

"x = y" := **eq** x y : *type_scope*

```
Print eq.
```

Inductive **eq** (A : Type) (x : A) : $A \to$ Prop := *eq_refl* : **eq** x x

As we can see, the equality is just yet another inductive predicate, similar to the logical connectives we've seen in Chapter 3. However, there are differences, which are of importance. First, equality as a predicate is *parametrized* over two arguments: a `Type` A of an unspecified universe (so, it can be `Set`, `Prop` or any of the higher universes) and an element x of type A. There is nothing particularly new here: we have seen parametrized inductive predicates before, for instance, conjunction and disjunction in Section 3.4. The novel part of this definition is what comes after the colon trailing the parameter list. Unlike all previously seen logical connectives, the equality predicate has type $A \to$ `Prop`

[1] The reader could have, probably, heard how mathematics sometimes is referred to as a "science of rewritings".

in contrast to just `Prop`. In the Coq terminology, it means that **eq** is not just inductively-defined datatype, but is an *indexed type family*. In this particular case, it is indexed by elements of type A, which appears at the left of the arrow.

It is common to think of indexed type families in Coq as of *generalized algebraic datatypes* (GADTs) [50, 69], familiar from Haskell, and allowing one to refine the process pattern matching basing on the type index of the scrutinee. However, another analogy turns out to be much more useful in the Coq setting: indexed type families in fact allow one to encode *rewriting principles*. To understand, what the indexed datatype definition has to do with rewriting, let us take a close look at the definition of **eq**. The type of its only constructor *eq_refl* is a bit misleading, as it looks like it is applied to two arguments: x and ... x. To disambiguate it, we shall put some parentheses, so, in fact, it should read as

`Inductive` **eq** $(A : \texttt{Type})\ (x : A) : A \to \texttt{Prop} :=$ *eq_refl* $: (\textbf{eq}\ x)\ x$

That is, the constructor *eq_refl* delivers an element of type (**eq** x), whose *parameter* is some x (and **eq** is directly applied to it), and its *index* (which comes second) is constrained to be x as well. That is, case-analysing on an instance of **eq** $x\ y$ in the process of the proof construction will inevitably lead the side condition implying that x and y actually correspond to the *same object*. Coq will take advantage of this fact immediately, by performing the *unification* and substituting all occurrences of y in the subsequent goal with x. Let us see how it works in practice.

4.1.1 Case analysis on an equality witness

To demonstrate the actual proofs on the case analysis by equality, we will have to perform an awkward twist: define *our own* equality predicate.

```
Set Implicit Arguments.
Inductive my_eq (A : Type) (x : A) : A → Prop := my_eq_refl : my_eq x x.
Notation "x === y" := (my_eq x y) (at level 70).
```

As we can see, this definition literally repeats the Coq's standard definition of propositional equality. The reason for the code duplication is that Ssreflect provides a specific treatment of Coq's standard equality predicate, so the case-analysis on its instances is completely superseded by the powerful `rewrite` tactics, which we will see in Section 4.2 of this chapter. Alas, this special treatment also leads to a non-standard behaviour of case-analysis on equality. This is why, for didactical purposes, we will have to stick with or own home-brewed definition until the end of this section.

Let us now prove some interesting properties of the freshly-defined equality. We start with symmetry of $===$ by formulating the following lemma:[2]

`Lemma` *my_eq_sym* $A\ (x\ y\colon A) : x === y \to y === x.$

First, we perform the case analysis on the top assumption of the goal, $x === y$.

```
case.
```

[2]The Coq's command `Lemma` is identical to `Theorem`.

```
A : Type
x : A
y : A
============================
 x === x
```

This leads to the goal, being switched from $y === x$ to $x === x$, as all occurrences of y are now replaced by x, exactly as advertised. We can now finish the proof by applying the constructor (`apply`: *my_refl_eq*) or simply by `done`, which is powerful enough to figure out what to apply.

```
done.
Qed.
```

Our next exercise will be to show that the predicate we have just defined implies Leibniz equality. The proof is accomplished in one line by case-analysing on the equality, which leads to the automatic replacements of y by x.

```
Lemma my_eq_Leibniz A (x y: A) (P: A → Prop) : x === y → P x → P y.
Proof. by case. Qed.
```

4.1.2 Implementing discrimination

Another important application of the equality predicate family and similar ones are *proofs by discrimination*, in which the contradiction is reached (i.e., the falsehood is derived) out of the fact that two clearly non-equal elements are assumed to be equal. The next lemma demonstrates the essence of the proof by discrimination using the **my_eq** predicate.

```
Lemma disaster : 2 === 1 → False.
Proof.
move⇒ H.
```

```
H : 2 === 1
============================
 False
```

As it is already hinted by the name of the method, the key insight in the proofs by discrimination is to construct a function that can distinguish between values of the type with an implicit *definitional equality*, which relates two values if they have identical structure.[3] In particular, natural numbers can be compared against each other by means of direct pattern matching, which is decidable for them, thanks to the inductive definition. Using this insight we define a local "discriminating" function D using the Ssreflect's enhanced `pose` tactic:

```
pose D x := if x is 2 then False else True.
```

[3]It is not trivial to establish computable definitional equality on *any* values, as the values might be of an infinite nature. For instance, stating the equality of two functions would require checking their results on all elements of the common domain, which might be infinite. in this respect, propositional equality acts like it "compares the references", whereas definitional equality "compares the structure" of two elements.

```
H : 2 === 1
D := fun x : nat ⇒
     match x with
     | 0 ⇒ True
     | 1 ⇒ True
     | 2 ⇒ False
     | S (S (S _)) ⇒ True
     end : nat → Prop
============================
 False
```

Now, proving $D\ 1$ is **True** can be accomplished by simple executing D with appropriate arguments (recall that D is an always-terminating function, whose result is a computable value). That Ssreflect's tactic `have` allows to declare the local fact, which can be then proved in-place by simple computation (which is performed via `by []`).

```
have D1: D 1.
by [].
```

```
H : 2 === 1
D := ...
D1 : D 1
============================
 False
```

Next we "push" *D1* and *H* back to the goal (using the : tactical), and case-analyse on the top assumption *H*. Notice that the semantics of : is such that it first performs a series of "pushings" and then runs the tactic on the left of itself (i.e., `case`).

```
case: H D1.
```

```
D := ...
============================
 D 2 → False
```

Now, we got what we have needed: the proof of the falsehood! Thanks to the equality-provided substitution, $D\ 1$ turned into $D\ 2$, and the only thing that remains now is to *evaluate* it.

```
move=>/=.
```

The tactical /=, coming after ⇒ runs all possible simplifications on the result obtained by the tactics, preceding ⇒, finishing the proof.

```
done.
Qed.
```

Let us provide a bit more explanation how did it happen that we managed to derive the falsehood in the process of the proof. The discrimination function D is a function from **nat** to `Prop`, and, indeed, it can return **True** and **False**, so it contains no contradictions

by itself. We also managed to prove easily a trivial proposition $D\ 1$, which is just **True**, so it's derivable. The genuine twist happened when we managed to turn the assumption $D\ 1$ (which was **True**) to $D\ 2$ (which is **False**). This was only possible because of the assumed equality $2 === 1$, which contained the "falsehood" from the very beginning and forced Coq to substitute the occurrence of 1 in the goal by 2, so the discrimination function in the assumption finished the job.

Exercise 4.1. Let us change the statement of a previous lemma for a little bit:

`Lemma` *disaster2* : $1 === 2 \to$ *False*.

Now, try to prove it using the same scheme. What goes wrong and how to fix it?

4.1.3 Reasoning with Coq's standard equality

Now we know what drives the reasoning by equality and discrimination, so let us forget about the home-brewed predicate **my_eq** and use the standard equality instead. Happily, the discrimination pattern we used to implement "by hand" now is handled by Coq/Ssreflect automatically, so the trivially false equalities deliver the proofs right away by simply typing `done`.

```
Lemma disaster3: 2 = 1 → False.
Proof. done. Qed.
```

Moreover, the case-analysing on the standard equality now comes in the form of the powerful `rewrite` tactics, which takes the reasoning to the whole new level and is a subject of the next section.

4.2 Proofs by rewriting

The vast majority of the steps when constructing real-life proofs in Coq are *rewriting* steps. The general flow of the interactive proof (considered in more detail in Chapter 6) is typically targeted on formulating and proving small auxiliary hypotheses about equalities in the forward-style reasoning and then exploiting the derived equalities by means of rewriting in the goal and, occasionally, other assumptions in the context. All rewriting machinery is handled by Ssreflect's enhanced `rewrite` tactics, and in this section we focus on its particular uses.

4.2.1 Unfolding definitions and in-place rewritings

One of the common uses of the `rewrite` tactic is to fold/unfold transparent definitions. In general, Coq is capable to perform the unfoldings itself, whenever it's required. Nevertheless, manual unfolding of a definition might help to understand the details of the implementation, as demonstrated by the following example.

```
Definition double {A} (f: A → A) (x: A) := f (f x).
Fixpoint nat_iter (n : nat) {A} (f : A → A) (x : A) : A :=
  if n is S n' then f (nat_iter n' f x) else x.
```

```
Lemma double2 A (x: A) f t:
  t = double f x → double f t = nat_iter 4 f x.
Proof.
```

The first thing to do in this proof is to get rid of the auxiliary variable t, as it does not occur in any of the assumptions, but just in the subsequent goal. This can be done using the following sequence of tactics that first moves the equality assumption to the top and then rewrites by it in the goal.

```
move⇒Et; rewrite Et.
```

```
  A : Type
  x : A
  f : A → A
  t : A
  Et : t = double f x
  ============================
   double f (double f x) = nat_iter 4 f x
```

Even though the remaining goal is simple enough to be completed by `done`, let us unfold both definition to make sure that the two terms are indeed equal structurally. Such unfoldings can be *chained*, just as any other rewritings.

```
rewrite /double /nat_iter.
```

```
  x : A
  f : A → A
  ============================
   f (f (f (f x))) = f (f (f (f x)))
```

An alternative way to prove the same statement would be to use the `->` tactical, which is usually combined with `move` or `case`, but instead of moving the assumption to the top, it makes sure that the assumption is an equality and rewrites by it.

```
Restart.
by move=>->; rewrite /double.
Qed.
```

Notice that the tactical has a companion one `<-`, which performs the rewriting by an equality assumption from right to left, in contrast to `->`, which rewrites left to right.

Folding, the reverse operation to unfolding, is done by using `rewrite -/...` instead of `rewrite /...`[4]

4.2.2 Proofs by congruence and rewritings by lemmas

```
Definition f x y := x + y.
Goal ∀ x y, x + y + (y + x) = f y x + f y x.
```

[4] As the reader will notice soon, it is a general pattern with Ssreflect's rewriting to prefix a `rewrite` argument with `-`, if the *reverse* rewriting operation should be performed.

```
Proof.
move⇒ x y.
```

First, let us unfold only all occurrences of `f` in the goal.

```
rewrite /f.
```

```
  x : nat
  y : nat
  ============================
   x + y + (y + x) = y + x + (y + x)
```

We can now reduce the goal by appealing to Ssreflect's `congr` tactics, which takes advantage of the fact that equality implies Leibniz' equality, in particular, with respect to the addition taken as a function, so the external addition of equal elements can be "stripped off".

```
congr (_ + _).
```

```
  x : nat
  y : nat
  ============================
   x + y = y + x
```

Now, the only thing left to prove is that the addition is commutative, so at this point we will just make use of Ssreflect's ssrnat library lemma for integer addition.

```
Check addnC.
```

```
addnC
     : ssrfun.commutative addn
```

At this point such signature might seem a bit cryptic, but worry not: this is just a way to express in a generic way that the addition over natural numbers is commutative, which can be witnessed by checking the definition of *ssrfun.commutative* predicate:

```
Print ssrfun.commutative.
```

```
ssrfun.commutative =
  fun (S T : Type) (op : S → S → T) ⇒ ∀ x y : S, op x y = op y x
       : ∀ S T : Type, (S → S → T) → Prop
```

As we can see, the definition of the commutative predicate ensures the equality of the operation's result with its arguments, permuted, hence *op x y = op y x*. The type of the lemma addnC therefore refines *op* to be "_ + _", so, after specializing the definition appropriately, the type of addnC should be read as:

```
addnC
     : ∀ n m: nat, n + m = m + n
```

Now, we can take advantage of this equality and rewrite by it a part of the goal. Notice that Coq will figure out how the universally-quantified variables should be instantiated (i.e., with y and x, respectively):

```
by rewrite [y + _]addnC.
Qed.
```

The *r-pattern* (regex pattern) $[y + \_]$, preceding the lemma to be used for rewriting, specifies, which subexpression of the goal should be a subject of rewriting. When non-ambiguous, some parts of the expressions can be replaced by wildcard underscores _. In this particular case, it does not matter that much, since any single rewriting by commutativity in any of the sums, on the left or on the right, would make the proof to go through. However, in a more sophisticated goal it makes sense to specify explicitly, what should be rewritten:

```
Goal ∀ x y z, (x + (y + z)) = (z + y + x).
Proof.
by move⇒x y z; rewrite [y + _]addnC; rewrite [z + _ + _]addnC.
Qed.
```

Proofs of "obvious" equalities that hold modulo, e.g., commutativity and subjectivity, usually require several rewriting to be established, which might be tedious. There are ways to automate such proofs by means of overloaded lemmas via *canonical structures*. These techniques, hinted briefly in Chapter 7, are mostly outside of the scope of this course, so we address the reader to a number of papers, presenting the state of the art in this direction [25, 37].

4.2.3 Naming in subgoals and optional rewritings

When working with multiple cases, it is possible to "chain" the execution of several tactics. Then, in the case of a script *tac1*; *tac2*, if the goal is replaced by several after applying *tac1*, then *tac2* will be applied to *all* subgoals, generated by *tac1*. For example, let us consider a proof of the following lemma from the standard ssrnat module:

```
Lemma addnCA: ∀ m n p, m + (n + p) = n + (m + p).
Proof.
move⇒m n.
```

```
  m : nat
  n : nat
  ============================
   ∀ p : nat, m + (n + p) = n + (m + p)
```

The proof will proceed by induction on m. We have already seen the use of the case tactics, which just performs the case analysis. Another Ssreflect tactic elim generalizes case by applying the default induction principle (*nat_ind* in this case) with the respect to the remaining goal (that is, the predicate $[\forall\ p : \textbf{nat},\ m + (n + p) = n + (m + p)]$) is to be proven by induction. The following sequence of tactics proceeds by induction on m with the default induction principle. It also names some of the generated assumptions.

```
elim: m=>[ | m Hm ] p.
```

In particular, the following steps are performed:

- m is pushed as a top assumption of the goal;
- `elim` is run, which leads to generation of the two goals;
  - The first goal is of the shape

    $\forall\ p : \mathbf{nat},\ 0 + (n + p) = n + (0 + p)$
  - The second goal has the shape

    $\forall\ n0 : \mathbf{nat},$
    $(\forall\ p : \mathbf{nat},\ n0 + (n + p) = n + (n0 + p)) \to$
    $\forall\ p : \mathbf{nat},\ n0.{+}1 + (n + p) = n + (n0.{+}1 + p)$
- The subsequent structured naming $\Rightarrow$ [$|m\ Hm$] p names zero assumptions in the first goal and the two top assumptions, m and Hm, in the second goal. It then next names the assumption p in *both* goals and moves it to the top.

The first goal can now be proved by multiple rewritings via the lemma addOn, stating that 0 is the left unit with respect to the addition:

`by rewrite` $!add0n$.

The second goal can be proved by a series of rewritings using the fact about the ($\_ + 1$) function:

`by rewrite` $!addSnnS\ -addnS$.

Notice that the conclusion of the addnS lemma is rewritten right-to-left.

The whole proof could be, however, accomplished in one line using the *optional* rewritings. The intuitions is to *chain* the rewritings in the goals, generated by `elim` in a way that the unsuccessful rewriting would not fail the whole proof construction, as they are irrelevant for some goals anyway. This is how it can be done:

```
Restart.
by move⇒m n; elim: m=>[ | m Hm ] p; rewrite ?add0n ?addSnnS -?addnS.
Qed.
```

Notice that the optional rewritings (e.g., ?addSnnS) are performed as many times as they can be.

4.2.4 Selective occurrence rewritings

Sometimes, instead of providing an r-pattern to specialize the rewriting, it is more convenient to specify, which particular syntactic occurrences in the goal term should be rewritten. This is demonstrated by the following alternative proof of commutativity of addition from the lemma addnCA, which we have proved before:

```
Lemma addnC: ∀ m n, m + n = n + m.
Proof.
by move⇒ m n; rewrite -{1}[n]addn0 addnCA addn0.
Qed.
```

The first rewriting with addn0 "adds" 0 to the first occurrence of addn0, so the left-hand side of the equality becomes $m + (n + 0)$. The next rewriting employs the lemma addnCA, so we get $n + (m + 0) = n + m$ as the goal, and the last one "removes" zero, so the result trivially follows.

We conclude this section by noticing that the same rewriting machinery is applicable not only to the goal, but also to hypotheses in the assumption context using the `rewrite` *H1* `in` *H2* syntax (where *H1* is the rewriting hypothesis and *H2* is a hypothesis, where the rewriting should happen). There are many more tricks that can be done with rewritings, and we address the reader to Chapter 7 of Ssreflect manual [23].

4.3 Indexed datatype families as rewriting rules

In Section 4.1 of this chapter we have already seen how defining indexed datatype families makes it possible for Coq to provide a convenient rewriting machinery, which is implicitly invoked by case analysis on such families' refined types, thanks to sophisticated Coq's unification procedure.

Although so far this approach has been demonstrated by only one indexed type family example—propositional equality, defined by means of the **eq** family, in this section, concluding the chapter, we will show how to define other client-specific rewriting rules. Let us start from a motivating example in the form of an "obvious" lemma.

`Lemma` *huh n m*: $(m \leq n) \wedge (m > n) \rightarrow$ *False*.

From now on, we will be consistently including yet another couple of Ssreflect modules, ssrbool and eqtype, into our development. The need for them is due to the smooth combination of reasoning with `Prop`ositions and **bool**eans, which is a subject of the next chapter. Even though in Ssreflect's library, relations on natural numbers, such as $\leq$ and $>$, are defined as *boolean* functions, so far we recommend to the reader to think of them as of predicates defined in `Prop` and, therefore, valid arguments to the $\wedge$ connective.

Although the statement is somewhat obvious, in the setting of Coq's inductive definition of natural numbers it should be no big surprise that it is proved by induction. We present the proof here, leaving the details aside, so the reader could figure them out on her own, as a simple exercise.

```
Proof.
suff X: m ≤ n → ~(m > n) by case=>/X.
by elim: m n ⇒ [ | m IHm ] [ | n] //; exact: IHm n.
Qed.
```

Even this small example should make it feel like "something is not right", as a trivial mutual exclusion property required some inductive reasoning. A bigger problem is, however, that this mutual exclusion does not directly provide us with a "case-analysis" principle, which a human prover would naturally employ when reasoning about, for instance, a natural definition of the "maximum" function

`Definition` *maxn m n* := `if` $m < n$ `then` *n* `else` *m*.

and the following fact about its correctness

`Lemma` max_is_max *m n*: $n \leq$ maxn *m n* $\wedge$ *m* $\leq$ maxn *m n*.

The stated lemma max_is_max can be, indeed, proved by induction on m and n, which is a rather tedious exercise, so we will not be following this path.

4.3.1 Encoding custom rewriting rules

In the rest of this section, we will leverage the intuition behind indexed type families considered as *rewriting rules*, and will try to encode a "truth table" with two disjoint variants of relation between n and m, namely, $m \leq n$ and $n < m$. The table itself is encoded by the following inductive definition:

```
Inductive leq_xor_gtn m n : bool → bool → Set :=
  | LeqNotGtn of m ≤ n : leq_xor_gtn m n true false
  | GtnNotLeq of n < m : leq_xor_gtn m n false true.
```

However, this is not yet enough to enjoy the custom rewriting and case analysis on these two variant. At this moment, the datatype family **leq_xor_gtn**, whose constructors' indices encode a truth table's "rows", specifies two substitutions in the case when $m \leq n$ and $n < m$, respectively and diagrammatically looks as follows:

```
          |   C1  |   C2
-------------------------
m <= n    | true  | false
-------------------------
n < m     | false | true
```

The boolean values in the cells specify what the values of C1 and C2 will be substituted *with* in each of the two cases. However, the table does not capture, what to substitute them *for*. Therefore, our next task is to provide suitable variants for C1 and C2, so the table would describe a real situation and capture exactly the "case analysis" intuition. This values of the columns are captured by the following lemma, which, informally speaking, states that the table with this particular values of C1 and C2 "makes sense".

```
Lemma leqP m n : leq_xor_gtn m n (m ≤ n) (n < m).
Proof.
rewrite ltnNge.
by case le_mn: (m ≤ n); constructor=>//; rewrite ltnNge le_mn.
Qed.
```

Moreover, the lemma leqP, which we have just proved, delivers the necessary instance of the "truth" table, which we can now case-analyse against.[5]

4.3.2 Using custom rewriting rules

Let us see now, how some proofs might be changed to the good:

[5] In theory, a different lemma could be proven for the same table but for different values of indices, which would give us a *different* rewriting principle. However, the datatype family **leq_xor_gtn**, as it's currently specified, is too "tight" to admit other instances than the one provided by the lemma leqP, thanks to the explicit constructors' arguments: $m \leq n$ and $n < m$.

`Lemma` *huh' n m*: $(m \leq n) \wedge (m > n) \rightarrow$ *False*.
`Proof`.

Let us first "switch" from the propositional conjunction $\wedge$ to the boolean one && using the *view* mechanism by using the `move` tactics the trailing tactical `/`. This trick might look a bit unfair at the moment, but it will be soon explained in Section 5.1 of Chapter 5.

`move/`*andP*.

```
  n : nat
  m : nat
  ============================
   m ≤ n < m → False
```

The top assumption $m \leq n < m$ of the goal is just a syntactic sugar for $(m \leq n)$ && $(n < m)$. It is time now to make use of our rewriting rule/truth table, constructed by means of leqP.

`case:`*leqP*.

```
  n : nat
  m : nat
  ============================
   m ≤ n → true && false → False

subgoal 2 (ID 638) is:
 n < m → false && true → False
```

We would recommend to try stepping this line several times, back and forth to see, what is happening. Two goals were generated, so let us focus on the first one, as the second one will proceed by analogy. Case-analysing on the statement of the lemma leqP resulted in two different "options", as one would expect from the shape of the table. The first, case, $m \leq n$, resulted in generating the assumption $m \leq n$, as it is an argument of the corresponding constructor. What is more important, *all* occurrences of the columns' values were replaced in the goal by the corresponding boolean values, just as it was encoded in the table! The similar thing happened with the second goal, which encoded the alternative case, i.e., $n < m$.

Now, considering a boolean value true && false in a goal simply as a proposition (true && false) = true, the proof is trivial by simplification of the boolean conjunction.

```
done.
done.
Qed.
```

The proof of huh' is now indeed significantly shorter than the proof of its predecessor, huh. However, it might look like the definition of the rewriting rule **leq_xor_gtn** and its accompanying lemma leqP is quite narrowly-scoped, and it is not clear how useful it might be for other proofs.

To demonstrate the custom rewriting rules defined by means of indexed datatype families in their shine, let us get back to the definition of maxn and the lemma about it:

`Lemma` *max_is_max m n*: $n \leq maxn\ m\ n \wedge m \leq maxn\ m\ n$.
`Proof`.

The proof begins by unfolding the definition of maxn.
`rewrite` /*maxn*.

```
  m : nat
  n : nat
  ============================
   n ≤ (if m < n then n else m) ∧ m ≤ (if m < n then n else m)
```

We are now in the position to unleash our rewriting rule, which, together with simplifications by means of the // tactical does most of the job.

`case`: *leqP*=>//.

```
  m : nat
  n : nat
  ============================
   m < n → n ≤ n ∧ m ≤ n
```

The rest of the proof employs rewriting by some trivial lemmas from ssrnat, but conceptually is very easy.

`move`⇒*H*; `split`.
`by apply`: *leqnn*.
`by rewrite` *ltn_neqAle* `in` *H*; `case`/*andP*: *H*.
`Qed`.

The key advantage we got out of using the custom rewriting rule, defined as an indexed datatype family is lifting the need to prove *by induction* a statement, which one would intuitively prove by means of *case analysis*. In fact, all inductive reasoning was conveniently "sealed" by the proof of leqP and the lemmas it made use of, so just the tailored "truth table"-like interface for case analysis was given to the client.

We invite the reader to exercise in using the custom rewriting rules by proving a series of properties of maxn.

Exercise 4.2. Prove the following lemmas about maxn.

`Lemma` *max_l m n*: $n \leq m \rightarrow maxn\ m\ n = m$.

`Lemma` *succ_max_distr n m* : $(maxn\ n\ m).+1 = maxn\ (n.+1)\ (m.+1)$.

`Lemma` *plus_max_distr_l m n p*: $maxn\ (p + n)\ (p + m) = p + maxn\ n\ m$.

Hint: It might be useful to employ the lemmas ltnNge, leqNgt, ltnS and similar from Ssreflect's ssrnat module. Use the `Search` command to find propositions that might help you to deal with the goal.
Hint: Forward-style reasoning via `suff` and `have` might be more intuitive.
Hint: A hypothesis of the shape H: $n < m$ is a syntactic sugar for H: $n < m = \mathsf{true}$, since $n < m$ in fact has type **bool**, as will be explained in Chapter 5.

We conclude this section and the chapter by showing an instance of a more sophisticated custom rewriting rule, which now encodes a three-variant truth table for the ordering relations on natural numbers.

```
Inductive nat_rels m n : bool → bool → bool → Set :=
  | CompareNatLt of m < n : nat_rels m n true false false
  | CompareNatGt of m > n : nat_rels m n false true false
  | CompareNatEq of m = n : nat_rels m n false false true.
```

Exercise 4.3 (Comparing natural numbers as a rewriting rule)**.** Prove the following rewriting lemma for **nat_rels**:

```
Lemma natrelP m n : nat_rels m n (m < n) (n < m) (m == n).
```

Exercise 4.4. Let us define the minimum function minn on natural numbers as follows:

```
Definition minn m n := if m < n then m else n.
```

Prove the following lemma about *minm* and maxn:

```
Lemma addn_min_max m n : minn m n + maxn m n = m + n.
```

5 Views and Boolean Reflection

In Chapter 4, we have seen how custom rewriting rules and truth tables can be encoded in Coq using its support for indexed datatype families, so they are of great help for constructing the proofs by case analysis and rewriting. In this chapter, we will show how the custom rewriting machinery can be taken to the whole new level and be used to facilitate the reasoning about *computable* properties and predicates. We will consider a series of insights that lead to the idea of the *small-scale reflection*, the heart of the Ssreflect framework, which blurs the boundaries between computable predicates defined in the sort `Prop` (see Chapter 3) and Coq's recursive functions returning a result of type **bool** (in the spirit of the definitions that we have seen in Chapter 2). That said, in the vast number of cases these two are just the sides of the same coin and, hence, should be treated uniformly, serving to facilitate the reasoning in two different directions:

- expressing quantifications and building the proofs by means of *constructive reasoning* with logical connectives as datatypes defined in the sort `Prop`;
- employing brute-force computations for quantifier-free goals within the Coq framework itself, taken as a programming language, in order to reduce the goals to be proved by means of simply *computing* them.

We will elaborate more on the differences between predicates stated by means of `Prop` and **bool** in Section 5.2. The term *small-scale reflection*, which gives the name to the whole framework of Ssreflect, emphasizes the two complementary ways of building proofs: by means of intuitionistic inference (i.e., using the constructors of datatypes defined in `Prop`) and by means of mere computation (i.e., with **bool**-returning function). These two ways, therefore, serve as each other's "reflections" and, moreover, both are implemented within the same system, without the need to appeal to Coq's meta-object protocol,[1] which makes this reflection *small-scale*.

Unfortunately, the proper explanation of the implementation of the reflection mechanism between `Prop` and **bool** in Ssreflect strongly relies on the *views* machinery, so let us begin by describing it first.

[1] In contrast, reflection mechanism in Java, Python or Ruby actually does appeal to the meta-object protocol, e.g., the structure of the classes, which lies beyond the formally defined semantics of the language itself and, hence, allows one to modify the program's behaviour at runtime.

5.1 Proving with views in Ssreflect

```
From mathcomp
Require Import ssreflect ssrnat prime ssrbool eqtype.
```

Let us assume we have the following implication to prove:

```
Lemma imp_trans4 P Q R S: (P → Q) → (R → S) → (Q → R) → P → S.
Proof.
move⇒H1 H2 H3.
```

```
  P : Type
  Q : Type
  R : Type
  S : Type
  H1 : P → Q
  H2 : R → S
  H3 : Q → R
  ============================
   P → S
```

Since we are proficient in the proofs via implications, it is not difficult to construct the explicit proof term by a series of `apply:` tactic calls or via the `exact:` tactic, as it has been show in Chapter 3. Let us do something different, though, namely *weaken* the top assumption of the goal by means of applying the hypothesis *H1* to it, so the overall goal will become `Q → S`.

```
move⇒p; move: (H1 p).
```

This proof pattern of "switching the view" turns out to be so frequent that Ssreflect introduces a special *view* tactical `/` for it, which is typically combined with the standard `move` or `case` tactics. In particular, the last proof line could be replaced by the following:

```
Undo.
move/H1.
```

```
  ...
  H1 : P → Q
  H2 : R → S
  H3 : Q → R
  ============================
   Q → S
```

The assumption *H1* used for weakening is usually referred to as a *view lemma*. The spaces before and after `/` are optional. One can also *chain* the views into one series, so the current proof can be completed as follows:

```
by move/H3 /H2.
Qed.
```

5.1.1 Combining views and bookkeeping

The view tactical can be also combined with the standard bookkeeping machinery, so it will apply the specified view lemma to the corresponding assumption of the goal, as demonstrated by the following proof script, which use the partially-applied assumption $H\ p$ as a view lemma:

```
Goal ∀ P Q R, P → (P → Q → R) → Q → R.
Proof.
by move⇒P Q R p H /(H p).
```

In fact, this proof can be shortened even further by using the view notation for the *top* assumption (denoted using the underscore):

```
Undo.
move⇒ P Q R p.
by move/(_ p).
Qed.
```

The last proof script first moved four assumptions to the context, so the goal became $(P \to \mathsf{Q} \to R) \to \mathsf{Q} \to R$. Next, it partially applied the top assumption $(P \to \mathsf{Q} \to R)$ to $p : P$ from the context and moved the result back to the goal, so it became $(\mathsf{Q} \to R) \to \mathsf{Q} \to R$, which is trivially provable.

It is also possible to use views in combination with the `case` tactics, which first performs the "view switch" via the view lemma provided and then case-analysed on the result, as demonstrated by the following proof script:

```
Goal ∀ P Q R, (P → Q ∧ R) → P → R.
Proof.
move⇒ P Q R H.
by case/H.
Qed.
```

What has happened is that the combined tactic `case`/H first switched the top assumption of the goal from P to $\mathsf{Q} \wedge R$ and then case-analysed on it, which gave the proof of R right away, allowing us to conclude the proof.

5.1.2 Using views with equivalences

So far we have explored only views that help to weaken the hypothesis using the view lemma, which is an implication. In fact, Ssreflect's view mechanism is elaborate enough to deal with view lemmas defined by means of equivalence (double implication) `<->`, and the system can figure out itself, "in which direction" the view lemma should be applied. Let us demonstrate it with the following example, which makes use of the hypothesis *STequiv*,[2] whose nature is irrelevant for the illustration purposes:

```
Variables S T: bool → Prop.
Hypothesis STequiv : ∀ a b, T a ↔ S (a || b).

Lemma ST_False a b: (T a → False) → S (a || b) → False.
Proof.
```

[2]The Coq's command `Hypothesis` is a synonym for `Axiom` and `Variable`.

```
by move⇒H /STequiv.
Qed.
```

5.1.3 Declaring view hints

In the example from Section 5.1.2, we have seen how views can deal with equivalences. The mentioned elaboration, which helped the system to recognize, in which direction the double implication hypothesis *STequiv* should have been used, is not hard-coded into Ssreflect. Instead, it is provided by a flexible mechanism of *view hints*, which allows one to specify view lemmas that should be applied *implicitly* whenever it is necessary and can be figured out unambiguously.

In the case of the proof of the ST_False lemma the view hint iffRL from the included module ssreflect[3] has been "fired" in order to adapt the hypothesis *STequiv*, so the adapted variant could be applied as a view lemma to the argument of type $S\ (a \mid\mid b)$.

```
Check iffRL.
```

```
iffRL
     : ∀ P Q : Prop, (P ↔ Q) → Q → P
```

The type of iffRL reveals that what it does is simply switching the equivalence to the implication, which works right-to-left, as captured by the name. Let us now redo the proof of the ST_False lemma to see what is happening under the hood:

```
Lemma ST_False' a b: (T a → False) → S (a || b) → False.
Proof.
move⇒ H.
move/(iffRL (STequiv a b)).
done.
Qed.
```

The view switch on the second line of the proof is what has been done implicitly in the previous case: the implicit view iffRL has been applied to the call of *STequiv*, which was in its turn supplied the necessary arguments a and b, inferred by the system from the goal, so the type of (*STequiv* a b) would match the parameter type of iffRL, and the whole application would allow to make a view switch in the goal. What is left behind the scenes is the rest of the attempts made by Coq/Ssreflect in its search for a suitable implicit view, which ended when the system has finally picked iffRL.

In general, the design of powerful view hints is non-trivial, as they should capture precisely the situation when the "view switch" is absolutely necessary and the implicit views will not "fire" spuriously. In the same time, implicit view hints is what allows for the smooth implementation of the boolean reflection, as we will discuss in Section 5.3.

5.1.4 Applying view lemmas to the goal

Similarly to how they are used for *assumptions*, views can be used to interpret the goal by means of combining the Coq's standard `apply` and `exact` tactics with the view tactical `/`.

[3] Implicit view hints are defined by means of `Hint View` command, added to Coq by Ssreflect. See the implementation of the module ssrbool and Section 9.8 of the Reference Manual [23].

In the case if H is a view lemma, which is just an implication $P \to \mathsf{Q}$, where Q is the statement of the goal, the enhanced tactic `apply/` H will work exactly as the standard Ssreflect's `apply:`, that is, it will replace the goal Q with H's assumption P to prove.

However, interpreting goals via views turns out to be very beneficial in the presence of implicit view hints. For example, let us consider the following proposition to prove.

```
Definition TS_neg: ∀ a, T (negb a) → S ((negb a) || a).
Proof.
move⇒a H.
apply/STequiv.
done.
Qed.
```

The view switch on the goal via `apply/`$\mathsf{STequiv}$ immediately changed the goal from S ((negb a) || a) to T (negb a), so the rest of the proof becomes trivial. Again, notice that the system managed to infer the right arguments for the $\mathsf{STequiv}$ hypothesis by analysing the goal.

Now, if we print the body of $\mathsf{TS\_neg}$, we will be able to see how an application of the implicit application of the view lemma *iffLR* of type ∀ P Q : `Prop`, $(P \leftrightarrow \mathsf{Q}) \to P \to \mathsf{Q}$ has been inserted, allowing for the construction of the proof term:

```
Print TS_neg.
```

```
TS_neg =
  fun (a : bool) (H : T (negb a)) ⇒
  (fun F : T (negb a) ⇒
     iffLR (Q:=S (negb a || a)) (STequiv (negb a) a) F) H
     : ∀ a : bool, T (negb a) → S (negb a || a)
```

5.2 Prop versus bool

As we have already explored in the previous chapters, in CIC, the logical foundation of Coq, there is a number of important distinctions between logical propositions and boolean values. In particular, there is an infinite number of ways to represent different propositions in the sort `Prop` by means of defining the datatypes. In contrast, the type **bool** is represented just by two values: true and false. Moreover, as it was discussed in Chapter 3, in Coq only those propositions are considered to be *true*, whose proof term can be constructed. And, of course, there is no such thing as a "proof term of true", as true is simply a value.

A more interesting question, though, is for which propositions P from the sort `Prop` the proofs can be computed *automatically* by means of running a program, whose result will be an answer to the question "Whether P holds?". Therefore, such programs should always *terminate* and, upon terminating, say "true" or "false". The propositions, for which a construction of such programs (even a very inefficient one) is possible, are referred to as *decidable* ones. Alas, as it was discussed in Section 3.1 of Chapter 3, quite a lot of interesting propositions are undecidable. Such properties include the classical halting problem ("Whether the program p terminates or not?") and any higher-order formulae,

i.e., such that contain quantifiers. For instance, it is not possible to implement a higher-order function, which would take two arbitrary functions f_1 and f_2 of type **nat** → **nat** and return a boolean answer, which would indicate whether these two functions are equal (point-wise) or not, as it would amount to checking the result of the both function on each natural number, which, clearly, wouldn't terminate. Therefore, propositional equality of functions is a good example of a proposition, which is undecidable in general, so we cannot provide a terminating procedure for any values of its arguments (i.e., f_1 and f_2).

However, the *undecidability* of higher-order propositions (like the propositional equality of functions) does not make them *non-provable* for particular cases, as we have clearly observed thorough the past few chapters. It usually takes a human intuition, though, to construct a proof of an undecidable proposition by means of combining a number of hypotheses (i.e., constructing a proof term), which is what one does when building a proof using tactics in Coq. For instance, if we have some extra insight about the two functions f_1 and f_2, which are checked for equality, we might be able to construct the proof of them being equal or not, in the similar ways as we have carried the proofs so far. Again, even if the functions are unknown upfront, it does not seem possible to implement an always-terminating procedure that would automatically decide whether they are equal or not.

The above said does not mean that all possible propositions should be implemented as instances of `Prop`, making their clients to always construct their proofs, when it is necessary, since, fortunately, some propositions are *decidable*, so it is possible to construct a decision procedure for them. A good example of such proposition is a predicate, which ensures that a number n is prime. Of course, in Coq one can easily encode primality of a natural number by means of the following inductive predicate, which ensures that n is prime if it is 1 or has no other natural divisors but 1 and n itself.

```
Definition isPrime n : Prop :=
  ∀ n1 n2, n = n1 × n2 → (n1 = 1 ∧ n2 = n) ∨ (n1 = n ∧ n2 = 1).
```

Such definition, although correct, is quite inconvenient to use, as it does not provide a direct way to *check* whether some particular number (e.g., 239) is prime or not. Instead, it requires one to construct a proof of primality for *each* particular case using the constructors (or the contradiction, which would imply that the number is not prime). As it's well known, there is a terminating procedure to compute whether the number is prime or not by means of *enumerating* all potential divisors of n from 1 to the square root of n. Such procedure is actually implemented in the Ssreflect's prime module and proved correct with respect to the definition similar to the one above,[4] so now one can test the numbers by equality by simply *executing* the appropriate function and getting a boolean answer:

```
Eval compute in prime 239.

    = true
    : bool
```

Therefore, we can summarize that the *decidability* is what draws the line between propositions encoded by means of Coq's `Prop` datatypes and procedures, returning a **bool**

[4] Although the implementation and the proof are somewhat non-trivial, as they require to build a primitively-recursive function, which performs the enumeration, so we do not consider them here.

result. `Prop` provides a way to encode a *larger* class of logical statements, in particular, thanks to the fact that it allows one to use quantifiers and, therefore, encode higher-order propositions. The price to pay for the expressivity is the necessity to explicitly construct the proofs of the encoded statements, which might lead to series of tedious and repetitive scripts. **bool**-returning functions, when implemented in Coq, are decidable by construction (as Coq enforces termination), and, therefore, provide a way to compute the propositions they implement. Of course, in order to be reduced to true or false, all quantifiers should be removed by means of instantiated the corresponding bound variables, after which the computation becomes possible.

For instance, while the expression (prime 239) || (prime 42) can be evaluated to true right away, whereas the expression

$\forall\ n$, (prime n) || prime $(n + 1)$

is not even well-typed. The reason for this is that polymorphic $\forall$-quantification in Coq does not admit *values* to come after the comma (so the dependent function type "$\Pi n :$ nat, n" is malformed), similarly to how one cannot write a *type Int* $\rightarrow 3$ in Haskell, as it does not make sense. This expression can be, however, *coerced* into `Prop` by means of comparing the boolean expression with true using propositional equality

$\forall\ n$, ((prime n) || prime $(n + 1)$ = true)

which makes the whole expression to be of type `Prop`. This last example brings us to the insight that the **bool**-returning functions (i.e., decidable predicates) can be naturally *injected* into propositions of sort `Prop` by simply comparing their result with true via propositional equality, defined in Chapter 4. This is what is done by Ssreflect automatically using the implicit *coercion*, imported by the ssrbool module:

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

This coercion can be seen as an implicit type conversion, familiar from the languages like Scala or Haskell, and it inserted by Coq automatically every time it expects to see a proposition of sort `Prop`, but instead encounters a boolean value. Let us consider the following goal as an example:

```
Goal prime (16 + 14) → False.
Proof. done. Qed.
```

As we can see, the proof is rather short, and, in fact, done by Coq/Ssreflect fully automatically. In fact, the system first *computes* the value of prime $(16 + 14)$, which is, obviously false. Then the boolean value false is coerced into the propositional equality false = true, as previously described. The equality is then automatically discriminated (see Section 4.1.2), which allows the system to infer the falsehood, completing the proof.

This example and the previous discussion should convey the idea that *decidable propositions should be implemented as computable functions returning a boolean result.* This simple design pattern makes it possible to take full advantage of the computational power of Coq as a programming language and prove decidable properties automatically, rather than by means of imposing a burden of constructing an explicit proof. We have just seen how a boolean result can be easily injected back to the world of propositions. This

computational approach to proofs is what has been taken by Ssreflect to the extreme, making the proofs about common mathematical constructions to be very short, as most of the proof obligations simply *do not appear*, as the system is possible to reduce them by means of performing the computations on the fly. Even though, as discussed, some propositions can be only encoded as elements of `Prop`, our general advice is to rely on the computations whenever it is possible.

In the following subsections we will elaborate on some additional specifications and proof patterns, which are enabled by using boolean values instead of full-fledged propositions from `Prop`.

5.2.1 Using conditionals in predicates

The ternary conditional operator `if-then-else` is something that programmers use on a regular basis. However, when it comes to the specifications in the form of Coq's standard propositions it turns out one cannot simply employ the `if-then-else` connective in them, as it expects its conditional argument to be of type **bool**. This restriction is, again, a consequence of the fact that the result of `if-then-else` expression should be computable, which conflicts with the fact that not every proposition is decidable and, hence, there is no sound way overload the conditional operator, so it would rely on the existence of the proof of its conditional (or its negation).

```
Definition prime_spec_bad n m : Prop := m = (if isPrime n then 1 else 2).
```

```
Error: In environment
m : nat
n : nat
The term "isPrime n" has type "Prop" while it is expected to have type "bool".
```

Fortunately, the computable predicates are free from this problem, so on can freely use them in the conditionals:

```
Definition prime_spec n m : Prop := m = (if prime n then 1 else 2).
```

5.2.2 Case analysing on a boolean assumption

Another advantage of the boolean predicates is that they automatically come with a natural case analysis principle: reasoning about an outcome of a particular predicate, one can always consider two possibilities: when it returned true or false.[5] This makes it particularly pleasant to reason about the programs and specifications that use conditionals, which is demonstrated by the following example.

```
Definition discr_prime n := (if prime n then 0 else 1) + 1.
```

Let us now prove that the definition prime_spec gives a precise specification of the function discr_prime:

```
Lemma discr_prime_spec : ∀ n, prime_spec n (discr_prime n).
```

[5] We have already seen an instance of such case analysis in the proof of the *leqP* lemma in Section 4.3.1 of Chapter 4, although deliberately did not elaborate on it back then.

```
Proof.
move⇒n. rewrite /prime_spec /discr_prime.
```

```
n : nat
============================
 (if prime n then 0 else 1) + 1 = (if prime n then 1 else 2)
```

The proof of the specification is totally in the spirit of what one would have done when proving it manually: we just case-analyse on the value of prime n, which is either true or false. Similarly to the way the rewritings are handled by means of unification, in both cases the system substitutes prime n with its boolean value in the specification as well. The evaluation completes the proof.

```
by case: (prime n).
Qed.
```

Another common use case of boolean predicates comes from the possibility to perform a case analysis on the boolean *computable equality*, which can be employed in the proof proceeding by an argument "let us assume a to be equal to b (or not)". As already hinted by the example with the function equality earlier in this section, the computable equality is not always possible to implement. Fortunately, it can be implemented for a large class of datatypes, such as booleans, natural numbers, lists and sets (of elements with computable equality), and it was implemented in Ssreflect, so one can take an advantage of it in the proofs.[6]

5.3 The reflect type family

Being able to state all the properties of interest in a way that they are decidable is a true blessing. However, even though encoding everything in terms of **bool**-returning functions and connectives comes with the obvious benefits, reasoning in terms of Props might be more convenient when the information of the structure of the proofs matters. For instance, let us consider the following situation:

```
Variables do_check1 do_check2 : nat → bool.
Hypothesis H: ∀ n, do_check2 n → prime n.
Lemma check_prime n : (do_check1 n) && (do_check2 n) → prime n.
```

The lemma check_prime employs the boolean conjunction && from the ssrbool module in its assumption, so we know that its result is some boolean value. However simply case-analysing on its component does not bring any results. What we want indeed is a way to *decompose* the boolean conjunction into the components and then use the hypothesis *H*. This is what could be accomplished easily, had we employed the *propositional conjunction* $\wedge$ instead, as it comes with a case-analysis principle.

```
Abort.
```

This is why we need a mechanism to conveniently switch between two possible representation. Ssreflect solves this problem by employing the familiar rewriting machinery

[6] The way the computable equality is encoded so it would work uniformly for different types is an interesting topic by itself, so we postpone its explanation until Chapter 7

(see Section 4.3 of Chapter 4) and introducing the inductive predicate family **reflect**, which connects propositions and booleans:

```
Inductive reflect (P : Prop) : bool → Set :=
  | ReflectT of P : reflect P true
  | ReflectF of ~P : reflect P false.
```

Similarly to the custom rewriting rules, the **reflect** predicate is nothing but a convenient way to encode a "truth" table with respect to the predicate P, which is **reflect**'s only parameter. In other words, the propositions (**reflect** P b) ensures that (*is_true* b) and P are logically equivalent and can be replaced one by another. For instance, the following rewriting lemmas can be proved for the simple instances of `Prop`.

```
Lemma trueP : reflect True true.
Proof. by constructor. Qed.

Lemma falseP : reflect False false.
Proof. by constructor. Qed.
```

The proofs with boolean truth and falsehood can be then completed by case analysis, as with any other rewriting rules:

```
Goal false → False.
Proof. by case:falseP. Qed.
```

5.3.1 Reflecting logical connectives

The true power of the **reflect** predicate, though, is that it might be put to work with arbitrary logical connectives and user-defined predicates, therefore delivering the rewriting principles, allowing one to switch between **bool** and `Prop` (in the decidable case) by means of rewriting lemmas. Ssreflect comes with a number of such lemmas, so let us consider one of them, andP.

```
Lemma andP (b1 b2 : bool) : reflect (b1 ∧ b2) (b1 && b2).
Proof. by case b1; case b2; constructor⇒ //; case. Qed.
```

Notice that andP is stated over two boolean variables, *b1* and *b2*, which, nevertheless, are treated as instances of `Prop` in the conjunction ∧, being implicitly coerced.

We can now put this lemma to work and prove our initial example:

```
Lemma check_prime n : (do_check1 n) && (do_check2 n) → prime n.
Proof.
case: andP=>//.
```

```
  n : nat
  ============================
   do_check1 n ∧ do_check2 n → true → prime n
```

Case analysis on the rewriting rule andP generates two goals, and the second one has false as an assumption, so it is discharged immediately by using `//`. The remaining goal has a shape that we can work with, so we conclude the proof by applying the hypothesis *H* declared above.

```
by case⇒_ /H.
Qed.
```

Although the example above is a valid usage of the reflected propositions, Ssreflect leverages the rewriting with respect to boolean predicates even more by defining a number of *hint views* for the rewriting lemmas that make use of the **reflect** predicates. This allows one to use the rewriting rules (e.g., andP) in the form of *views* , which can be applied directly to an assumption or a goal, as demonstrated by the next definition.

```
Definition andb_orb b1 b2: b1 && b2 → b1 || b2.
Proof.
case/andP⇒H1 H2.
by apply/orP; left.
Qed.
```

The first line of the proof switched the top assumption from the boolean conjunction to the propositional one by means of andP used as a view. The second line applied the orP view, doing the similar switch in the goal, completing the proof by using a constructor of the propositional disjunction.

```
Print andb_orb.
```

Let us take a brief look to the obtained proof term for andb_orb.

```
andb_orb =
fun (b1 b2 : bool) (goal : b1 && b2) ⇒
(fun F : ∀ (a : b1) (b : b2),
                  (fun _ : b1 ∧ b2 ⇒ is_true (b1 || b2)) (conj a b) ⇒
 match
   elimTF (andP b1 b2) goal as a return ((fun _ : b1 ∧ b2 ⇒ is_true (b1 || b2)) a)
 with
 | conj x x0 ⇒ F x x0
 end)
  (fun (H1 : b1) (_ : b2) ⇒
   (fun F : if true then b1 ∨ b2 else ~(b1 ∨ b2) ⇒
    introTF (c:=true) orP F) (or_introl H1))
     : ∀ b1 b2 : bool, b1 && b2 → b1 || b2
```

As we can see, the calls to the rewriting lemmas andP and orP were implicitly "wrapped" into the call of hints elimTF and *introTF*, correspondingly. Defined via the conditional operator, both these view hints allowed us to avoid the second redundant goal, which we would be be forced to deal with, had we simply gone with case analysis on andP and orP as rewriting rules.

```
Check elimTF.
```

```
elimTF
     : ∀ (P : Prop) (b c : bool),
       reflect P b → b = c → if c then P else ~P
```

Exercise 5.1 (Reflecting exclusive disjunction)**.** Let us define a propositional version of the *exclusive or* predicate:

```
Definition XOR (P Q: Prop) := (P ∨ Q) ∧ ~(P ∧ Q).
```

as well as its boolean version (in a curried form, so it takes just one argument and returns a function):

```
Definition xorb b := if b then negb else fun x ⇒ x.
```

Now, prove the following *generalized* reflection lemma xorP_gen and its direct consequence, the usual reflection lemma xorP:
Hint: Recall that the *reflect* predicate is just a rewriting rule, so one can perform a case analysis on it.

```
Lemma xorP_gen (b1 b2 : bool)(P1 P2: Prop):
  reflect P1 b1 → reflect P2 b2 → reflect (XOR P1 P2) (xorb b1 b2).
Lemma xorP (b1 b2 : bool): reflect (XOR b1 b2) (xorb b1 b2).
```

Exercise 5.2 (Alternative formulation of exclusive disjunction)**.** Let us consider an alternative version of exclusive or, defined by means of the predicate XOR':

```
Definition XOR' (P Q: Prop) := (P ∧ ~Q) ∨ (~P ∧ Q).
```

Prove the following equivalence lemma between to versions of XOR:

```
Lemma XORequiv P Q: XOR P Q ↔ XOR' P Q.
```

The final step is to use the equivalence we have just proved in order to establish an alternative version of the reflective correspondence of exclusive disjunction.
Hint: Use the `Search` machinery to look for lemmas that might help to leverage the equivalence between two predicates and make the following proof to be a one-liner.

```
Lemma xorP' (b1 b2 : bool): reflect (XOR' b1 b2) (xorb b1 b2).
```

Unsurprisingly, every statement about exclusive or, e.g., its commutativity and associativity, is extremely easy to prove when it is considered as a boolean function.

```
Lemma xorbC (b1 b2: bool) : (xorb b1 b2) = (xorb b2 b1).
Proof. by case: b1; case: b2. Qed.

Lemma xorbA (b1 b2 b3: bool) : (xorb (xorb b1 b2) b3) = (xorb b1 (xorb b2 b3)).
Proof. by case: b1; case: b2; case: b3=>//. Qed.
```

It is also not difficult to prove the propositional counterparts of the above lemmas for decidable propositions, reflected by them, hence the following exercise.

Exercise 5.3. Prove the following specialized lemmas for decidable propositions represented by booleans (without using the `intuition` tactic):

```
Lemma xorCb (b1 b2: bool) : (XOR b1 b2) ↔ (XOR b2 b1).
Lemma xorAb (b1 b2 b3: bool) : (XOR (XOR b1 b2) b3) ↔ (XOR b1 (XOR b2 b3)).
```

Hint: In the proof of xorAb the generalized reflection lemma xorP_gen might come in handy.
Hint: A redundant assumption H in the context can be erased by typing `clear H` or `move ⇒ {H}`. The latter form can be combined with any bookkeeping sequence, not only with `move` tactics.

Hint: The Coq's embedded tactic `intuition` can be helpful for automatically solving goals in propositional logic.

5.3.2 Reflecting decidable equalities

Logical connectives are not the only class of inductive predicates that is worth building a **reflect**-based rewriting principle for. Another useful class of decidable propositions, which are often reflected, are equalities.

Postponing the description of a generic mechanism for declaring polymorphic decidable equalities until Chapter 7, let us see how switching between decidable **bool**-returning equality == (defined in the Ssreflect's module eqtype) and the familiar propositional equality can be beneficial.

```
Definition foo (x y: nat) := if x == y then 1 else 0.
```

The function foo naturally uses the natural numbers' boolean equality == in its body, as it is the only one that can be used in the conditional operator. The next goal, though, assumes the propositional equality of x and y, which are passed to foo as arguments.

```
Goal ∀ x y, x = y → foo x y = 1.
Proof.
move⇒x y; rewrite /foo.
```

```
  x : nat
  y : nat
  ============================
   x = y → (if x == y then 1 else 0) = 1
```

The rewriting rule/view lemma eqP, imported from eqtype allows us to switch from propositional to boolean equality, which makes the assumption to be $x == y$. Next, we combine the implicit fact that $x == y$ in the assumption of a proposition is in fact $(x == y) =$ true to perform in-place rewriting (see Section 4.2.1) by means of the -> tactical, so the rest of the proof is simply by computation.

```
by move/eqP=>->.
Qed.
```

Exercise 5.4. Sometimes, the statement "there exists unique x and y, such that $P(x, y)$ holds" is mistakingly formalized as $\exists!x\ \exists!y\ P(x, y)$. In fact, the latter assertion is much weaker than the previous one. The goal of this exercise is to demonstrate this formally.[7]

First, prove the following lemma, stating that the first assertion can be weakened from the second one.

```
Lemma ExistsUnique1 A (P : A → A → Prop):
  (∃ !x, ∃ y, P x y) →
  (∃ !x, ∃ y, P y x) →
  (∃ !x, ∃ !y, P x y).
```

The notation $\exists!\ x, P\ x$ is an abbreviation for the sigma-type, whose second component is the higher-order predicate unique, defined as follows:

[7] I am grateful to Vladimir Reshetnikov (@vreshetnikov) for making this observation on Twitter.

```
Print unique.
```

```
unique =
fun (A : Type) (P : A → Prop) (x : A) ⇒
P x ∧ (∀ x' : A, P x' → x = x')
     : ∀ A : Type, (A → Prop) → A → Prop
```

As we can see, the definition unique not just ensures that $P\ x$ holds (the left conjunct), but also that any x' satisfying P is, in fact, equal to x. As on the top level unique is merely a conjunction, it can be decomposed by `case` and proved using the `split` tactics.

Next, let us make sure that the statement in the conclusion of lemma ExistsUnique1, in fact, admits predicates, satisfied by non-uniquely defined pair (x, y). You goal is to prove that the following predicate Q, which obviously satisfied by (true, true), (false, true) and (false, false) is nevertheless a subject of the second statement.

```
Definition Q x y : Prop :=
  (x == true) && (y == true) || (x == false).
Lemma qlm : (∃ !x, ∃ !y, Q x y).
```

Hint: The following lemma eqxx, stating that the boolean equality $x == x$ always holds, might be useful for instantiating arguments for hypotheses you will get during the proof.

```
Check eqxx.
```

```
eqxx
     : ∀ (T : eqType) (x : T), x == x
```

Finally, you are invited to prove that the second statement is *strictly* weaker than the first one by proving the following lemma, which states that the reversed implication of the two statements for an arbitrary predicate P implies falsehood.

```
Lemma ExistsUnique2 :
  (∀ A (P : A → A → Prop),
   (∃ !x, ∃ !y, P x y) →
   (∃ !x, ∃ y, P x y) ∧ (∃ !x, ∃ y, P y x)) →
  False.
```

6 Inductive Reasoning in Ssreflect

In the previous chapters of this course, we have become acquainted with the main concepts of constructive logic, Coq and Ssreflect. However, the proofs we have seen so far are mostly done by case analysis, application of hypotheses and various forms of rewriting. In this chapter we will consider in more detail the proofs that employ inductive reasoning as their main component. We will see how such proofs are typically structured in Ssreflect, making the corresponding scripts very concise, yet readable and maintainable. We will also learn a few common techniques that will help to adapt an induction hypothesis to become more suitable for a goal.

In the rest of the chapter we will be constantly relying on a series of standard Ssreflect modules, such as ssrbool, ssrnat and eqtype, which we import right away.

```
From mathcomp
Require Import ssreflect ssrbool ssrnat eqtype ssrfun seq.
```

6.1 Structuring the proof scripts

An important part of the proof process is keeping to an established proof layout, which helps to maintain the proofs readable and restore the intuition driving the prover's hand. Ssreflect offers a number of syntactic primitives that help to maintain such a layout, and in this section we give a short overview of them. As usual, the Ssreflect reference manual [23] (Chapter 6) provides an exhaustive formal definition of each primitive's semantics, so we will just cover the base cases here, hoping that the subsequent proofs will provide more intuition on typical usage scenarios.

6.1.1 Bullets and terminators

The proofs proceeding by induction and case analysis typically require us to prove several goals, one by one, in a sequence picked by the system. It is considered to be a good practice to indent the subgoals (except for the last one) when there are several to prove. For instance, let us consider the following almost trivial lemma:

```
Lemma andb_true_elim b c: b && c → c = true.
```

Indeed, the reflection machinery, presented in Section 5.3 of Chapter 5, makes this proof a one liner (by case/andP.). However, for the sake of demonstration, let us not appeal to it this time and do the proof as it would be done in a traditional Coq style: by mere case analysis.

```
Proof.
case: c.
```

```
true = true

subgoal 2 (ID 15) is:
 b && false → false = true
```

Case analysis on c (which is first moved down to the goal to become an assumption) immediately gives us two subgoals to prove. Each of them can be subsequently proved by the *inner* cases analysis on b, so we do, properly indenting the goals.

```
- by case: b.
```

The proof script above successfully solves the first goal, as ensured by the *terminator* tactical `by`, which we have seen previously. More precisely, `by` *tac.* first runs the script *tac* and then applies a number of simplifications and discriminations to see if the proof can be completed. If the current goal is solved, `by` *tac.* simply terminates and proceeds to the next goal; otherwise it reports a proof script error. Alternative equivalent uses of the same principle would be *tac*; `by []`. or *tac*; `done`, which do exactly the same.

Notice that the first goal was indented and preceded by the *bullet* -. The bullet token, preceding a tactic invocation, has no operational effect on the proof and serves solely for the readability purposes. Alternative forms of tokens are `+` and `*`.

6.1.2 Using selectors and discharging subgoals

Let us restart this proof and show an alternative way to structure the proof script, which should account for multiple cases.

```
Restart.
case: c; first by [].
```

```
  b : bool
  ============================
   b && false → false = true
```

Now, right after case-analysing on c, the proof script specifies that the *first* of the generated subgoals should be solved using `by []`. In this case `first` is called *selector*, and its counterpart `last` would specify that the last goal should be taken care of instead, before proceeding.

Finally, if several simple goals can be foreseen as a result of case analysis, Coq provides a convenient way to discharge them in a structured way using the [|...|] tactical:

```
Restart.
case:c; [by [] | by case: b].
```

The script above solves the first generated goal using `by []`, and then solves the second one via `by case: b.`

6.1.3 Iteration and alternatives

Yet another possible way to prove the statement of our subject lemma is by employing Coq's *repetition* tactical `do`. The script of the form `do !`*tac.* tries to apply the tactic *tac* as many times as possible, as long as new goals are generated or no more goals are left

to prove.[1] The do-tactical can be also combined with the [|...|] tactical, so it will try to apply all of the enumerated tactics as alternatives. The following script finishes the proof of the whole lemma.

```
Restart.
by do ![done | apply: eqxx | case: b | case: c].
```

Notice that we have added two tactics into the alternative list, done and apply: eqxx, which were doomed to fail. The script, nevertheless, has succeeded, as the remaining two tactics, case: *b* and case: *c*, did all the job. Lastly, notice that the do-tactical can be specified *how many* times it should try to run each tactic from the list by using the restricted form do *n*!*tac*, where *n* is the number of attempts (similarly to iterating the rewrite tactics). The lemma above could be completed by the script of the form by do 2![...] with the same list of alternatives.

```
Qed.
```

6.2 Inductive predicates that should be functions

It has been already discussed in Chapter 5 that, even though a lot of interesting propositions are inherently undecidable and should be, therefore, represented in Coq as instances of the sort Prop, one should strive to implement as many *decidable* propositions as possible as **bool**-returning function. Such "computational" approach to the propositions turns out to pay off drastically in the long-term perspective, as most of the usual proof burden will be carried out by Coq's computational component. In this section we will browse through a series of predicates defined both as inductive datatypes and boolean functions and compare the proofs of various properties stated over the alternative representations.

One can define the fact that the only natural number which is equal to zero is the zero itself, as shown below:

```
Inductive isZero (n: nat) : Prop := IsZero of n = 0.
```

Naturally, such equality can be exploited to derived paradoxes, as the following lemma shows:

```
Lemma isZero_paradox: isZero 1 → False.
Proof. by case. Qed.
```

However, the equality on natural numbers is, decidable, so the very same definition can be rewritten as a function employing the boolean equality (==) (see Section 5.3.2 of Chapter 5), which will make the proofs of paradoxes even shorter than they already are:

```
Definition is_zero n : bool := n == 0.

Lemma is_zero_paradox: is_zero 1 → False.
Proof. done. Qed.
```

[1] Be careful, though, as such proof script might never terminate if more and more new goals will be generated after each application of the iterated tactic. That said, while Coq itself enjoys the strong normalization property (i.e., the programs in it always terminate), its tactic meta-language is genuinely Turing-complete, so the tactics, while constructing Coq programs/proofs, might never in fact terminate. Specifying the behaviour of tactics and their possible effects (including non-termination and failures) is a topic of an ongoing active research [62, 70].

That is, instead of the unavoidable case-analysis with the first `Prop`-based definition, the functional definition made Coq compute the result for us, deriving the falsehood automatically.

The benefits of the computable definitions become even more obvious when considering the next example, the predicate defining whether a natural number is even or odd. Again, we define two versions, the inductive predicate and a boolean function.

```
Inductive evenP n : Prop :=
  Even0 of n = 0 | EvenSS m of n = m.+2 & evenP m.

Fixpoint evenb n := if n is n'.+2 then evenb n' else n == 0.
```

Let us now prove a simple property: that fact that $(n + 1 + n)$ is even leads to a paradox. We first prove it for the version defined in `Prop`.

```
Lemma evenP_contra n : evenP (n + 1 + n) → False.
Proof.
elim: n=>//[| n Hn]; first by rewrite addn0 add0n; case=>//.
```

We start the proof by induction on n, which is triggered by `elim`: n.[2] The subsequent two goals (for the zero and the successor cases) are then simplified via `//` and the induction variable and hypothesis are given the names n and Hn, respectively, in the second goal (as described in Section 4.2.3). Then, the first goal (the 0-case) is discharged by simplifying the sum via two rewritings by `addn0` and `add0n` lemmas from the `ssrnat` module and case-analysis on the assumption of the form **evenP** 1, which delivers the contradiction.

The second goal is annoyingly trickier.

```
  n : nat
  Hn : evenP (n + 1 + n) → False
  ============================
   evenP (n.+1 + 1 + n.+1) → False
```

First, let us do some rewritings that make the hypothesis and the goal look alike.[3]

```
rewrite addn1 addnS addnC !addnS.
rewrite addnC addn1 addnS in Hn.
```

```
  n : nat
  Hn : evenP (n + n).+1 → False
  ============================
   evenP (n + n).+3 → False
```

Now, even though the hypothesis Hn and the goal are almost the same (modulo the natural "(.+2)-orbit" of the **evenP** predicate and some rewritings), we cannot take an advantage of it right away, and instead are required to case-analysed on the assumption of the form **evenP** $(n + n).+3$:

```
case=>// m /eqP.
```

[2] Remember that the `elim`, as most of other Ssreflect's tactics operates on the top assumption.

[3] Recall that $n.+1$ stands for the *value* of the successor of n, whereas $n + 1$ is a function call, so the whole expression in the goal cannot be just trivially simplified by Coq's computation and requires some rewritings to take the convenient form.

n : **nat**
Hn : **evenP** $(n + n).+1 \to$ **False**
m : **nat**
==========
$(n + n).+3 = m.+2 \to$ **evenP** $m \to$ **False**

Only now we can make use of the rewriting lemma to "strip off" the constant summands from the equality in the assumption, so it could be employed for brushing the goal, which would then match the hypothesis exactly.

```
by rewrite !eqSS; move/eqP=><-.
Qed.
```

Now, let us take a look at the proof of the same fact, but with the computable version of the predicate evenb.

```
Lemma evenb_contra n: evenb (n + 1 + n) → False.
Proof.
elim: n=>[|n IH] //.
```

n : **nat**
IH : evenb $(n + 1 + n) \to$ **False**
==========
evenb $(n.+1 + 1 + n.+1) \to$ **False**

In the case of zero, the proof by induction on n is automatically carried out by computation, since evenb $1 =$ false. The inductive step is not significantly more complicated, and it takes only two rewriting to get it into the shape, so the computation of evenb could finish the proof.

```
by rewrite addSn addnS.
Qed.
```

Sometimes, though, the value "orbits", which can be advantageous for the proofs involving **bool**-returning predicates, might require a bit trickier induction hypotheses than just the statement required to be proved. Let us compare the two proofs of the same fact, formulated with **evenP** and evenb.

```
Lemma evenP_plus n m : evenP n → evenP m → evenP (n + m).
Proof.
elim=>//n'; first by move=>->; rewrite add0n.
```

n : **nat**
m : **nat**
n' : **nat**
==========
$\forall\ m0$: **nat**,
$n' = m0.+2 \to$
evenP $m0 \to$ (**evenP** $m \to$ **evenP** $(m0 + m)$) $\to$ **evenP** $m \to$ **evenP** $(n' + m)$

The induction here is on the predicate **evenP**, so the first case is discharged by rewriting.

`move⇒` *m'->{n'} H1 H2 H3*; `rewrite` *addnC* !*addnS addnC*.

```
  n : nat
  m : nat
  m' : nat
  H1 : evenP m'
  H2 : evenP m → evenP (m' + m)
  H3 : evenP m
  ============================
   evenP (m' + m).+2
```

In order to proceed with the inductive case, again a few rewritings are required.

`apply`: (*EvenSS* _ _)=>//.

```
  n : nat
  m : nat
  m' : nat
  H1 : evenP m'
  H2 : evenP m → evenP (m' + m)
  H3 : evenP m
  ============================
   evenP (m' + m)
```

The proof script is continued by explicitly applying the constructor EvenSS of the **evenP** datatype. Notice the use of the wildcard underscores in the application of EvenSS. Let us check its type:

`Check` *EvenSS*.

```
EvenSS
     : ∀ n m : nat, n = m.+2 → evenP m → evenP n
```

By using the underscores, we allowed Coq to *infer* the two necessary arguments for the EvenSS constructor, namely, the values of n and m. The system was able to do it basing on the goal, which was reduced by applying it. After the simplification and automatic discharging the of the trivial subgoals (e.g., $(m' + m).+2 = (m' + m).+2$) via the `//` tactical, the only left obligation can be proved by applying the hypothesis *H2*.

`by apply`: *H2*.

`Qed`.

In this particular case, the resulting proof was quite straightforward, thanks to the explicit equality $n = m.+2$ in the definition of the EvenSS constructor.

In the case of the boolean specification, though, the induction should be done on the natural argument itself, which makes the first attempt of the proof to be not entirely trivial.

```
Lemma evenb_plus n m : evenb n → evenb m → evenb (n + m).
Proof.
elim: n=>[|n Hn]; first by rewrite add0n.
```

```
  m : nat
  n : nat
  Hn : evenb n → evenb m → evenb (n + m)
  ============================
   evenb n.+1 → evenb m → evenb (n.+1 + m)
```

The problem now is that, if we keep building the proof by induction on n or m, the induction hypothesis and the goal will be always "mismatched" by one, which will prevent us finishing the proof using the hypothesis.

There are multiple ways to escape this vicious circle, and one of them is to *generalize* the induction hypothesis. To do so, let us restart the proof.

```
Restart.
move: (leqnn n).
```

```
  n : nat
  m : nat
  ============================
   n ≤ n → evenb n → evenb m → evenb (n + m)
```

Now, we are going to proceed with the proof by *selective* induction on n, such that some of its occurrences in the goal will be a subject of inductive reasoning (namely, the second one), and some others will be left generalized (that is, bound by a forall-quantified variable). We do so by using Ssreflect's tactics `elim` with explicit *occurrence selectors*.

```
elim: n {-2}n.
```

```
  m : nat
  ============================
   ∀ n : nat, n ≤ 0 → evenb n → evenb m → evenb (n + m)

subgoal 2 (ID 860) is:
 ∀ n : nat,
 (∀ n0 : nat, n0 ≤ n → evenb n0 → evenb m → evenb (n0 + m)) →
 ∀ n0 : nat, n0 ≤ n.+1 → evenb n0 → evenb m → evenb (n0 + m)
```

The same effect could be achieved by using `elim: n {1 3 4}n`, that is, indicating which occurrences of n *should* be generalized, instead of specifying, which ones should not (as we did by means of $\{\text{-}2\}n$).

Now, the first goal can be solved by case-analysis on the top assumption (that is, n).

```
- by case=>//.
```

For the second goal, we first move some of the assumptions to the context.

```
move⇒n Hn.
```

```
  m : nat
  n : nat
  Hn : ∀ n0 : nat, n0 ≤ n → evenb n0 → evenb m → evenb (n0 + m)
  ============================
   ∀ n0 : nat, n0 ≤ n.+1 → evenb n0 → evenb m → evenb (n0 + m)
```

We then perform the case-analysis on *n0* in the goal, which results in two goals, one of which is automatically discharged.

```
case=>//.
```

```
  m : nat
  n : nat
  Hn : ∀ n0 : nat, n0 ≤ n → evenb n0 → evenb m → evenb (n0 + m)
  ============================
   ∀ n0 : nat, n0 < n.+1 → evenb n0.+1 → evenb m → evenb (n0.+1 + m)
```

Doing *one more* case analysis will add one more 1 to the induction variable *n0*, which will bring us to the desired (.+2)-orbit.

```
case=>// n0.
```

```
  m : nat
  n : nat
  Hn : ∀ n0 : nat, n0 ≤ n → evenb n0 → evenb m → evenb (n0 + m)
  n0 : nat
  ============================
   n0.+1 < n.+1 → evenb n0.+2 → evenb m → evenb (n0.+2 + m)
```

The only thing left to do is to tweak the top assumption (by relaxing the inequality via the ltnW lemma), so we could apply the induction hypothesis *Hn*.

```
by move/ltnW /Hn=>//.
Qed.
```

It is fair to notice that this proof was far less direct that one could expect, but it taught us an important trick—selective generalization of the induction hypothesis. In particular, by introducing an extra assumption $n \leq n$ in the beginning, we later exploited it, so we could apply the induction hypothesis, which was otherwise general enough to match the ultimate goal at the last step of the proof.

6.2.1 Eliminating assumptions with a custom induction hypothesis

The functions like evenb, with specific value orbits, are not particularly uncommon, and it is useful to understand the key induction principles to reason about them. In particular, the above discussed proof could have been much more straightforward if we first proved a different induction principle nat2_ind for natural numbers.

```
Lemma nat2_ind (P: nat → Prop):
```

```
  P 0 → P 1 → (∀ n, P n → P (n.+2)) → ∀ n, P n.
Proof.
move⇒ H0 H1 H n.
```

```
  P : nat → Prop
  H0 : P 0
  H1 : P 1
  H : ∀ n : nat, P n → P n.+2
  n : nat
  ============================
   P n
```

Unsurprisingly, the proof of this induction principle follows the same pattern as the proof of evenb_plus—generalizing the hypothesis. In this particular case, we generalize it in the way that it would provide an "impedance matcher" between the 1-step "default" induction principle on natural numbers and the 2-step induction in the hypothesis H. We show that for the proof it is sufficient to establish $(P\ n \wedge P\ (n.+1))$:

```
suff: (P n ∧ P (n.+1)) by case.
```

The rest of the proof proceeds by the standard induction on n.

```
by elim: n=>//n; case⇒ H2 H3; split=>//; last by apply: H.
Qed.
```

Now, since the new induction principle nat2_ind exactly matches the 2-orbit, we can directly employ it for the proof of the previous result.

```
Lemma evenb_plus' n m : evenb n → evenb m → evenb (n + m).
Proof.
by elim/nat2_ind : n.
Qed.
```

Notice that we used the version of the `elim` tactics with specific *elimination view* nat2_ind, different from the default one, which is possible using the view tactical `/`. In this sense, the "standard induction" `elim: n` would be equivalent to `elim/nat_ind: n`.

Exercise 6.1. Let us define the binary division function div2 as follows.

```
Fixpoint div2 (n: nat) := if n is p.+2 then (div2 p).+1 else 0.
```

Prove the following lemma directly, *without* using the nat2_ind induction principle.

```
Lemma div2_le n: div2 n ≤ n.
```

6.3 Inductive predicates that are hard to avoid

Although formulating predicates as boolean functions is often preferable, it is not always trivial to do so. Sometimes, it is (seemingly) much simpler to come up with an inductive predicate, which explicitly witnesses the property of interest. As an example for such property, let us consider the notion of *beautiful* and *gorgeous* numbers, which we borrow from Pierce et al.'s electronic book [53] (Chapter MoreInd).

```
Inductive beautiful (n: nat) : Prop :=
| b_0 of n = 0
| b_3 of n = 3
| b_5 of n = 5
| b_sum n' m' of beautiful n' & beautiful m' & n = n' + m'.
```

The number is beautiful if it's either 0, 3, 5 or a sum of two beautiful numbers. Indeed, there are many ways to decompose some numbers into the sum $3 \times n + 5 \times n$.[4] Encoding a function, which checks whether a number is beautiful or not, although not impossible, is not entirely trivial (and, in particular, it's not trivial to prove the correctness of such function with respect to the definition above). Therefore, if one decides to stick with the predicate definition, some operations become tedious, as, even for constants the property should be *inferred* rather than proved:

```
Theorem eight_is_beautiful: beautiful 8.
Proof.
apply: (b_sum _ 3 5)=>//; first by apply: b_3.
by apply b_5.
Qed.

Theorem b_times2 n: beautiful n → beautiful (2 × n).
Proof.
by move⇒H; apply: (b_sum _ n n)=>//; rewrite mul2n addnn.
Qed.
```

In particular, the negation proofs become much less straightforward than one would expect:

```
Lemma one_not_beautiful n: n = 1 → ~beautiful n.
Proof.
move⇒E H.
```

```
  n : nat
  E : n = 1
  H : beautiful n
  ============================
   False
```

The way to infer the falsehood will be to proceed by induction on the hypothesis H:

```
elim: H E⇒n'; do?[by move=>->].
move⇒ n1 m' _ H2 _ H4 → {n' n}.
```

Notice how the assumptions n' and n are removed from the context (since we don't need them any more) by enumerating them using $\{n'\ n\}$ notation.

```
case: n1 H2=>// n'⇒ H3.
by case: n' H3=>//; case.
Qed.
```

Exercise 6.2. Prove the following theorem about beautiful numbers.

[4] In fact, the solution of this simple Diophantine equation are all natural numbers, greater than 7.

`Lemma` *b_timesm n m*: *beautiful n* → *beautiful* (*m* × *n*).

Hint: Choose wisely, what to build the induction on.

To practice with proofs by induction, let us consider yet another inductive predicate, borrowed from Pierce et al.'s course and defining which of natural numbers are *gorgeous*.

```
Inductive gorgeous (n: nat) : Prop :=
| g_0 of n = 0
| g_plus3 m of gorgeous m & n = m + 3
| g_plus5 m of gorgeous m & n = m + 5.
```

Exercise 6.3. Prove by induction the following statements about gorgeous numbers:

`Lemma` *gorgeous_plus13 n*: *gorgeous n* → *gorgeous* (*n* + 13).

`Lemma` *beautiful_gorgeous* (*n*: *nat*) : *beautiful n* → *gorgeous n*.

`Lemma` *g_times2* (*n*: *nat*): *gorgeous n* → *gorgeous* (*n* × 2).

`Lemma` *gorgeous_beautiful* (*n*: *nat*) : *gorgeous n* → *beautiful n*.

As usual, do not hesitate to use the `Search` utility for finding the necessary rewriting lemmas from the ssrnat module.

Exercise 6.4 (Gorgeous reflection)**.** Gorgeous and beautiful numbers, defining, in fact, exactly the same subset of **nat** are a particular case of Frobenius coin problem, which asks for the largest integer amount of money, that cannot be obtained using only coins of specified denominations.[5] In the case of **beautiful** and **gorgeous** numbers we have two denominations available, namely 3 and 5. An explicit formula exists for the case of only two denominations n_1 and n_2, which allows one to compute the Frobenius number as $g(n_1, n_2) = n_1 \times n_2 - n_1 - n_2$. That said, for the case $n_1 = 3$ and $n_2 = 5$ the Frobenius number is 7, which means that all numbers greater or equal than 8 are in fact beautiful and gorgeous (since the two are equivalent, as was established by Exercise 6.3).

In this exercise, we suggest the reader to prove that the efficient procedure of "checking" for gorgeousness is in fact correct. First, let us defined the following candidate function.

```
Fixpoint gorgeous_b n : bool := match n with
 | 1 | 2 | 4 | 7 ⇒ false
 | _ ⇒ true
 end.
```

The ultimate goal of this exercise is to prove the statement reflect (**gorgeous** *n*) (gorgeous_b *n*), which would mean that the two representations are equivalent. Let us divide the proof into two stages:

- The first stage is proving that all numbers greater or equal than 8 are gorgeous. To prove this it might be useful to have the following two facts established:

Hint: Use the tactic `constructor` *i* to prove a goal, which is an *n*-ary disjunction, which is satisfied if its *i*th disjunct is true.

[5] http://en.wikipedia.org/wiki/Frobenius_problem

```
Lemma repr3 n : n ≥ 8 →
  ∃ k, [\/ n = 3 × k + 8, n = 3 × k + 9 | n = 3 × k + 10].
Lemma gorg3 n : gorgeous (3 × n).
```

Next, we can establish by induction the following criteria using the lemmas repr3 and gorg3 in the subgoals of the proof.

```
Lemma gorg_criteria n : n ≥ 8 → gorgeous n.
```

This makes the proof of the following lemma trivial.

```
Lemma gorg_refl' n: n ≥ 8 → reflect (gorgeous n) true.
```

- In the second stage of the proof of reflection, we will need to prove four totally boring but unavoidable lemmas.

Hint: The rewriting lemmas addnC and eqSS from the ssrnat module might be particularly useful here.

```
Lemma not_g1: ~(gorgeous 1).
Lemma not_g2: ~(gorgeous 2).
Lemma not_g4: ~(gorgeous 4).
Lemma not_g7: ~(gorgeous 7).
```

We can finally provide prove the ultimate reflection predicate, relating **gorgeous** and gorgeous_b.

```
Lemma gorg_refl n : reflect (gorgeous n) (gorgeous_b n).
```

Exercise 6.5 (Complete trees)**.** In this exercise, we will consider a binary tree datatype and several functions on such trees.

```
Inductive tree : Set :=
| leaf
| node of tree & tree.
```

A tree is either a leaf or a node with two branches. The height of a leaf is zero, and height of a node is the maximum height of its branches plus one.

```
Fixpoint height t :=
if t is node t1 t2 then (maxn (height t1) (height t2)).+1 else 0.
```

The number of leaves in a node is the total number of leaves in both its branches.

```
Fixpoint leaves t :=
if t is node t1 t2 then leaves t1 + leaves t2 else 1.
```

A node is deemed a *complete* tree if both its branches are complete and have the same height; a leaf is considered a complete tree.

```
Fixpoint complete t :=
if t is node t1 t2 then complete t1 && complete t2 && (height t1 == height t2)
else true.
```

We can now prove by induction that in a complete tree, the number of leaves is two to the power of the tree's height.

```
Theorem complete_leaves_height t : complete t → leaves t = 2 ^ height t.
```

6.4 Working with Ssreflect libraries

As it was mentioned in Chapter 1, Ssreflect extension to Coq comes with an impressive number of libraries for reasoning about the large collection of discrete datatypes and structures, including but not limited to booleans, natural numbers, sequences, finite functions and sets, graphs, algebras, matrices, permutations etc. As discussed in this and previous chapters, all these libraries give preference to the computable functions rather than inductive predicates and leverage the reasoning via rewriting by equality. They also introduce a lot of notations that are worth being re-used in order to make the proof scripts tractable, yet concise.

We would like to conclude this chapter with a short overview of a subset of the standard Ssreflect programming and naming policies, which will, hopefully, simplify the use of the libraries in a standalone development.

6.4.1 Notation and standard properties of algebraic operations

Ssreflect's module ssrbool introduces convenient notation for predicate connectives, such as $\wedge$ and $\vee$. In particular, multiple conjunctions and disjunctions are better to be written as [$\wedge$ *P1*, *P2* & *P3*] and [$\vee$ *P1*, *P2* | *P3*], respectively, opposed to *P1* $\wedge$ *P2* $\wedge$ *P3* and *P1* $\vee$ *P2* $\vee$ *P3*. The specific notation makes it more convenient to use such connectives in the proofs that proceed by case analysis. Compare.

```
Lemma conj4 P1 P2 P3 P4 : P1 ∧ P2 ∧ P3 ∧ P4 → P3.
Proof. by case⇒p1 [p2][p3]. Qed.

Lemma conj4' P1 P2 P3 P4 : [ ∧ P1, P2, P3 & P4] → P3.
Proof. by case. Qed.
```

In the first case, we had progressively decompose binary right-associated conjunctions, which was done by means of the *product naming* pattern [...],[6] so eventually all levels were "peeled off", and we got the necessary hypothesis *p3*. In the second formulation, conj4', the case analysis immediately decomposed the whole 4-conjunction into the separate assumptions.

For functions of arity bigger than one, Ssreflect's module ssrfun also introduces convenient notation, allowing them to be curried with respect to the second argument:

```
Locate "_ ^~ _".
```

```
"f ^~ y" := fun x ⇒ f x y : fun_scope
```

For instance, this is how one can now express the partially applied function, which applies its argument to the list [:: 1; 2; 3]:

```
Check map ^~ [:: 1; 2; 3].
```

```
map^~ [:: 1; 2; 3]
     : (nat → ?2919) → seq ?2919
```

[6] The same introduction pattern works in fact for *any* product type with one constructor, e.g., the existential quantification (see Chapter 3).

Finally, ssrfun defines a number of standard operator properties, such as commutativity, distributivity etc in the form of the correspondingly defined predicates: *commutative*, *right_inverse* etc. For example, since we have now ssrbool and ssrnat imported, we can search for left-distributive operations defined in those two modules (such that they come with the proofs of the corresponding predicates):

```
Search _ (left_distributive _).
```

```
andb_orl left_distributive andb orb
orb_andl left_distributive orb andb
andb_addl left_distributive andb addb
addn_maxl left_distributive addn maxn
addn_minl left_distributive addn minn
...
```

A number of such properties is usually defined in a generic way, using Coq's canonical structures, which is a topic of Chapter 7.

6.4.2 A library for lists

Lists, being one of the most basic inductive datatypes, are usually a subject of a lot of exercises for the fresh Coq hackers. Ssreflect's modules seq collect a number of the most commonly used procedures on lists and their properties, as well as some non-standard induction principles, drastically simplifying the reasoning.

For instance, properties of some of the functions, such as *list reversal* are simpler to prove not by the standard "direct" induction on the list structure, but rather iterating the list from its last element, for which the seq library provides the necessary definition and induction principle:

```
Fixpoint rcons s z := if s is x :: s' then x :: rcons s' z else [:: z].
```

```
Check last_ind.
```

```
last_ind
     : ∀ (T : Type) (P : seq T → Type),
       P [::] →
       (∀ (s : seq T) (x : T), P s → P (rcons s x)) →
       ∀ s : seq T, P s
```

That is, last_ind is very similar to the standard list induction principle *list_ind*, except for the fact that its "induction step" is defined with respect to the *rcons* function, rather than the list's constructor *cons*. We encourage the reader to check the proof of the list function properties, such as *nth_rev* or *foldl_rev* to see the reasoning by the last_ind induction principle.

To demonstrate the power of the library for reasoning with lists, let us prove the following property, known as *Dirichlet's box principle* (sometimes also referred to as *pigeonhole principle*), the formulation of which we have borrowed from Chapter MoreLogic of Pierce et al.'s course [53].

```
Variable A : eqType.

Fixpoint has_repeats (xs : seq A) :=
  if xs is x :: xs' then (x \in xs') || has_repeats xs' else false.
```

The property has_repeats is stated over the lists with elements that have decidable equality, which we have considered in Section 5.3.2. Following the computational approach, it is a boolean function, which makes use of the boolean disjunction || and Ssreflect's element inclusion predicate \in, which is defined in the module seq.

The following lemma states that for two lists *xs1* and *xs2*, is the size *xs2* is strictly smaller than the size of *xs1*, but nevertheless *xs1* as a set is a subset of *xs2* then there ought to be repetitions in *xs1*.

```
Theorem dirichlet xs1 xs2 :
        size xs2 < size xs1 → {subset xs1 ≤ xs2} → has_repeats xs1.
```

Let us go through the proof of this statement, as it is interesting by itself in its intensive use of Ssreflect's library lemmas from the seq module.

```
Proof.
```

First, the proof scripts initiates the induction on the structure of the first, "longer", list *xs1*, simplifying and moving to the context some hypotheses in the "step" case (as the *nil*-case is proved automatically).

```
elim: xs1 xs2=>[|x xs1 IH] xs2 //= H1 H2.
```

```
  x : A
  xs1 : seq A
  IH : ∀ xs2 : seq A,
       size xs2 < size xs1 → {subset xs1 ≤ xs2} → has_repeats xs1
  xs2 : seq A
  H1 : size xs2 < (size xs1).+1
  H2 : {subset x :: xs1 ≤ xs2}
  ============================
   (x \in xs1) || has_repeats xs1
```

Next, exactly in the case of a paper-and-pencil proof, we perform the case-analysis on the fact $(x \in xs1)$, i.e., whether the "head" element x occurs in the remainder of the list *xs1*. If it is, the proof is trivial and automatically discharged.

```
case H3: (x \in xs1) ⇒ //=.
```

```
  ...
  H3 : (x \in xs1) = false
  ============================
   has_repeats xs1
```

Therefore, we are considering now the situation when x was the *only* representative of its class in the original "long" list. For the further inductive reasoning, we will have to remove the same element from the "shorter" list *xs2*, which is done using the following filtering operation (pred1 x checks every element for equality to x and predC constructs

the negation of the passed predicate), resulting in the list *xs2'*, to which the induction hypothesis is applied, resulting in two goals

```
pose xs2' := filter (predC (pred1 x)) xs2.
apply: (IH xs2'); last first.
```

```
  ...
  H2 : {subset x :: xs1 ≤ xs2}
  H3 : (x \in xs1) = false
  xs2' := [seq x ← xs2 | (predC (pred1 x)) x0] : seq A
  ============================
   {subset xs1 ≤ xs2'}

subgoal 2 (ID 5716) is:
 size xs2' < size xs1
```

The first goal is discharged by first "de-sugaring" the {*subset* ...} notation and moving a universally-quantified variable to the top, and then performing a number of rewriting with the lemmas from the seq library, such as inE and mem_filter (check their types!).

```
- move⇒y H4; move: (H2 y); rewrite inE H4 orbT mem_filter /=.
  by move ⇒ → //; case: eqP H3 H4 ⇒ // ->->.
```

The second goal requires to prove the inequality, which states that after removal of x from *xs2*, the length of the resulting list *xs2* is smaller than the length of *xs1*. This is accomplished by the transitivity of $<$ and several rewritings by lemmas from the seq and ssrnat modules, mostly targeted to relate the filter function and the size of the resulting list.

```
rewrite ltnS in H1; apply: leq_trans H1.
rewrite -(count_predC (pred1 x) xs2) -addn1 addnC.
rewrite /xs2' size_filter leq_add2r -has_count.
```

```
  ...
  H2 : {subset x :: xs1 ≤ xs2}
  H3 : (x \in xs1) = false
  xs2' := [seq x ← xs2 | (predC (pred1 x)) x0] : seq A
  ============================
   has (pred1 x) xs2
```

The remaining goal can be proved by *reflecting* the boolean proposition *has* into its Prop-counterpart *exists2* from Ssreflect library. The switch is done using the view hasP, and the proof is completed by supplying explicitly the existential witness x.

```
by apply/hasP; ∃ x=>//=; apply: H2; rewrite inE eq_refl.
Qed.
```

7 Encoding Mathematical Structures

Long before programming has been established as a discipline, mathematics came to be perceived as a science of building abstractions and summarizing important properties of various entities necessary for describing nature's phenomenons.[1] From the basic course of algebra, we are familiar with a number of mathematical structures, such as monoids, groups, rings, fields etc., which couple a *carrier* set (or a number of sets), a number of operations on it (them), and a collection of properties of the set itself and operations on them.

From a working programmer's perspective, a notion of a mathematical abstract structure is reminiscent to a notion of a class from object-oriented programming, modules from Standard ML and type classes [64] from Haskell: all these mechanisms are targeted to solve the same goal: *package* a number of operations manipulating with some data, while abstracting of a particular implementation of this data itself. What neither of these programming mechanisms is capable of doing, comparing to mathematics, is enforcing the requirement for one to provide the *proofs* of properties, which restrict the operations on the data structure. For instance, one can implement a type class for a *lattice* in Haskell as follows:

```
class Lattice a where
  bot :: a
  top :: a
  pre :: a -> a -> Bool
  lub :: a -> a -> a
  glb :: a -> a -> a
```

That is, the class `Lattice` is parametrized by a *carrier* type `a`, and provides the abstract interfaces for top and bottom elements of the lattice, as well as for the ordering predicate `pre` and the binary *least-upper-bound* and *greatest-lower-bound* operations. What this class cannot capture is a number of restrictions, for instance, that the `pre` relation should be transitive, reflexive and antisymmetric. That said, one can instantiate the `Lattice` class, e.g., for integers, providing an implementation of `pre`, which is *not* a partial order (e.g., just constant `true`). While this relaxed approach is supposedly fine for the programming needs, as the type classes are used solely for computing, not the reasoning about the correctness of the computations, this is certainly unsatisfactory from the mathematical perspective. Without the possibility to establish and enforce the necessary properties of a mathematical structure's operations, we would not be able to carry out any sort of sound formal reasoning, as we simply could not distinguish a "correct" implementation from a flawed one.

Luckily, Coq's ability to work with dependent types and combine programs and propositions about them in the same language, as we've already witnessed in the previous

[1] In addition to being a science of rewriting, as we have already covered in Chapter 4.

chapters, makes it possible to define mathematical structures with a necessary degree of rigour and describe their properties precisely by means of stating them as *types* (i.e., propositions) of the appropriate implementation's parameters. Therefore, any faithful instance of an abstract mathematical structure implemented this way, would be enforced to provide not just the *carrier* and implementations of the declared operations but also *proofs* of propositions that constrain these operations and the carrier.

In this chapter we will learn how to encode common algebraic data structures in Coq in a way very similar to how data structures are encoded in languages like C (with a bit of Haskell-ish type class-like machinery), so the representation, unlike the one in C or Haskell, would allow for flexible and generic reasoning about the structures' properties. In the process, we will meet some old friends from the course of abstract algebra—partial commutative monoids, and implement them using Coq's native constructs: dependent records and canonical structures.

As usual, we will require a number of Ssreflect package imported.

```
From mathcomp
Require Import ssreflect ssrbool ssrnat eqtype ssrfun.
```

We will also require to execute a number of Vernacular commands simplifying the handling of implicit datatype arguments.

```
Set Implicit Arguments.
Unset Strict Implicit.
Unset Printing Implicit Defensive.
```

7.1 Encoding partial commutative monoids

We will be using partial commutative monoids (PCMs) as an illustrative example of a simple algebraic data structure, a subject of encoding and formalization. A PCM is defined as an algebraic structure with a carrier set U, abstract binary "join" operation $\bullet$ and a unit element $\mathbb{1}$.[2] The join operation is required to be associative and commutative, and for the unit element the left and right identity equalities should hold. Moreover, partiality means that the operation $\bullet$ might be undefined for some pairs of elements a and b (and in this case it is denoted as $a \bullet b = \bot$). PCMs are fairly ubiquitous: in particular, natural numbers with addition and multiplication, sets with a disjoin union, partially-defined functions with a point-wise union, are all PCM instances. Furthermore, partial commutative monoids are omnipresent in program verification [43], as they capture exactly the properties of *heaps*, as well as the effect of programs that can be executed in parallel [44]. Therefore, it is useful to have PCMs formalized as a structure, so they could be employed for future reasoning.

[2] Sometimes also referred to as an *identity* or *neutral* element.

7.1.1 Describing algebraic data structures via dependent records

`Module` *PCMDef*.

In Section 3.6 of Chapter 3 we have already seen a use of a dependent pair type, exemplified by the Coq's definition of the universal quantification.

`Print` *ex*.

```
Inductive ex (A : Type) (P : A → Prop) : Prop :=
    ex_intro : ∀ x : A, P x → ex P
```

The only constructor *ex_intro* of the predicate **ex**, whose type is a dependent function type, is a way to encode a Σ-type of a dependent pair, such that its second component's type *depends* on the value of the first one. More specifically, the result of the existential quantification's encoding in Coq is a dependent pair $(\Sigma x : A, P\ x)$, such that the proposition in the second component is determined by the value of the first component x.

Coq provides an alternative way to encode *iterated* dependent pairs via the mechanism of *dependent records*, also allowing one to give names to the subsequent components. Dependent records are defined using the `Record` command. Getting back to our PCM example, we illustrate the use of dependent records by the following definition of the abstract PCM structure.

```
Record mixin_of (T : Type) := Mixin {
    valid_op : T → bool;
    join_op : T → T → T;
    unit_op : T;
    _ : commutative join_op;
    _ : associative join_op;
    _ : left_id unit_op join_op;
    _ : ∀ x y, valid_op (join_op x y) → valid_op x;
    _ : valid_op unit_op
}.
```

```
mixin_of is defined
mixin_of_rect is defined
mixin_of_ind is defined
mixin_of_rec is defined
valid_op is defined
join_op is defined
unit_op is defined
```

The syntax of Coq's dependent records is reminiscent to the one of records in C. Following Ssreflect's naming pattern [20], we call the record type (defined in a dedicated module for the reasons explained further) **mixin_of** and its only constructor Mixin. The reasons for such naming convention will be explained soon, and for now let us discuss the definition. The PCM record type is parametrized over the carrier type T, which determines the carrier set of a PCM. It then lists three *named* fields: join_op describes an implementation of the PCM's binary operation, unit_op defines the unit element, finally,

the valid_op predicate determines whether a particular element of the carrier set T is valid or not, and, thus, serves as a way to express the partiality of the join_op operation (the result is undefined, whenever the corresponding value of T is non-valid). Next, the PCM record lists five unnamed PCM *properties*, which should be satisfied whenever the record is instantiated and are defined using the standard propositions from Ssreflect's ssrfun module (see Section 6.4.1). In particular, the PCM type definition requires the operation to be commutative and associative. It also states that if $a \bullet b \neq \bot$ then $a \neq \bot$ (the same statement about b can be proved by commutativity), and that the unit element is a valid one.

Notice that in the definition of the **mixin_of** record type, the types of some of the later fields (e.g., any of the properties) depend on the values of fields declared earlier (e.g., unit_op and join_op), which makes **mixin_of** to be a truly dependent type.

Upon describing the record, a number of auxiliary definitions have been generated by Coq automatically. Along with the usual recursion and induction principles, the system also generated three *getters*, valid_op, join_op and unit_op for the record's named fields. That is, similarly to Haskell's syntax, given an instance of a PCM, one can extract, for example, its operation, via the following getter function.

```
Check valid_op.
```

```
valid_op
     : ∀ T : Type, mixin_of T → T → bool
```

Coq supports the syntax for anonymous record fields (via the underscore _), so getters for them are not generated. We have decided to make the property fields of **mixin_of** to be anonymous, since they will usually appear only in the proofs, where the structure is going to be decomposed by case analysis anyway, as we will soon see.

We can now prove a number of facts about the structure, very much in the spirit of the facts that are being proven in the algebra course. For instance, the following lemma states that unit_op is also the *right unit*, in addition to it being the left unit, as encoded by the structure's definition.

```
Lemma r_unit T (pcm: mixin_of T) (t: T) : (join_op pcm t (unit_op pcm)) = t.
Proof.
case: pcm⇒_ join unit Hc _ Hlu _ _ /=.
```

```
  T : Type
  t : T
  join : T → T → T
  unit : T
  Hc : commutative join
  Hlu : left_id unit join
  ============================
   join t unit = t
```

The first line of the proof demonstrates that dependent records in Coq are actually just product types in disguise, so the proofs about them should be done by case analysis. In this particular case, we decompose the pcm argument of the lemma into its components,

replacing those of no interest with wildcards _. The join and unit, therefore, bind the operation and the identity element, whereas *Hc* and *Hlu* are the commutativity and left-unit properties, named explicitly in the scope of the proof. The trailing Ssreflect's simplification tactical `/=` replaces the calls to the getters in the goal (e.g., join_op pcm) by the bound identifiers. The proof can be now accomplished by a series of rewritings by the *Hc* and *Hlu* hypotheses.

```
by rewrite Hc Hlu.
Qed.
```

7.1.2 An alternative definition

In the previous section we have seen how to define the algebraic structure of PCMs using Coq's dependent record mechanism. The same PCM structure could be alternatively defined using the familiar syntax for inductive types, as a datatype with precisely one constructor:

```
Inductive mixin_of' (T: Type) :=
  Mixin' (valid_op: T → bool) (join_op : T → T → T) (unit_op: T) of
    commutative join_op &
    associative join_op &
    left_id unit_op join_op &
    ∀ x y, valid_op (join_op x y) → valid_op x &
    valid_op unit_op.
```

Although this definition seems more principled and is closer to what we have seen in previous chapters, the record notation is more convenient in this case, as it defined getters automatically as well as allows one to express inheritance between data structures by means of the coercion operator `:>` operator [20].[3]

7.1.3 Packaging the structure from mixins

```
Section Packing.
```

By now, we have defined a structure of a PCM "interface" in a form of a set of the components (i.e., the carrier set and operations on it) and their properties. However, it might be the case that the same carrier set (which we represented by the type parameter T), should be given properties from other algebraic data structures (e.g., lattices), which are essentially orthogonal to those of a PCM. Moreover, at some point one might be interested in implementing the proper inheritance of the PCM structure with respect to the carrier type T. More precisely, if the type T comes with some additional operations, they should be available from it, even if it's seen as being "wrapped" into the PCM structure. That said, if T is proven to be a PCM, one should be able to use this fact as well as the functions, defined on T separately.

These two problems, namely, (a) combining together several structures into one, and (b) implementing inheritance and proper mix-in composition, can be done in Coq using the description pattern, known as "packed classes" [20]. The idea of the approach is to define

[3] In the next section will show a different way to encode implicit inheritance, though.

a "wrapper" record type, which would "pack" several mix-ins together, similar to how it is done in object-oriented languages with implicit trait composition, e.g., Scala [47].[4]

```
Structure pack_type : Type := Pack {type : Type; _ : mixin_of type}.
```

The dependent data structure **pack_type** declares two fields: the field `type` of type `Type`, which described the carrier type of the PCM instance, and the actual PCM structure (without an explicit name given) of type (**mixin_of** `type`). That is, in order to construct an instance of **pack_type**, one will have to provide *both* arguments: the carrier set and a PCM structure for it.

Next, we specify that the field `type` of the **pack_type** should be also considered as a *coercion*, that is, whenever we have a value of type **pack_type**, whose field `type` is some T, it can be implicitly seen as an element of type T. The coercion is specified locally, so it will work only in the scope of the current section (i.e., Packing) by using Coq's `Local Coercion` command. We address the reader to Chapter 18 of the Coq Reference Manual [10] for more details of the implicit coercions.

```
Local Coercion type : pack_type >-> Sortclass.
```

The `>->` simply specifies the fact of the coercion, whereas *Sortclass* is an abstract class of sorts, so the whole command postulates that whenever an instance of **pack_type** should be coerced into an element of an arbitrary sort, it should be done via referring to is `type` field.

Next, in the same section, we provide a number of abbreviations to simplify the work with the PCM packed structure and prepare it to be exported by clients.

```
Variable cT: pack_type.

Definition pcm_struct : mixin_of cT :=
    let: Pack _ c := cT return mixin_of cT in c.
```

The function pcm_struct extracts the PCM structure from the "packed" instance. Notice the use of dependent pattern matching in the `let`:-statement with the explicit `return`-statement, so Coq would be able to refine the result of the whole expression basing on the dependent type of the c component of the data structure cT, which is being scrutinized. With the help of this definition, we can now define three aliases for the PCM's key components, "lifted" to the packed data structure.

```
Definition valid := valid_op pcm_struct.
Definition join := join_op pcm_struct.
Definition unit := unit_op pcm_struct.

End Packing.
```

Now, as the packaging mechanism and the aliases are properly defined, we come to the last step of the PCM package description: preparing the batch of definitions, notations and facts to be exported to the client. Following the pattern of nesting modules, presented in Section 2.6, we put all the entities to be exported into the inner module *Exports*.

```
Module Exports.

Notation pcm := pack_type.
```

[4] Using this mechanism will, however, afford us a greater degree of flexibility, as it is up to the Coq programmer to define the resolution policy of the combined record's members, rather than to rely on an implicit mechanism of field linearization.

```
Notation PCMMixin := Mixin.
Notation PCM T m := (@Pack T m).

Notation "x \+ y" := (join x y) (at level 43, left associativity).
Notation valid := valid.
Notation Unit := unit.
```

We will have to define the coercion from the PCM structure with respect to its `type` field once again, as the previous one was defined locally for the section Packing, and, hence, is invisible in this submodule.

```
Coercion type : pack_type >-> Sortclass.
```

7.2 Properties of partial commutative monoids

Before we close the *Exports* module of the *PCMDef* package, it makes sense to supply as many properties to the clients, as it will be necessary for them to build the reasoning involving PCMs. In the traditions of proper encapsulation, requiring to expose only the relevant and as abstract as possible elements of the interface to its clients, it is undesirable for users of the pcm datatype to perform any sort of analysis on the structure of the **mixin_of** datatype, as it will lead to rather tedious and cumbersome proofs, which will first become a subject of massive changes, once we decide to change the implementation of the PCM mixin structure.

This is why in this section we supply a number of properties of PCM elements and operations, derived from its structure, which we observe to be enough to build the reasoning with arbitrary PCM instances.

```
Section PCMLemmas.
Variable U : pcm.
```

For instance, the following lemma re-establishes the commutativity of the \+ operation:

```
Lemma joinC (x y : U) : x \+ y = y \+ x.
Proof.
by case: U x y⇒ tp [v j z Cj *]; apply Cj.
Qed.
```

Notice that in order to make the proof to go through, we had to "push" the PCM elements x and y to be the assumption of the goal before case-analysing on U. This is due to the fact that the structure of U affects the type of x and y, therefore destructing it by means of `case` would change the representation of x and y as well, doing some rewriting and simplifications. Therefore, when U is being decomposed, all values, whose type depends on it (i.e., x and y) should be in the scope of decomposition. The naming pattern $*$ helped us to give automatic names to all remaining assumptions, appearing from decomposition of U's second component before moving it to the context before finishing the proof by applying the commutativity "field" *Cj*.

```
Lemma joinA (x y z : U) : x \+ (y \+ z) = x \+ y \+ z.
Proof.
by case: U x y z⇒tp [v j z Cj Aj *]; apply: Aj.
Qed.
```

Exercise 7.1 (PCM laws)**.** Prove the rest of the PCM laws.

```
Lemma joinAC (x y z : U) : x \+ y \+ z = x \+ z \+ y.
Lemma joinCA (x y z : U) : x \+ (y \+ z) = y \+ (x \+ z).
Lemma validL (x y : U) : valid (x \+ y) → valid x.
Lemma validR (x y : U) : valid (x \+ y) → valid y.
Lemma unitL (x : U) : (@Unit U) \+ x = x.
Lemma unitR (x : U) : x \+ (@Unit U) = x.
Lemma valid_unit : valid (@Unit U).

End PCMLemmas.
End Exports.
End PCMDef.
Export PCMDef.Exports.
```

7.3 Implementing inheritance hierarchies

By packaging an arbitrary type T into one record with the PCM structure in Section 7.1.3 and supplying it with a specific implicit coercion, we have already achieved some degree of inheritance: any element of a PCM can be also perceived by the system in an appropriate context, as an element of its carrier type.

In this section, we will go even further and show how to build hierarchies of mathematical structures using the same way of encoding inheritance. We will use a *cancellative PCM* as a running example.

```
Module CancelPCM.
```

PCMs with cancellation extend ordinary PCMs with an extra property, that states that the equality $a \bullet b = a \bullet c$ for any a, b and c, whenever $a \bullet b$ is defined, implies $b = c$. We express such property via an additional mixin record type, parametrized over an arbitrary PCM U.

```
Record mixin_of (U : pcm) := Mixin {
  _ : ∀ a b c: U, valid (a \+ b) → a \+ b = a \+ c → b = c
}.
```

Notice that the validity of the sum $a \backslash+ c$ is not imposed, as it can be proven from propositional equality and the validity of $a \backslash+ b$.

We continue the definition by describing the standard packaging data structure.

```
Structure pack_type : Type := Pack {pcmT : pcm; _ : mixin_of pcmT}.

Module Exports.

Notation cancel_pcm := pack_type.
Notation CancelPCMMixin := Mixin.
Notation CancelPCM T m:= (@Pack T m).
```

There is a tiny twist in the definition of the specific coercion, though, as now we it specifies that the instance of the packed data structure, describing the cancellative PCM,

can be seen as an instance of the underlying PCM. The coercions are transitive, which means that the same instance can be coerced even further to U's carrier type T.

```
Coercion pcmT : pack_type >-> pcm.
```

We finish the definition of the cancellative PCM by providing its only important law, which is a direct consequence of the newly added property.

```
Lemma cancel (U: cancel_pcm) (x y z: U):
  valid (x \+ y) → x \+ y = x \+ z → y = z.
Proof.
by case: U x y z⇒Up [Hc] x y z; apply: Hc.
Qed.

End Exports.
End CancelPCM.

Export CancelPCM.Exports.
```

The proof of the following lemma, combining commutativity and cancellativity, demonstrates how the properties of a cancellative PCM work in combination with the properties of its base PCM structure.

```
Lemma cancelC (U: cancel_pcm) (x y z : U) :
  valid (y \+ x \+ z) → y \+ x = x \+ z → y = z.
Proof.
by move/validL; rewrite ![y \+ _]joinC; apply: cancel.
Qed.
```

7.4 Instantiation and canonical structures

Now, as we have defined a PCM structure along with its specialized version, a cancellative PCM, it is time to see how to *instantiate* these abstract definitions with concrete datatypes, i.e., *prove* the latter ones to be instances of a PCM.

7.4.1 Defining arbitrary PCM instances

Natural numbers form a PCM, in particular, with addition as a join operation and zero as a unit element. The validity predicate is constant true, because the addition of two natural numbers is again a valid natural number. Therefore, we can instantiate the PCM structure for **nat** as follows, first by constructing the appropriate mixin.

```
Definition natPCMMixin :=
  PCMMixin addnC addnA add0n (fun x y ⇒ @id true) (erefl _).
```

The constructor PCMMixin, defined in Section 7.1.3 is invoked with five parameters, all of which correspond to the properties, ensured by the PCM definition. The rest of the arguments, namely, the validity predicate, the join operation and the zero element are implicit and are soundly inferred by Coq's type inference engine from the types of lemmas, provided as propositional arguments. For instance, the first argument addnC, whose type is commutative *addn* makes it possible to infer that the join operation is the

addition. In the same spirit, the third argument, add0n makes it unambiguous that the unit element is zero.

After defining the PCM mixin, we can instantiate the PCM packed class for **nat** by the following definition:

```
Definition NatPCM := PCM nat natPCMMixin.
```

This definition will indeed work, although, being somewhat unsatisfactory. For example, assume we want to prove the following lemma for natural numbers treated as elements of a PCM, which should trivially follow from the PCM properties of **nat** with addition and zero:

```
Lemma add_perm (a b c : nat) : a \+ (b \+ c) = a \+ (c \+ b).
```

```
The term "a" has type "nat" while it is expected to have type "PCMDef.type ?135".
```

This error is due to the fact that Coq is unable to recognize natural numbers to be elements of the corresponding PCM, and one possible way to fix it is to declare the parameters of the lemma *add_perm*, *a*, *b* and *c* to be of type NatPCM rather than **nat**. This is still awkward: it means that the lemmas cannot be just applied to mere natural numbers, instead they need to be *coerced* to the NatPCM type explicitly whenever we need to apply this lemma. Coq suggests a better solution to this problem by providing a mechanism of *canonical structures* as a flexible way to specify *how exactly* each concrete datatype should be embedded into an abstract mathematical structure [57].

The Vernacular syntax for defining canonical structures is similar to the one of definitions and makes use of the Canonical command.[5] The following definition defines natPCM to be a canonical instance of the PCM structure for natural numbers.

```
Canonical natPCM := PCM nat natPCMMixin.
```

To see what kind of effect it takes, we will print all *canonical projections*, currently available in the context of the module.

```
Print Canonical Projections.
```

```
...
nat ← PCMDef.type ( natPCM )
pred_of_mem ← topred ( memPredType )
pred_of_simpl ← topred ( simplPredType )
sig ← sub_sort ( sig_subType )
number ← sub_sort ( number_subType )
...
```

The displayed list enumerates all *canonical projections* that specify, which implicit canonical instances are currently available and will be picked implicitly for appropriate types (on the left of the arrow ←). That is, for example, whenever an instance of **nat** is available, but in fact it should be treated as the type field of the PCM structure (with all getters typed properly), the canonical instance natPCM will be automatically picked by

[5] The command `Canonical Structure` serves the same purpose.

Coq for such embedding. In other words, the machinery of canonical structures allows us to define the policy for finding an appropriate *dictionary* of functions and propositions for an arbitrary concrete datatype, whenever it is supposed to have them. In fact, upon declaring the canonical structure natPCM, the canonical projections are registered by Coq for all *named* fields of the record PCM, which is precisely just the `type` field, since PCM's second component of type (**mixin_of** `type`) was left unnamed (see the definition of the record **pack_type** on page 102).

The mechanism of defining canonical structures for concrete data types is reminiscent of the resolution of type class constraints in Haskell [64]. However, unlike Haskell, where the resolution algorithm for type class instances is *hard-coded*, in the case of Coq one can actually *program* the way the canonical instances are resolved.[6] This leads to a very powerful technique to automate the process of theorem proving by encoding the way to find and apply necessary lemmas, whenever it is required. These techniques are, however, outside of the scope of this course, so we direct the interested reader to the relevant research papers that describe the patterns of programming with canonical structures [19, 25, 37].

Similarly to the way we have defined a canonical instance of PCM for **nat**, we can define a canonical instance of a PCM with cancellativity. In order to instantiate it, we will, however, need to prove the following lemma, which states that the addition on natural numbers is indeed cancellative, so this fact will be used as an argument for the CancelPCMMixin constructor.

```
Lemma cancelNat : ∀ a b c: nat, true → a + b = a + c → b = c.
Proof.
move⇒ a b c; elim: a=>// n /(_ is_true_true) Hn _ H.
by apply: Hn; rewrite !addSn in H; move/eq_add_S: H.
Qed.
```

Notice the first assumption true of the lemma. Here it serves as a placeholder for the general validity hypothesis valid $(a \setminus\!+ b)$, which is always true in the case of natural numbers.

```
Definition cancelNatPCMMixin := CancelPCMMixin cancelNat.
Canonical cancelNatPCM := CancelPCM natPCM cancelNatPCMMixin.
```

Let us now see the canonical instances in action, so we can prove a number of lemmas about natural numbers employing the general PCM machinery.

```
Section PCMExamples.

Variables a b c: nat.

Goal a \+ (b \+ c) = c \+ (b \+ a).
by rewrite joinA [c \+ _]joinC [b \+ _]joinC.
Qed.
```

The next goal is proved by using the combined machinery of PCM and CancelPCM.

```
Goal c \+ a = a \+ b → c = b.
```

[6] In addition to canonical structures, Coq also provides mechanism of type classes, which are even more reminiscent of the ones from Haskell, and, similar to the latter ones, do not provide a way to program the resolution policy [59].

```
by rewrite [c \+ _]joinC; apply: cancel.
Qed.
```

It might look a bit cumbersome, though, to write the PCM join operation $\backslash+$ instead of the boolean addition when specifying the facts about natural numbers (even though they are treated as elements of the appropriate PCM). Unfortunately, it is not trivial to encode the mechanism, which will perform such conversion implicitly. Even though Coq is capable of figuring out what PCM is necessary for a particular type (if the necessary canonical instance is defined), e.g., when seeing (a b : **nat**) being used, it infers the natPCM, alas, it's not powerful enough to infer that by writing the addition function + on natural numbers, we mean the PCM's join. However, if necessary, in most of the cases the conversion like this can be done by manual rewriting using the following trivial "conversion" lemma.

```
Lemma addn_join (x y: nat): x + y = x \+ y.
Proof. by []. Qed.

End PCMExamples.
```

Exercise 7.2 (Partially ordered sets)**.** A partially ordered set order is a pair $(T, \sqsubseteq)$, such that T is a set and $\sqsubseteq$ is a (propositional) relation on T, such that

1. $\forall x \in T, x \sqsubseteq x$ (reflexivity);
2. $\forall x, y \in T, x \sqsubseteq y \wedge y \sqsubseteq x \implies x = y$ (antisymmetry);
3. $\forall x, y, z \in T, x \sqsubseteq y \wedge y \sqsubseteq z \implies x \sqsubseteq z$ (transitivity).

Implement a data structure for partially-ordered sets using mixins and packed classes. Prove the following laws:

```
Lemma poset_refl (x : T) : x ⊑ x.
Lemma poset_asym (x y : T) : x ⊑ y → y ⊑ x → x = y.
Lemma poset_trans (y x z : T) : x ⊑ y → y ⊑ z → x ⊑ z.
```

Exercise 7.3 (Canonical instances of partially ordered sets)**.** Provide canonical instances of partially ordered sets for the following types:

- **nat** with $\leq$ as a partial order;
- *prod*, whose components are partially-ordered sets;
- functions $A \to B$, whose codomain (range) B is a partially ordered set.

In order to provide a canonical instance for functions, you will need to assume and make use of the following axiom of functional extensionality:

```
Axiom fext : ∀ A (B : A → Type) (f1 f2 : ∀ x, B x),
                (∀ x, f1 x = f2 x) → f1 = f2.
```

7.4.2 Types with decidable equalities

When working with Ssreflect and its libraries, one will always come across multiple canonical instances of a particularly important dependent record type—a structure with decidable equality. As it has been already demonstrated in Section 5.3.2, for concrete datatypes, which enjoy the decidable boolean equality (==), the "switch" to Coq's propositional equality and back can be done seamlessly by means of using the view lemma *eqP*, leveraging the *reflect* predicate instance of the form *reflect* (*b1* = *b2*) (*b1* == *b2*). Let us now show how the decidable equality is defined and instantiated.

The module eqtype of Ssreflect's standard library provides a definition of the equality mixin and packaged class of the familiar shape, which, after some simplifications, boil to the following ones:

```
Module Equality.

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

Structure mixin_of T := Mixin {op : rel T; _ : axiom op}.
Structure type := Pack {sort; _ : mixin_of sort}.

...

Notation EqMixin := Mixin.
Notation EqType T m := Pack T m.

End Equality.
```

That is, the mixin for equality is a dependent record, whose first field is a relation *op* on a particular carrier type T (defined internally as a function $T \times T \to$ **bool**), and the second argument is a proof of the definition *axiom*, which postulates that the relation is in fact equivalent to propositional equality (which is established by means of inhabiting the *reflect* predicate instance). Therefore, in order to make a relation *op* to be a decidable equality on T, one needs to prove that, in fact, it is equivalent to the standard, propositional equality.

Subsequently, Ssreflect libraries deliver the canonical instances of the decidable equality structure to all commonly used concrete datatypes. For example, the decidable equality for natural numbers is implemented in the ssrnat module by the following recursive function:[7]

```
Fixpoint eqn m n {struct m} :=
  match m, n with
  | 0, 0 ⇒ true
  | m'.+1, n'.+1 ⇒ eqn m' n'
  | _, _ ⇒ false
```

[7] Coq's {struct *n*} annotation explicitly specifies, which of the two arguments should be considered by the system as a decreasing one, so the recursion would be well-founded and *eqn* would terminate.

```
  end.
```

The following lemma ensures that *eqn* correctly reflects the propositional equality.

```
Lemma eqnP : Equality.axiom eqn.
Proof.
move⇒ n m; apply: (iffP idP) ⇒ [ | ← ]; last by elim n.
by elim: n m ⇒ [ | n IHn] [| m ] //= /IHn →.
Qed.
```

Finally, the following two definitions establish the canonical instance of the decidable equality for **nat**, which can be used whenever ssrnat is imported.

```
Canonical nat_eqMixin := EqMixin eqnP.
Canonical nat_eqType := EqType nat nat_eqMixin.
```

8 Case Study: Program Verification in Hoare Type Theory

In this chapter, we will consider a fairly large case study that makes use of most of Coq's features as a programming language with dependent types and as a framework to build proofs and reason about mathematical theories.

Programming language practitioners usually elaborate on the dichotomy between *declarative* and *imperative* languages, emphasizing the fact that a program written in a declarative language is pretty much documenting itself, as it already specifies the *result* of a computation. Therefore, logic and constraint programming languages (such as Prolog [36] or Ciao [28]) as well as data definition/manipulation languages (e.g., SQL), whose programs are just sets of constraints/logical clauses or queries describing the desired result, are naturally considered to be declarative. Very often, pure functional programming languages (e.g., Haskell) are considered as declarative as well. The reason for this is the *referential transparency* property, which ensures that programs in such languages are in fact effect-free expressions, evaluating to some result (similar to mathematical functions) or diverging. Therefore, such programs, whose outcome is only a value, but not some side effect (e.g., output to a file), can be replaced safely by their result, if it is computable. This possibility provides a convenient way of reasoning algebraically about such programs by means of equality rewritings—precisely what we were observing and leveraging in Chapters 4 and 6 of this course in the context of Coq taken as a functional programming language.

That said, pure functional programs tend to be considered to be good specifications for themselves. Of course, the term "specification" (or simply, "spec") is overloaded and in some context it might mean the result of the program, its effect or some of the program's algebraic properties. While a functional program is already a good description of its result (due to referential transparency), its algebraic properties (e.g., some equalities that hold over it) are usually a subject of separate statements, which should be proved [4]. Good examples of such properties are the commutativity and cancellation properties, which we proved for natural numbers with addition, considered as an instance of PCM on page 107 of Chapter 7. Another classical series of examples, which we did not focus on in this course, are properties of list functions, such as appending and reversal (e.g., that the list

reversal is an inverse to itself).[1]

The situation is different when it comes to imperative programs, whose outcome is typically their side-effect and is achieved by means of manipulating mutable state, throwing an exception or performing input/output. While some of the modern programming languages (e.g., Scala, OCaml) allow one to mix imperative and declarative programming styles, it is significantly harder to *reason* about such programs, as now they cannot be simply replaced by their results: one should also take into account the effect of their execution (i.e., changes in the mutable state). A very distinct approach to incorporating both imperative and declarative programming is taken by Haskell, in which effectful programs can always be distinguished from pure ones by means of enforcing the former ones to have very specific types [49]—the idea we will elaborate more on a bit further.

In the following sections of this chapter, we will learn how Coq can be used to give specifications to imperative programs, written in a domain-specific language, similar to C, but in fact being a subset of Coq itself. Moreover, we will observe how the familiar proof construction machinery can be used to establish the correctness of these specifications, therefore, providing a way to *verify* a program by means of checking, whether it satisfies a given spec. In particular, we will learn how the effects of state-manipulating programs can be specified via dependent types, and the specifications of separate effectful programs can be *composed*, therefore allowing us to structure the reasoning in a modular way, similarly to mathematics, where one needs to prove a theorem only once and then can just rely on its statement, so it can be employed in the proofs of other facts.

8.1 Imperative programs and their specifications

The first attempts to specify the behaviour of state-manipulating imperative programs with assignments originated in late '60s and are due to Tony Hoare and Robert Floyd [18, 29], who considered programs in a simple imperative language with mutable variables (but without pointers or procedures) and suggested to give a specification to a program c in the form of a triple $\{P\}\ c\ \{Q\}$, where P and Q are logical propositions, describing the values of the mutable variables and possible relations between them. P and Q are usually referred to as *assertions*; more specifically, P is called *precondition* of c (or just "pre"), whereas Q is called *postcondition* (or simply "post"). The triple $\{P\}\ c\ \{Q\}$ is traditionally referred to as *Hoare triple*.[2] Its intuitive semantics can be expressed as follows: "if right before the program c is executed the state of mutable variables is described by the proposition P, then, *if* c *terminates*, the resulting state satisfies the proposition Q".

[1] A common anti-pattern in dependently-typed languages and Coq in particular is to encode such algebraic properties into the definitions of the datatypes and functions themselves (a canonical example of such approach are length-indexed lists). While this approach looks appealing, as it demonstrates the power of dependent types to capture certain properties of datatypes and functions on them, it is inherently non-scalable, as there will be always another property of interest, which has not been foreseen by a designer of the datatype/function, so it will have to be encoded as an external fact anyway. This is why we advocate the approach, in which datatypes and functions are defined as close to the way they would be defined by a programmer as possible, and all necessary properties of them are proved separately.

[2] The initial syntax for the triples used by Hoare was $P\ \{c\}\ Q$. The notation $\{P\}\ c\ \{Q\}$, which is used now consistently, is due to Niklaus Wirth and emphasizes the comment-like nature of the assertions in the syntax reminiscent to the one of Pascal.

The reservation on termination of the program c is important. In fact, while the Hoare triples in their simple form make sense only for terminating programs, it is possible to specify non-terminating programs as well. This is due to the fact that the semantics of a Hoare triple implies that a non-terminating program can be given *any* postcondition, as one won't be able to check it anyway, because the program will never reach the final state.[3] Such interpretation of a Hoare triple "modulo termination" is referred to as *partial correctness*, and in this chapter we will focus on it. It is possible to give to a Hoare triple $\{P\}\ c\ \{Q\}$ a different interpretation, which would deliver a stronger property: "if right before the program c is executed the state of mutable variables is described by a proposition P, *then* c *terminates* and the resulting state satisfies the proposition Q". Such property is called *total correctness* and requires tracking some sort of "fuel" for the program in the assertions, so it could run further. We do not consider total correctness in this course and instead refer the reader to the relevant research results on Hoare-style specifications with resource bounds [16].

8.1.1 Specifying and verifying programs in a Hoare logic

The original Hoare logic worked over a very simplistic imperative language with loops, conditional operators and assignments. This is how one can specify a program, which just assigns 3 to a specific variable named `x`:

$$\{\mathsf{true}\}\ \texttt{x := 3}\ \{x = 3\}$$

That is, the program's precondition doesn't make any specific assumptions, which is expressed by the proposition `true`; the postcondition ensures that the value of a mutable variable `x` is equal to three.

The formalism, which allows us to validate particular Hoare triples for specific programs is called *program logic* (or, equivalently, *Hoare logic*).

Intuitively, logic in general is a formal system, which consists of axioms (propositions, whose inhabitance is postulated) and *inference rules*, which allow one to construct proofs of larger propositions out of proofs of small ones. This is very much of the spirit of Chapter 3, where we were focusing on a particular formalism—propositional logic. Its *inference rules* were encoded by means of Coq's *datatype constructors*. For instance, in order to construct a proof of conjunction (i.e., inhabit a proposition of type $A \wedge B$), one should have provided a proof of a proposition A and a proposition B and then *apply* the only conjunction's constructor *conj*, as described in Section 3.4. The logicians, however, prefer to write inference rules as "something with a bar", rather than as constructors. Therefore, an inference rule for conjunction introduction in the constructive logic looks as follows:

$$\frac{A \qquad B}{A \wedge B}\ (\wedge\text{-Intro})$$

[3] This intuition is consistent with the one, enforced by Coq's termination checker, which allows only terminating programs to be written, since non-terminating program can be given any type and therefore compromise the consistency of the whole underlying logical framework of CIC.

That is, the rule (∧-Intro) is just a paraphrase of the *conj* constructor, which specifies how an instance of conjunction can be created. Similarly, the disjunction *or* has two inference rules, for each of its constructors. The elimination rules are converses of the introduction rules and formalize the intuition behind the case analysis. An alternative example of an inference rule for a proposition encoded by means of Coq's datatype constructor is the definition of the predicate for beautiful numbers *beautiful* from Section 6.3. There, the constructor *b_sum* serves as an inference rule that, given the proofs that *n'* is beautiful and *m'* is beautiful, constructs the proof of the fact that their sum is beautiful.[4]

Hoare logic also suggests a number of axioms and inference rules that specify which Hoare triple can in fact be inferred. We postpone the description of their encoding by means of Coq's datatypes till Section 8.4 of this chapter and so far demonstrate some of them in the logical notation "with a bar". For example, the Hoare triple for a variable assignment is formed by the following rule:

$$\{Q[e/x]\}\ x := e\ \{Q\}\ \text{(Assign)}$$

The rule (Assign) is in fact an axiom (since it does not assume anything, i.e., does not take any arguments), which states that if a proposition Q is valid after substituting all occurrences of x in it with e (which is denoted by $[e/x]$), then it is a valid postcondition for the assignment $x := e$.

The inference rule for sequential composition is actually a constructor, which takes the proofs of Hoare triples for c_1 and c_2 and then delivers a composed program $c_1; c_2$ *as well as* the proof for the corresponding Hoare triple, ensuring that the postcondition of c_1 matches the precondition of c_2.

$$\frac{\{P\}\ c_1\ \{R\} \qquad \{R\}\ c_2\ \{Q\}}{\{P\}\ c_1; c_2\ \{Q\}}\ \text{(Seq)}$$

The rule (Seq) is in fact too "tight", as it requires the two composed program agree exactly on their post-/preconditions. In order to relax this restriction, on can use the *rule of consequence*, which makes it possible to *strengthen* the precondition and *weaken* the postcondition of a program. Intuitively, such rule is adequate, since the program that is fine to be run in a precondition P', can be equivalently run in a stronger precondition P (i.e., the one that implies P'). Conversely, if the program terminates in a postcondition Q', it would not hurt to weaken this postcondition to Q, such that Q' implies Q.

$$\frac{P \implies P' \qquad \{P'\}\ c\ \{Q'\} \qquad Q' \implies Q}{\{P\}\ c\ \{Q\}}\ \text{(Conseq)}$$

[4] Actually, some courses on Coq introduce datatype constructors exactly from this perspective—as a programming counterpart of the introduction rules for particular kinds of logical propositions [53]. We came to the same analogy by starting from an opposite side and exploring the datatypes in the programming perspective first.

With this respect, we can make the analogy between Hoare triples and function types of the form $A \to B$, such that the rule of consequence of a Hoare triple corresponds to subtyping of function types, where the precondition P is analogous to an argument type A and the postcondition Q is analogous to a result type B. Similarly to the functions with subtyping, Hoare triples are covariant with respect to consequence in their postcondition and *contravariant* in the precondition [52, Chapter 15]. This is the reason why, when establishing a specification, one should strive to infer the *weakest precondition and the strongest postcondition* to get the tightest possible (i.e., the most precise) spec, which can be later weakened using the (Conseq) rule.

The observed similarity between functions and commands in a Hoare logic should serve as an indicator that, perhaps, it would be a good idea to implement the Hoare logic in a form of a type system. Getting a bit ahead of ourselves, this is exactly what is going to happen soon in this chapter (as the title of the chapter suggests).

At this point, we can already see a simple paper-and-pencil proof of a program that manipulates mutable variables. In the Hoare logic tradition, since most of the programs are typically compositions of small programs, the proofs of specifications are written to follow the structure of the program, so the first assertion corresponds to the overall precondition, the last one is the overall postcondition, and the intermediate assertions correspond to R from the rule (Seq) modulo weakening via the rule of consequence (Conseq). Let us prove the following Hoare-style specification for a program that swaps the values of two variables x and y.

$$\{x = a \land y = b\}\ \texttt{t := } x\texttt{; } x\texttt{ := } y\texttt{; } y\texttt{ := t}\ \{x = b \land y = a\}$$

The variables a and b are called *logical* and are used in order to name unspecified values, which are a subject of manipulation in the program, and their identity should be preserved. The logical variables are implicitly universally quantified over in the scope of the *whole* Hoare triple they appear, but usually the quantifiers are omitted, so, in fact, the specification above should have been read as follows.

$$\forall\ a\ b,\ \{x = a \land y = b\}\ \texttt{t := } x\texttt{; } x\texttt{ := } y\texttt{; } y\texttt{ := t}\ \{x = b \land y = a\}$$

This universal quantification should give some hints that converting Hoare triples into types will, presumably, require to make some use of dependent types in order to express value-polymorphism, similarly to how the universal quantification has been previously used in Coq. Let us see a proof sketch of the above stated specification with explanations of the rules applied after each assertion.

$\{x = a \land y = b\}$ The precondition
`t := x;`
$\{x = a \land y = b \land t = a\}$ by (Assign) and equality
x `:= y;`
$\{x = b \land y = b \land t = a\}$ by (Assign) and equality
y `:= t`
$\{x = b \land y = a\}$ by (Assign) equality and weakening via (Conseq)

The list of program constructs and inference rules for them would be incomplete without conditional operators and loops.

$$\frac{\{P \wedge e\}\ c_1\ \{Q\} \qquad \{P \wedge \neg e\}\ c_2\ \{Q\}}{\{P\}\ \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2\ \{Q\}}\ (\textsc{Cond})$$

$$\frac{\{I \wedge e\}\ c\ \{I\}}{\{I\}\ \texttt{while}\ e\ \texttt{do}\ c\ \{I \wedge \neg e\}}\ (\textsc{While})$$

The inference rule for a conditional statement should be intuitively clear and reminds of a typing rule for conditional expressions in Haskell or OCaml, which requires both branches of the statement to have the same type (and here, equivalently, to satisfy the same postcondition). The rule (While) for the loops is more interesting, as it makes use of the proposition I, which is called *loop invariant*. Whenever the body of the cycle is entered, the invariant should hold (as well as the condition e, since the iteration has just started). Upon finishing, the body c should restore the invariant, so the next iteration would start in a consistent state again. Generally, it takes a human prover's intuition to come up with a non-trivial resource invariant for a loop, so it can be used in the rest of the program. Inference of the best loop invariant is an undecidable problem in general and it has a deep relation to type inference with polymorphically-recursive functions [27]. This should not be very surprising, since every loop can be encoded as a recursive function, and, since, as we have already started guessing, Hoare triples are reminiscent of types, automatic inferring of loop invariants would correspond to type inference for recursive functions. In the subsequent sections we will see examples of looping/recursive programs with loop invariants and exercise in establishing some of them.

8.1.2 Adequacy of a Hoare logic

The original Hoare logic is often referred to as *axiomatic semantics* of imperative programs. This term is only partially accurate, as it implies that the Hoare triples describe precisely what is the program and how it behaves. Even though Hoare logic can be seen as a program semantics as a way to describe the program's behaviour, it is usually not the only semantics, which imperative programs are given. In particular, it does not say how a program should be executed—a question answered by operational semantics [67, Chapter 2]. Rather, Hoare logic allows one to make statements about the effect the program takes to the mutable state, and, what is more important, construct finite proofs of these statements. With this respect, Hoare logic serves the same purpose as type systems in many programming languages—determine statically (i.e., without *executing* the program), whether the program is well-behaved or not. In other words, it serves as an "approximation" of another, more low-level semantics of a program. This intuition is also implied by the very definition of a hoare triple on page 112, which relies to the fact that a program can be executed.

That said, in order to use a Hoare logic for specifying and verifying a program's behaviour, a *soundness* result should be first established. In the case of a program logic, soundness means the logic rules allow one to infer only those statements that do not contradict the definition of a Hoare triple (page 112). This result can be proven in many

different ways, and the nature of the proof usually depends on the underlying operational/denotational semantics, which is typically not questioned, being self-obvious, and defines precisely what does it mean for a program *to be executed*. Traditional ways of proving soundness of a program logic are reminiscent to the approaches for establishing soundness of type systems [52, 68]. Of course, all program logics discussed in this chapter have been proven to be sound with respect to some reasonable operational/denotational semantics.

8.2 Basics of Separation Logic

The original Hoare logic has many limitations. It works only with mutable variables and does not admit procedures or first-order code. But its most severe shortcoming becomes evident when it comes to specifying programs that manipulate *pointers*, i.e., the most interesting imperative cases of imperative code. In the presence of pointers and a heap, mutable variables become somewhat redundant, so for now by *local variables* we will be assuming immutable, once-assigned variables, akin to those bound by the `let`-expression. Such variables can, of course, have pointers as their values. Let us first enrich the imperative programming language of interest to account for the presence of heap and pointers. We will be using the syntax `x ::= e` to denote the assignment of a value `e` to the pointer bound by `x`. Similarly, the syntax `!e` stands for dereferencing a pointer, whose address is a value obtained by evaluating a *pure* expression `e`. We will assume that every program returns a value as a result (and the result of a pointer assignment is of type **unit**). To account for this, we will be using the syntax `x <- c1; c2` (pronounced "bind") as a generalization of the sequential composition from Section 8.1.1. The bind first executes the program `c1`, then *binds* this result to an immutable variable `x` and proceeds to the execution of the program `c2`, which can possibly make use of the variable `x`, so all the occurrences of `x` in it are replaced by its value before it starts evaluating. If the result of `c1` is not used by `c2`, we will use the abbreviation `c1 ;; c2` to denote this specific case. Specifications in the simple Hoare logic demonstrated in Section 8.1.1 are stated over mutable local variables, which are implicitly supposed to be all distinct, as they have distinct "abstract" names. In a language with a heap and pointers, the state is no longer a set of mutable variables, but the heap itself, which can be thought of as a partial map from natural numbers to arbitrary values. In a program, operating with a heap, two pointer variables, x and y can in fact be *aliases*, i.e., refer to the same memory entry, and, therefore, changing a value of a pointer, referenced by x will affect the value, pointed to by y. Aliasing is an aspect that renders reasoning in the standard Hoare logic tedious and unbearable. To illustrate the problem, let us consider the following program, which runs in the assumption that the heap, which is being available to the program, has only two entries with addresses, referred to by local variables x and y correspondingly, so the specification states it by means of the "points-to" assertions $x \mapsto -$, where x is assumed to be a pointer value.

$\{x \mapsto - \wedge y \mapsto Y\}$ `x ::= 5` $\{x \mapsto 5 \wedge y \mapsto Y\}$

The logical variable Y is of importance, as it is used to state that the value of the

pointer y remains unchanged after the program has terminated.[5] Alas, this specification is not correct, as the conjunction of the two does not distinguish between the case when x and y are the same pointer and when they are not, which is precisely the aliasing problem. It is not difficult to fix the specification for this particular example by adding a conditional statement (or, equivalently, a disjunction) into the postcondition that would describe two different outcomes of the execution with respect to the value of y, depending on the fact whether x and y are aliases or not. However, if a program manipulates with a large number of pointers, or, even worse, with an array (which is obviously just a sequential batch of pointers), things will soon go wild, and the conditionals with respect to equality or non-equality of certain pointers will pollute all the specifications, rendering them unreadable and eventually useless. This was precisely the reason, why after being discovered in late '60s and investigated for a decade, Hoare-style logics were soon almost dismissed as a specification and verification method, due to the immense complexity of the reasoning process and overwhelming proof obligations. The situation has changed when in 2002 John C. Reynolds, Peter O'Hearn, Samin Ishtiaq and Hongseok Yang suggested an alternative way to state Hoare-style assertions about heap-manipulating programs with pointers [55]. The key idea was to make *explicit* the fact of disjointness (or, *separation*) between different parts of a heap in the pre- and postconditions. This insight made it possible to reason about disjointness of heaps and absence of aliasing without the need to emit side conditions about equality of pointers. The resulting formal system received the name *separation logic*, and below we consider a number of examples to specify and verify programs in it. For instance, the program, shown above, which assigns 5 to a pointer x can now be given the following specification in the separation logic:

$$\{h \mid h = x \mapsto - \bullet y \mapsto Y\}\ x ::= 5\ \{\mathsf{res}, h \mid h = x \mapsto 5 \bullet y \mapsto Y\}$$

We emphasize the fact that the heaps, being just partial maps from natural numbers to arbitrary values, are elements of a PCM (Section 7.1) with the operation $\bullet$ taken to be a disjoint union and the unit element to be an empty heap (denoted empty). The above assertions therefore ensure that, before the program starts, it operates in a heap h, such that h is a partial map, consisting of two *different* pointers, x and y, such that y points to some universally-quantified value Y, and the content of x is of no importance (which is denoted by -). The postcondition makes it explicit that only the value of the pointer x has changed, and the value of y remained the same. The postcondition also mentions the result res of the whole operations, which is, however, not constrained anyhow, since, as it has been stated, it is just a value of type **unit**.[6]

[5] We will abuse the terminology and refer to the values and immutable local variables uniformly, as, unlike the setting of Section 8.1, the latter ones are assumed to be substituted by the former ones during the evaluation anyway.

[6] The classical formulation of Separation Logic [55] introduces the logical connective $*$, dubbed *separating conjunction*, which allows to describe the split of a heap h into two disjoint parts without mentioning h explicitly. That is, the assertion $P * Q$ holds for a heap h, if there exist heaps h_1 and h_2, such that $h = h_1 \bullet h_2$, P is satisfied by h_1 and Q is satisfied by h_2. We will stick to the "explicit" notation, though, as it allows for greater flexibility when stating the assertions, mixing both heaps and values.

8.2.1 Selected rules of Separation Logic

Let us now revise some of the rules of Hoare logic and see how they will look in separation logic. The rules, stated over the heap, are typically given in the *small footprint*, meaning that they are stated with the smallest possible heap and assume that the "rest" of the heap, which is unaffected by the program specified, can be safely assumed. The rules for assigning and reading the pointers are natural.

$$\{h \mid h = x \mapsto -\}\ x ::= e\ \{\mathsf{res}, h \mid h = x \mapsto e \wedge \mathsf{res} = \mathtt{tt}\}\ (\textsc{Write})$$

$$\{h \mid h = x \mapsto v\}\ !x\ \{\mathsf{res}, h \mid h = x \mapsto v \wedge \mathsf{res} = v\}\ (\textsc{Read})$$

Notice, though, that, unlike the original Hoare logic for mutable variables, the rule for writing explicitly requires the pointer x to be present in the heap. In other words, the corresponding memory cell should be already *allocated*. This is why the traditional separation logic assumes presence of an allocator, which can allocate new memory cells and dispose them via the effectful functions `alloc()` and `dealloc()`, correspondingly.

$$\{h \mid h = h'\}\ \mathtt{alloc}(v)\ \{\mathsf{res}, h \mid h = \mathsf{res} \mapsto v \bullet h'\}\ (\textsc{Alloc})$$

$$\{h \mid h = x \mapsto - \bullet h'\}\ \mathtt{dealloc}(x)\ \{\mathsf{res}, h \mid h = h'\}\ (\textsc{Dealloc})$$

For the sake of demonstration, the rules for `alloc`() and `dealloc`() are given in a *large footprint* that, in contrast with small footprint-like specifications, mentions the "additional" heap h' in the pre- and post-conditions, which can be arbitrarily instantiated, emphasizing that it remains unchanged (recall that h' is implicitly universally-quantified over, and its scope is the whole triple), so the resulting heap is just being "increased"/"decreased" by a memory entry that has been allocated/deallocated.[7] The rule for binding is similar to the rule for sequential composition of programs c_1 and c_2 from the standard Hoare logic, although it specifies that the immutable variables can be substituted in c_2.

$$\frac{\{h \mid P(h)\}\ c_1\ \{\mathsf{res}, h \mid Q(\mathsf{res}, h)\} \qquad \{h \mid Q(x, h)\}\ c_2\ \{\mathsf{res}, h \mid R(\mathsf{res}, h)\}}{\{h \mid P(h)\}\ x \leftarrow c_1; c_2\ \{\mathsf{res}, h \mid R(\mathsf{res}, h)\}}\ (\textsc{Bind})$$

The predicates P, Q and R in the rule (Bind) are considered to be functions of the heap and result, correspondingly. This is why for the second program, c_2, the predicate Q in a precondition is instantiated with x, which can occur as a free variable in c_2. The rule of weakening (Conseq) is similar to the one from Hoare logic modulo the technical details on how to weaken heap/result parametrized functions, so we omit it here as an intuitive one. The rule for conditional operator is the same one as in Section 8.1.1, and, hence, is omitted as well. In order to support procedures in separation logic, we need to consider two additional rules—for function invocation and returning a value.

[7] The classical separation logic provides a so-called *frame rule*, which allows for the switch between the two flavours of footprint by attaching/removing the additional heap h'. In our reasoning we will be assuming it implicitly.

$$\{h \mid P(h)\}\ \mathsf{ret}\ e\ \{\mathsf{res}, h \mid P(h) \wedge \mathsf{res} = e\}\ (\textsc{Return})$$

$$\frac{\forall x, \{h \mid P(x,h)\}\ f(x)\ \{\mathsf{res}, h \mid Q(x, \mathsf{res}, h)\} \in \Gamma}{\Gamma \vdash \forall x, \{h \mid P(x,h)\}\ f(x)\ \{\mathsf{res}, h \mid Q(x, \mathsf{res}, h)\}}\ (\textsc{Hyp})$$

$$\frac{\Gamma \vdash \forall x, \{h \mid P(x,h)\}\ f(x)\ \{\mathsf{res}, h \mid Q(x, \mathsf{res}, h)\}}{\Gamma \vdash \{h \mid P(e,h)\}\ f(e)\ \{\mathsf{res}, h \mid Q(e, \mathsf{res}, h)\}}\ (\textsc{App})$$

The rule for returning simply constraints the dedicated variable res to be equal to the expression e. The rule (Hyp) (for "hypothesis") introduces the assumption context Γ that contains specifications of available "library" functions (bearing the reminiscence with the typing context in typing relations [52, Chapter 9]) and until now was assumed to be empty. Notice that, similarly to dependently-typed functions, in the rule (Hyp) the pre- and postcondition in the spec of the assumed function can depend on the value of its argument x. The rule (App) accounts for the function application and instantiates all occurrences of x with the argument expression e. Finally, sometimes we might be able to infer two different specifications about the same program. In this case we should be able to combine them into one, which is stronger, and this is what the rule of conjunction (Conj) serves for:

$$\frac{\{h \mid P(h)\}\ c\ \{\mathsf{res}, h \mid Q_1(\mathsf{res}, h)\} \qquad \{h \mid P(h)\}\ c\ \{\mathsf{res}, h \mid Q_2(\mathsf{res}, h)\}}{\{h \mid P(h)\}\ c\ \{\mathsf{res}, h \mid Q_1(\mathsf{res}, h) \wedge Q_2(\mathsf{res}, h)\}}\ (\textsc{Conj})$$

8.2.2 Representing loops as recursive functions

It is well-known in a programming language folklore that every imperative loop can be rewritten as a function, which is tail-recursive, i.e., it performs the call of itself only as the very last statement in some possible execution branches and doesn't call itself at all in all other branches. Therefore, recursive functions in general are a more expressive mechanism, as they also allow one to write non-tail recursive programs, in which recursive calls occur in any position.[8] Therefore, an imperative program of the form

```
while (e) do c
```

can be rewritten using a recursive function, defined via the in-place fixpoint operator as

```
(fix f (x : bool). if x then c;; f(e') else ret tt)(e)
```

That is, the function `f` is defined with an argument of the bool type and is immediately invoked. If the condition argument `x` is satisfied, the body `c` is executed and the function calls itself recursively with a new argument `e'`; otherwise the function just returns a unit

[8] Although, such programs can be made tail-recursive via the continuation-passing style transformation (CPS) [15]. They can be also converted into imperative loops via the subsequent transformation, known as *defunctionalization* [54].

result. For the first time, the function is invoked with some initial argument `e`. Given this relation between imperative loops and effectful recursive functions, we won't be providing a rule for loops in separation logic at all, and rather provide one for recursive definitions.

$$\frac{\Gamma, \forall x, \{h \mid P(x,h)\}\ f(x)\ \{\mathsf{res}, h \mid Q(x,\mathsf{res},h)\} \vdash \{h \mid P(x,h)\}\ c\ \{\mathsf{res}, h \mid Q(x,\mathsf{res},h)\}}{\Gamma \vdash \forall y, \{h \mid P(y,h)\}\ (\mathsf{fix}\ f(x).c)(y)\ \{\mathsf{res}, h \mid Q(y,\mathsf{res},h)\}}\ (\textsc{Fix})$$

The premise of the rule (Fix) already *assumes* the specification of a function f (i.e., its *loop invariant*) in the context Γ and requires one to verify its body c for the same specification, similarly to how recursive functions in some programming languages (e.g., Scala [46, § 4.1]) require explicit type annotations to be type-checked. In the remainder of this chapter we will be always implementing imperative loops via effectful recursive functions, whose specifications are stated explicitly, so the rule above would be directly applicable.

8.2.3 Verifying heap-manipulating programs

Let us now see how a simple imperative program with conditionals and recursion would be verified in a version of separation logic that we presented here. A subject of our experiment will be an efficient imperative implementation of a factorial-computing function, which *accumulates* the factorial value in a specific variable, while decreasing its argument in a loop, and returns the value of the accumulator when the iteration variable becomes zero. In the pseudocode, the `fact` program is implemented as follows:

```
fun fact (N : nat): nat = {
  n   <-- alloc(N);
  acc <-- alloc(1);
  res <--
    (fix loop (_ : unit).
      a' <-- !acc;
      n' <-- !n;
      if n' == 0 then ret a'
      else acc ::= a' * n';;
           n   ::= n' - 1;;
           loop(tt)
    )(tt);
  dealloc(n);;
  dealloc(acc);;
  ret res
}
```

The function `fact` first allocates two pointers, `n` and `acc` for the iteration variable and the accumulator, correspondingly. It will then initiate the loop, implemented by the recursive function `loop`, that reads the values of `n` and `acc` into local immutable variables `n'` and `a'`, correspondingly and then checks whether `n'` is zero, in which case it returns the value of the accumulator. Otherwise it stores into the accumulator the old

value multiplied by `n'`, decrements `n` and re-iterates. After the loop terminates, the two pointers are deallocated and the main function returns the result. Our goal for the rest of this section will be to verify this program semi-formally, using the rules for separation logic presented above, against its *functional* specification. In other words, we will have to check that the program `fact` returns precisely the factorial of its argument value N. To give such specification to `fact`, we define two auxiliary mathematical functions, f and F_{inv}:

$$\begin{array}{rcl} f(N) & \widehat{=} & \mathsf{if}\ N = N' + 1\ \mathsf{then}\ N \times f(N')\ \mathsf{else}\ 1 \\ F_{inv}(n, acc, N, h) & \widehat{=} & \exists n', a', (h = n \mapsto n' \bullet acc \mapsto a') \wedge (f(n') \times a' = f(N)) \end{array}$$

It is not difficult to see that f defines exactly the factorial function as one would define it in a pure functional language (not very efficiently, though, but in the most declarative form). The second function F_{inv} is in fact a predicate, which we will use to give the loop invariant to the loop function `loop`. Now, the function `fact` can be given the following specification in separation logic, stating that it does not *leak* memory and its result is the factorial of its argument N.

$$\{h \mid h = \mathsf{empty}\}\ \texttt{fact}(N)\ \{\mathsf{res}, h \mid h = \mathsf{empty} \wedge \mathsf{res} = f(N)\}$$

In the course of the proof of the above stated spec of `fact`, in order to apply the rule (Fix), we pose the specification of `loop` (in an implicit assumption context Γ from the rules) to be the following one. The specification states that the body of the loop preserves the invariant F_{inv}, and, moreover its result is the factorial of N.

$$\{h \mid F_{inv}(\mathtt{n}, \mathtt{acc}, N, h)\}\ \texttt{loop(tt)}\ \{\mathsf{res}, h \mid F_{inv}(\mathtt{n}, \mathtt{acc}, N, h) \wedge \mathsf{res} = f(N)\}$$

Below, we demonstrate a proof sketch of verification of the body of `fact` against its specification by systematically applying all of the presented logic rules.

$\{h \mid h = \mathsf{empty}\}$ (by precondition)
```
 n   <-- alloc(N);
```
$\{h \mid h = \mathtt{n} \mapsto N\}$ (by (Alloc) and PCM properties)
```
 acc <-- alloc(1);
```
$\{h \mid h = \mathtt{n} \mapsto N \bullet \mathtt{acc} \mapsto 1\}$ (by (Alloc))
$\{h \mid F_{inv}(\mathtt{n}, \mathtt{acc}, N, h)\}$ (by definition of F_{inv} and (Conseq))
```
 res <--
  (fix loop (_ : unit).
```
$\{h \mid F_{inv}(\mathtt{n}, \mathtt{acc}, N, h)\}$ (precondition)
```
      a' <-- !acc;
```
$\{h \mid \exists n',\ (h = \mathtt{n} \mapsto n' \bullet \mathtt{acc} \mapsto \mathtt{a}') \wedge (f(n') \times a' = f(N))\}$ (F_{inv}, (Read) and (Conj))
```
      n' <-- !n;
```
$\{h \mid (h = \mathtt{n} \mapsto \mathtt{n}' \bullet \mathtt{acc} \mapsto \mathtt{a}') \wedge (f(\mathtt{n}') \times \mathtt{a}' = f(N))\}$ ((Read) and (Conj))
```
      if n' == 0 then ret a'
```
$\{\mathsf{res},\ h \mid (h = \mathtt{n} \mapsto 0 \bullet \mathtt{acc} \mapsto f(N)) \wedge (\mathsf{res} = f(N))\}$ (defn. f, (=) and (Return))
$\{\mathsf{res},\ h \mid F_{inv}(\mathtt{n}, \mathtt{acc}, N, h) \wedge (\mathsf{res} = f(N))\}$ (defn. of F_{inv})
```
      else
```

$\{h \mid (h = \mathtt{n} \mapsto \mathtt{n}' \bullet \mathtt{acc} \mapsto \mathtt{a}') \wedge (f(\mathtt{n}') \times \mathtt{a}' = f(N))\}$ (by (Cond))

```
          acc ::= a' * n';;
```

$\{h \mid (h = \mathtt{n} \mapsto \mathtt{n}' \bullet \mathtt{acc} \mapsto \mathtt{a}' \times \mathtt{n}') \wedge (f(\mathtt{n}') \times \mathtt{a}' = f(N))\}$ (by (Write))

```
          n   ::= n' - 1;;
```

$\{h \mid (h = \mathtt{n} \mapsto \mathtt{n}' - 1 \bullet \mathtt{acc} \mapsto \mathtt{a}' \times \mathtt{n}') \wedge (f(\mathtt{n}') \times \mathtt{a}' = f(N))\}$ (by (Write))

$\{h \mid (h = \mathtt{n} \mapsto \mathtt{n}' - 1 \bullet \mathtt{acc} \mapsto \mathtt{a}' \times \mathtt{n}') \wedge (f(\mathtt{n}' - 1) \times \mathtt{a}' \times \mathtt{n}' = f(N))\}$ (by defn. of f)

$\{h \mid F_{inv}(\mathtt{n},\ \mathtt{acc},\ N,\ h)\}$ (by defn. of f)

```
          loop(tt)
```

$\{\mathsf{res},\ h \mid F_{inv}(\mathtt{n},\ \mathtt{acc},\ N,\ h) \wedge (\mathsf{res} = f(N))\}$ (by assumption and (Fix))

```
  )(tt);
```

$\{h \mid F_{inv}(\mathtt{n},\ \mathtt{acc},\ N,\ h) \ \wedge\ (\mathtt{res} = f(N))\}$ (by (Bind))

$\{h \mid (h = \mathtt{n} \mapsto - \bullet \mathtt{acc} \mapsto -) \wedge (\mathtt{res} = f(N))\}$ (by defn. of f)

```
 dealloc(n);;
```

$\{h \mid (h = \mathtt{acc} \mapsto -) \ \wedge\ (\mathtt{res} = f(N))\}$ (by (Dealloc))

```
 dealloc(acc);;
```

$\{h \mid (h = \mathsf{empty}) \wedge (\mathtt{res} = f(N))\}$ (by (Dealloc))

```
 ret res
```

$\{\mathsf{res},\ h \mid (h = \mathsf{empty}) \wedge (\mathsf{res} = f(N))\}$ (by (Ret))

Probably, the most tricky parts of the proof, which indeed require a human prover's insight, are (a) "decomposition" of the loop invariant F_{inv} at the beginning of the loop, when it falls into the components, constraining the values of `n` and `acc` in the heap and (b) the "re-composition" of the same invariant immediately before the recursive call of `loop` in order to ensure its precondition. The latter is possible because of algebraic properties of the factorial function f, namely the fact that for any n, if $n > 0$ then $f(n) \times a = f(n-1) \times n \times a$, the insight we have used in order to "re-distribute" the values between the two pointers, `n` and `acc` so the invariant F_{inv} could be restored. It should be clear by this moment, that, even though the proof is proportional to the size of the program, it has combined some mathematical reasoning with a machinery of consistent rule application, until the postcondition has been reached, which, when done by a human solely, might be an error-prone procedure. Nevertheless, this proof process is very reminiscent to the proofs that we have seen so far in Coq, when one gradually applies the lemmas, assumptions and performs rewritings until the final goal is proved. This is why using Coq seems like a good idea to mechanize the process of proofs in separation logic, so one can be sure that there is nothing missed during the reasoning process and the specification is certainly correct. Employing Coq for this purpose is indeed our ultimate goal and the topic of this chapter. However, before we reach that point, let us recall that in a nutshell Coq is in fact a *functional* programming language and make yet another short detour to the world of pure functional programming, to see how effects might be specified by means of *types*.

8.3 Specifying effectful computations using types

In imperative programs there is a significant distinction between *expressions* and *programs* (or *commands*). While the former ones are *pure*, i.e., will always evaluate to the same result, by which they can be safely replaced, the latter ones are *effectful*, as their

result and the ultimate outcome may produce some irreversible effect (e.g., mutating references, throwing exceptions, performing output or reading from input), which one should account for. Hoare logics, and, in particular, separation logic focus on specifying effectful programs taking expressions for granted and assuming their referential transparency, which makes it not entirely straightforward to embed such reasoning into the purely functional setting of the Coq framework. It has been a long-standing problem for the functional programming community to reconcile the *pure* expressions, enjoying referential transparency, with effectful computations, until Eugenio Moggi suggested to use the mechanism of *monads* to separate the *results* of computations from the possible *effects* they can produce [40], and Philip Wadler popularized this idea with a large number of examples [63], as it was adopted in the same time by the Haskell programming language. There is a countless number of tutorials written and available on the Web that are targeted to help building the intuition about the "monad magic". Although grasping some essence of monadic computations is desirable for understanding how verification of the imperative programs can be structured in Coq, providing the reader with yet another "monad tutorial" is not the task of this course. Luckily, in order to proceed to the verification in separation logic, which is the topic of this chapter, we need only very basic intuition on what monads are, and how they are typically used to capture the essence of computations and their effects.

8.3.1 On monads and computations

While presenting rules for Hoare and separation logic, we have seen a number of operators, allowing to construct larger programs from smaller ones: conditionals, loops, binding, etc. However, only two of the connectives are inherent to imperative programming and make it distinct from the programming with pure functions:

- *Binding* (i.e., a program of the form $x \leftarrow c_1; c_2$) is a way to specify that the effect of the computation c_1 takes place strictly *before* the computation c_2 is executed. Following its name this program constructor also performs binding of the (pure) result of the first computation, so it can be substituted to all occurrences of x in the second command, c_2. In this sense, binding is different from expressions of the form `let` $x = e1$ `in` $e2$, omnipresent in functional programs, as the latter ones might allow for both strict and lazy evaluation of the right-hand side expression $e1$ depending on a semantics of the language (e.g., call-by-value in Standard ML vs. call-by-need in Haskell). This flexibility does not affect the result of a pure program (modulo divergence), since $e1$ and $e2$ are expressions, and, hence, are pure. However, in the case of computations, the order should be fixed and this is what the binding construct serves for.

- *Returning* a value is a command constructor (which we typeset as ret), which allows one to embed a pure expression into the realm of computations. Again, intuitively, this is explained by the fact that expressions and commands should be distinguished semantically,[9] but sometimes an expression should be treated as a command (with

[9] Although some mainstream languages prefer to blur the distinction between commands and expressions in order to save on syntax [46], at the price of losing the ability to differentiate statically between the effectful and pure code.

a trivial effect or none of it at all), whose result is the very same expression.

These two connectives, allowing one to construct the programs by means of binding and embedding expressions into them are captured precisely by the `Monad` interface, expressed, for instance, in Haskell via the following type class:

```
class Monad m where
    (>>=)            :: m a -> (a -> m b) -> m b
    return           :: a -> m a
```

The signature specifies that each instance of `Monad m` is parametrized by one type and requires two functions to be implemented. The `>>=` function is pronounced as *bind* and describes how a particular monad instance *combines* two computations, such that the second one, whose type is `m b`, may depend on the value of result of the first one, whose type is `m a`. The result of the overall computation is then the one of the second component, namely, `m b`. The function `return` specifies how to provide a "default" value for an effectful computation, i.e., how to "pair" a value of type `a` with an "effect" entity in order to receive an element of `m a`. In this specification, the type `m` serves as a generic placeholder of the effect, whose nature is captured by the monad. As it has been already pointed out, such effect might be a mutable state, exceptions, explicit control, concurrency or input/output (all captured by specific instances of monads in Haskell [48]), as well as the fact of program non-termination (i.e., *divergence*), which Haskell deliberately does not capture. In a more informal language, the monadic type `m` indicates that in the program "something fishy is going on, besides the result being computed", so this type serves as a mechanism, which is used by the type checker to make sure that only programs with the *same* effect are composed together by means of binding (hence the type of the bind operator in the `Monad` type class). This is an important insight, which will be directly used in the design of the verification methodology of imperative programs using dependent types, as we will see in Section 8.4.

8.3.2 Monadic do-notation

Since composing effectful/monadic computations is a very common operation in Haskell, the language provides a convenient `do`-notation to write programs in a monadic style, such that the invocation of the bind function in the expression of the form `c1 >>= (\x -> c2)`, where `x` might occur in `c2`, can be written as `{do x <- c1; c2}`. For example, the program below is composed of several computations within the `IO` monad, which indicates that the possible effect of the program, which has `IO` in its type, can be reading from input or writing into the output stream [49].

```
main = do putStrLn "Enter a character"
          c <- getChar
          putStrLn $ "\nThe character was: " ++ [c]
          return ()
```

The computations involved in the program, are represented, in particular, by the Haskell commands (i.e., monadically-typed function call) `putStrLn` "*Enter a character*", which prints a string to the output stream, and the call to `getChar`, which reads a caracter from the input stream. All these computations are bound using the `<-` syntax and `do`-notation, and the last one returns an embedded unit value `()`, so the type of the whole program `main` is inferred to be `IO ()`.

8.4 Elements of Hoare Type Theory

At this point we have acquired a number of important insights that should lead us to the idea of implementing verification of effectful imperative programs in Coq:

- Hoare specifications in separation logic behave like types of the computations they specify, which is witnessed by the rules of weakening (Conseq), function and application specification inference (Hyp) and (App) and recursive functions (Fix) (Section 8.2.1). Moreover, since pre- and postconditions can depend on the values of logical universally-quantified variables as well as on the values of the command's arguments, Hoare-style specs are in fact instances of *dependent* types.

- Hoare triples in separation logic specify *effectful* computations that are composed using the *binding* mechanism, with pure expressions being injected into them by means of "wrapping" them with a ret operator. This makes Hoare triples behave exactly like instances of *monads* from functional programming, whose composition is described by, e.g., the `Monad` type class from Haskell.

- Effectful computations can take effects, which should be accounted for in their specifications. The effects (or observation of an effectful state) are due to some dedicated operations, such as *pointer assignment*, *pointer reading*, *allocation* or *deallocation*. These operations come with dedicated specifications, similarly to how operations `putStrLn` and `getChar` in Haskell are typed with respect to the `IO` monad, whose state they modify.

- Another important effect, which has no explicit handling in the mainstream programming languages like Haskell, but should be dealt with in the context of pure, strongly-normalizing language of Coq, is *divergence*. We cannot allow one to have potentially non-terminating computations as expressions in Coq (i.e., those implemented by means of the general recursion operator fix from Section 8.2.2), but we can afford having a monadic type of computations such that they might possibly diverge *if* they are executed (and, even though, they will not be executed within Coq, they can still be type-checked, and, hence, verified). Therefore, monadic encoding of the fixpoint operator provides a way to escape the termination-checking conundrum and encode nonterminating programs in Coq.

All these observation resulted in a series of works on *Hoare Type Theory* (or just HTT), which defines a notion of an *indexed Hoare monad* (or, *Hoare type*) as a mechanism to encode Hoare-style specifications as dependent types and reduce the verification of effectful progress to proving propositions in Coq [41–43]. In the rest of this chapter we will consider a number of important concepts of HTT, so the necessary modules should be imported from the library folder *htt*, which contains the compiled files (see Section 1.3.3 for the instructions on obtaining and building HTT from the sources).

```
From mathcomp
Require Import ssreflect ssrbool ssrnat eqtype seq ssrfun.
From fcsl
Require Import prelude pred pcm unionmap heap.
From HTT
Require Import stmod stsep stlog stlogR.

Module HTT.

Set Implicit Arguments.
Unset Strict Implicit.
Unset Printing Implicit Defensive.
```

8.4.1 The Hoare monad

The Hoare monad (also dubbed as Hoare type), which is a type of result-returning effectful computations with pre- and postconditions is represented in HTT by the type *STsep*, which is, in fact, just a notation for a more general but less tractable type *STspec*, whose details we do not present here, as they are quite technical and are not necessary in order to verify programs in HTT.[10] The Hoare type is usually specified using the HTT-provided notation as {*x1 x2 ...*}, *STsep* (*p*, *q*), where *p* and *q* are the predicates, corresponding to the pre and postcondition with *p* being of type heap → Prop and *q* of type *A* → heap → Prop, such that *A* is the type of the result of the command being specified. The identifiers *x1*, *x2* etc. bind the logical variables that are assumed to be universally quantified and can appear freely in *p* and *q*, similarly to the free variables in the specifications in Hoare logics (Section 8.1). For example, the `alloc` function has the following (simplified compared to the original one) small footprint specification in the *STsep*-notation:

```
alloc : ∀ (A : Type) (v : A),
          STsep (fun h ⇒ h = Unit,
                 [vfun (res : ptr) h ⇒ h = res :-> v])
```

That is, alloc is a procedure, which starts in an empty heap Unit and whose argument *v* of type *A* becomes referenced by the pointer (which is also the alloc's result) in the resulting singleton-pointer heap. The notation *x* `:->` *y* corresponds to the points-to assertion $x \mapsto y$ in the mathematical representation of separation logic, and [`vfun` *x* ⇒ ...] notation accounts for the fact that the computation can throw an exception [42], the possibility we do not discuss in this course.

[10] A curious reader can take a look at the definitions in the module stmod of the HTT library.

8.4.2 Structuring program verification in HTT

Let us now consider how the examples from Section 8.2 can be given specifications and verified in Coq. The program on page 118, which modifies a pointer x and keeps a different pointer y intact can be given the following spec:

```
Program Definition alter_x A (x : ptr) (v : A):
  {y (Y : nat)},
  STsep (fun h ⇒ exists B (w : B), h = x :-> w \+ y :-> Y,
         [vfun (_: unit) h ⇒ h = x :-> v \+ y :-> Y]) :=
  Do (x ::= v).
```

The Coq command `Program Definition` is similar to the standard definition `Definition` except for the fact that it allows the expression being defined to have a type, some of whose components haven't yet been type-checked and remain to be filled by the programmer, similarly to Agda's incremental development [58]. That is, based on the expression itself (*Do* $(x ::= v)$), Coq will infer *the most general type* that the expression can be allowed to have, and then it becomes a programmer's *obligation* to show that the declared type is actually a specialization of the inferred type. In the context of HTT, the type, inferred by Coq based on the definition, can be seen as a specification with the *weakest pre* and *strongest postconditions*, which can then be weakened via the (Conseq) rule. The program itself is wrapped into the *Do*-notation, which is provided by the HTT library and indicates that the computations inside always deal with the *STsep* type, similar to the Haskell's treatment of `do`-notation. The type of the program `alter_x` is specified explicitly via the *STsep*-notation. There are two logical variables: the pointer y and the value Y of type **nat**, which is referenced by y. The precondition states the existence of some type B and value w, such that x points to it. The postcondition specifies that the result is of type **unit** (and, therefore, is unconstrained), and the content of the pointer x became v, while the content of the pointer y remained unchanged. Notice that we make explicit use of the PCM notation (Section 7.1) for the empty heap, which is paraphrased as `Unit` and for the disjoint union of heaps, which is expressed through the join operator `\+`. After stating the definition, Coq generates a series of obligations to prove in order to establish the defined program well-typed with respect to the stated type.

```
alter_x has type-checked, generating 1 obligation(s)
Solving obligations automatically...
1 obligation remaining
Obligation 1 of alter_x:
∀ (A : Type) (x : ptr) (v : A),
conseq (x ::= v)
  (logvar
     (fun y : ptr ⇒
      logvar
        (fun Y : nat ⇒
         binarify
           (fun h : heap ⇒ exists (B : Type) (w : B), h = x :-> w \+ y :-> Y)
           [vfun _ h ⇒ h = x :-> v \+ y :-> Y]))).
```

The statement looks rather convoluted due to a number of type definitions and no-

tations used and essentially postulates that from the proposition, corresponding to the specification inferred by Coq from the program definition, we should be able to prove the specification that we have declared explicitly. Instead of explaining each component of the goal, we will proceed directly to the proof and will build the necessary intuition as we go. The proof mode for each of the remaining obligations is activated by the Vernacular command `Next Obligation`, which automatically moves some of the assumptions to the context.

```
Next Obligation.
```

```
A : Type
x : ptr
v : A
============================
 conseq (x ::= v)
   (logvar
      (fun y : ptr ⇒
       logvar
         (fun Y : nat ⇒
          binarify
            (fun h : heap ⇒
             exists (B : Type) (w : B), h = x :-> w \+ y :-> Y)
            [vfun _ h ⇒ h = x :-> v \+ y :-> Y])))
```

A usual first step in every HTT proof, which deals with a spec with logical variables is to "pull them out", so they would just become simple assumptions in the goal, allowing one to get rid of the *logvar* and *binarify* calls in the goal.[11] This is what is done by applying the lemma ghR to the goal.

```
apply: ghR.
```

```
A : Type
x : ptr
v : A
============================
 ∀ (i : heap) (x0 : ptr × nat),
 (exists (B : Type) (w : B), i = x :-> w \+ x0.1 :-> x0.2) →
 valid i → verify i (x ::= v) [vfun _ h ⇒ h = x :-> v \+ x0.1 :-> x0.2]
```

We can now move a number of assumptions, arising from the "brushed" specification, to the context, along with some rewriting by equality and simplifications.

[11] In fact, the proper handling of the logical variables is surprisingly tricky in a type-based encoding, which is what HTT delivers. It is due to the fact that the *same* variables can appear in both pre- and postconditions. Earlier implementations of HTT used *binary* postconditions for this purpose [42, 43], which was a cause of some code duplication in specifications and made the spec look differently from those that someone familiar with the standard Hoare logic would expect. Current implementation uses an encoding with recursive notations to circumvent the code duplication problem. This encoding is a source of the observed occurrences of *logvar* and *binarify* definitions in the goal.

```
move⇒h1 [y Y][B][w]->{h1} _ /=.
```

```
  ...
  B : Type
  w : B
  ============================
   verify (x :-> w \+ y :-> Y) (x ::= v) [vfun _ h ⇒ h = x :-> v \+ y :-> Y]
```

The resulting goal is stated using the verify-notation, which means that in this particular case, in the heap of the shape x `:->` w `\+` y `:->` Y we need to be able to prove that the result and the produced heap of the command x `::=` v satisfy the predicate [`vfun _` $h \Rightarrow h = x$ `:->` v `\+` y `:->` Y]. This goal can be proved using one of the numerous verify-lemmas that HTT provides (try executing `Search _ (verify _ _ _)` to see the full list), however in this particular case the program and the goal are so simple and are obviously correct that the statement can be proved by means of proof automation, implemented in HTT by a brute-force tactic heval, which just tries a number of verify-lemmas applicable in this case modulo the shape of the heap.

```
by heval.
Qed.
```

8.4.3 Verifying the factorial procedure mechanically

Proving an assignment for two non-aliased pointers was a simple exercise, so now we can proceed to a more interesting program, which features loops and conditional expressions, namely, imperative implementation of the factorial function. Our specification and verification process will follow precisely the story of Section 8.2.3. We start by defining the factorial in the most declarative way—as a pure recursive function in Coq itself.

```
Fixpoint fact_pure n := if n is n'.+1 then n × (fact_pure n') else 1.
```

Next, we define the loop invariant fact_inv, which constraints the heap shape and the values of the involved pointers, n and acc, mimicking precisely the definition of F_{inv}:

```
Definition fact_inv (n acc : ptr) (N : nat) h : Prop :=
  exists n' a': nat,
  [/\ h = n :-> n' \+ acc :-> a' &
      (fact_pure n') × a' = fact_pure N].
```

To show how separation logic, in general and its particular implementation in HTT, allows one to build the reasoning *compositionally* (i.e., by building the proofs about large programs from the facts about their components), we will first provide and prove a specification for the internal factorial loop, which, in fact, performs all of the interesting computations, so the rest of the "main" function only takes care of allocation/deallocation of the pointers n and acc. The loop will be just a function, taking an argument of the type unit and ensuring the invariant fact_inv in its pre- and postcondition, as defined by the following type fact_tp, parametrized by the pointers n and acc.

```
Definition fact_tp n acc :=
  unit → {N},
     STsep (fact_inv n acc N,
```

$$[\texttt{vfun}\ (res : nat)\ h \Rightarrow fact\_inv\ n\ acc\ N\ h \wedge res = fact\_pure\ N]).$$

The type fact_tp ensures additionally that the resulting value is in fact a factorial of N, which is expressed by the conjunct $res =$ fact_pure N. The definition of the factorial "accumulator" loop is then represented as a recursive function, taking as arguments the two pointers, n and acc, and also a unit value. The body of the function is defined using the monadic fixpoint operator `Fix`, whose semantics is similar to the semantics of the classical *Y-combinator*, defined usually by the equation $Y\ f = f\ (Y\ f)$, where f is a fixpoint operator argument that should be thought of as a recursive function being defined. Similarly, the fixpoint operator `Fix`, provided by HTT, takes as arguments a function, which is going to be called recursively (*loop*, in this case), its argument and *body*. The named function (i.e., *loop*) can be then called from the body recursively. In the similar spirit, one can define nested loops in HTT as nested calls of the fixpoint operator.

```
Program Definition fact_acc (n acc : ptr): fact_tp n acc :=
  Fix (fun (loop : fact_tp n acc) (_ : unit) ⇒
    Do (a1 <-- read nat acc;
        n' <-- read nat n;
        if n' == 0 then ret a1
        else acc ::= a1 × n';;
             n ::= n' - 1;;
             loop tt)).
```

The body of the accumulator loop function reproduces precisely the factorial implementation in pseudocode from page 121. It first reads the values of the pointers *acc* and n into the local variables *a1* and *n'*. Notice that the binding of the local immutable variables is represented by the $<--$ notation, which corresponds to the *bind* operation of the Hoare monad *STsep*. The function then uses Coq's standard conditional operator and returns a value of *a1* if *n'* is zero using the monadic ret operator. In the case of `else`-branch, the new values are written to the pointers *acc* and n, after which the function recurs. Stating the looping function like this leaves us with one obligation to prove.

```
Next Obligation.
```

As in the previous example, we start by transforming the goal, so the logical variable N, coming from the specification of fact_tp would be exposed as an assumption. We immediately move it to the context along with the initial heap i.

```
apply: ghR⇒i N.
```

```
...
i : heap
N : nat
============================
 fact_inv n acc N i →
 valid i →
 verify i
   (a1 <-- ! acc;
    n' <-- ! n;
```

```
  (if n' == 0 then ret a1 else acc ::= a1 × n';; n ::= n' - 1;; loop tt))
  [vfun res h ⇒ fact_inv n acc N h ∧ res = fact_pure N]
```

We next case-analyse on the top assumption with the invariant fact_inv to acquire the equality describing the shape of the heap i. We then rewrite i in place and move a number of hypotheses to the context.

```
case⇒n' [a'][->{i}] Hi _.
```

Now the goal has the shape verify (n :-> n' \+ acc :-> a') ..., which is suitable to be hit with the automation by means of the heval tactic, progressing the goal to the state when we should reason about the conditional operator.

```
heval.
```

```
...
n' : nat
a' : nat
Hi : fact_pure n' × a' = fact_pure N
============================
 verify (n :-> n' \+ acc :-> a')
   (if n' == 0 then ret a' else acc ::= a' × n';; n ::= n' - 1;; loop tt)
   [vfun res h ⇒ fact_inv n acc N h ∧ res = fact_pure N]
```

The goal, containing a use of the conditional operator, is natural to be proved on case analysis on the condition $n' == 0$.

```
case X: (n' == 0).
```

Now, the first goal has the form

```
...
Hi : fact_pure n' × a' = fact_pure N
X : (n' == 0) = true
============================
 verify (n :-> n' \+ acc :-> a') (ret a')
   [vfun res h ⇒ fact_inv n acc N h ∧ res = fact_pure N]
```

To prove it, we will need one of the numerous *val*-lemmas, delivered as a part of HTT libraries and directly corresponding to the rules of separation logic (Section 8.2.1). The general recipe on acquiring intuition for the lemmas applicable for each particular verify-goal is to make use of Ssreflect's Search machinery. For instance, in this particular case, given that the command to be verified (i.e., the second argument of verify) is ret a', let us try the following query.

```
Search _ (verify _ _ _) (ret _).
```

The request results report, in particular, on the following lemma found:

```
val_ret
  ∀ (A : Type) (v : A) (i : heapPCM) (r : cont A),
  (valid i → r (Val v) i) → verify i (ret v) r
```

The lemma has a statement in its conclusion, which seems like it can be unified with our goal, so we proceed by applying it.

- apply: *val_ret*=>/= _.

The remaining part of the proof of this goal has absolutely nothing to do with program verification and separation logic and accounts to combining a number of arithmetical facts in the goal via the hypotheses *Hi* and *X*. We proceed by first turning boolean equality in *X* into propositional via the view eqP and then substituting all occurrences of *n'* in the goal and other assumptions via Coq's tactic subst. The rest of the proof is by providing existential witnesses and rewriting $1 \times a'$ to *a'* in *Hi*.

```
move/eqP: X⇒Z; subst n'.
split; first by exists 0, a'=>//.
by rewrite mul1n in Hi.
```

The second goal requires satisfying the specification of a sequence of assignments, which can be done automatically using the heval tactic.

```
heval.
```

```
loop : fact_tp n acc
...
Hi : fact_pure n' × a' = fact_pure N
X : (n' == 0) = false
============================
 verify (n :-> (n' - 1) \+ acc :-> (a' × n')) (loop tt)
   [vfun res h ⇒ fact_inv n acc N h ∧ res = fact_pure N]
```

The next step is somewhat less obvious, as we need to prove the specification of the recursive call to *loop*, whose spec is also stored in our assumption context. Before we can apply a lemma, which is an analogue of the (App), we need to *instantiate* the logical variables of *loop*'s specification (which is described by the type fact_tp). The spec fact_tp features only one logical variable, namely *N*, so we provide it using the HTT lemma gh_ex.[12]

```
apply: (gh_ex N).
```

Now to verify the call to *loop*, we can apply the lemma val_doR, corresponding to the rule (App), which will replace the goal by the precondition from the spec fact_tp *n acc*. In HTT there are several lemmas tackling this kind of a goal, all different in the way they treat the postconditions, so in other cases it is recommended to run Search "val_do" to see the full list and chose the most appropriate one.

```
apply: val_doR=>// _.
```

```
...
Hi : fact_pure n' × a' = fact_pure N
X : (n' == 0) = false
============================
 fact_inv n acc N (n :-> (n' - 1) \+ acc :-> (a' × n'))
```

[12] In a case of several logical variables, the lemma should have been applied the corresponding number of times with appropriate arguments.

As in the case of the previous goal, the remaining proof is focused on proving a statement about a heap and natural numbers, so we just present its proof below without elaborating on the details, as they are standard and mostly appeal to propositional reasoning (Chapter 3) and rewriting by lemmas from Ssreflect's ssrnat module.

```
exists (n'-1), (a' × n'); split=>//=.
rewrite -Hi=>{Hi}; rewrite [a' × _]mulnC mulnA [_ × n']mulnC.
by case: n' X=>//= n' _; rewrite subn1 -pred_Sn.
Qed.
```

We can now implement the main body of the factorial function, which allocates the necessary pointers, calls the accumulator loop and then frees the memory.

```
Program Definition fact (N : nat) :
  STsep ([Pred h | h = Unit],
         [vfun res h ⇒ res = fact_pure N ∧ h = Unit]) :=
  Do (n <-- alloc N;
      acc <-- alloc 1;
      res <-- fact_acc n acc tt;
      dealloc n;;
      dealloc acc;;
      ret res).
```

The specification of fact explicitly states that its execution starts and terminates in the empty heap; it also constraints its result to be a factorial of N.

```
Next Obligation.
```

Since the spec of fact does not have any logical variables (its postcondition only mentions its argument N), there is no need to make use of the ghR lemma. However, the current goal is somewhat obscure, so to clarify it let us unfold the definition of conseq (which simply states that the consequence between the inferred type of the program and the stated spec should be proved) and simplify the goal.

```
rewrite /conseq =>/=.
```

```
  N : nat
  ============================
   ∀ i : heap,
   i = Unit →
   verify i
     (n <-- alloc N;
      acc <-- alloc 1;
      res <-- fact_acc n acc tt; dealloc n;; dealloc acc;; ret res)
     (fun (y : ans nat) (m : heap) ⇒
      i = Unit → [vfun res h ⇒ res = fact_pure N ∧ h = Unit] y m)
```

Next, we can rewrite the equality on the heap (which is Unit) and proceed by two runs of the heval tactic, which will take care of the allocated pointers yielding new assumptions n and acc, arising from the implicit application of the (Bind) rule.

```
move⇒_ →.
```

heval⇒n; heval⇒acc; `rewrite` $joinC$ $unitR$.

We have now came to the point when the function fact_acc, which we have previously verified, is going to be invoked, so we need to make use of what corresponds to the rule (App) again. In this case, however, the tactic val_doR will not work immediately, so we will first need to reduce the program to be verified from the binding command to a mere function call by means of HTT's bnd_seq lemma, which tackles the binding *combined* with a call to a user-defined function, and this is exactly our case. Next, we instantiate the fact_acc specification's logical variable N by applying gh_ex and proceed with the application of val_doR.

`apply`: bnd_seq=>/=; `apply`: (gh_ex N); `apply`: val_doR=>//.

The first of the resulting two goals is an obligation arising from the need to prove fact_acc's precondition.

- `by exists` N, 1; `rewrite` $muln1$.

The second goal is the remainder of the program's body, which performs deallocation, so the proof for it is accomplished mostly by applying heval tactic.

`by move`⇒_ _ [[n'][a'][->] _ ->] _; heval.
`Qed`.

Exercise 8.1 (Swapping two values)**.** Implement in HTT a function that takes as arguments two pointers, x and y, which point to natural numbers, and swaps their values. Reflect this effect in the function's specification and verify it.
Hint: Instead of reading the value of a pointer into a variable t using the $t <-- !p$ notation, you might need to specify the *type* of the expected value explicitly by using the "de-sugared" version of the command $t <-- \mathsf{read}\ T\ p$, where T is the expected type. This way, the proof will be more straightforward.

Exercise 8.2. Try to redo the exercise 8.1 *without* using the automation provided by the heval tactic. The goal of this exercise is to explore the library of HTT lemmas, mimicking the rules of the separation logic. You can always display the whole list of the available lemmas by running the command `Search _` (verify _ _ _) and then refine the query for specific programs (e.g., read or write).

Exercise 8.3 (Fibonacci numbers)**.** Figure 8.1 presents the pseudocode listing of an efficient imperative implementation of the function `fib` that computes the Nth Fibonacci number. Your task will be to prove its correctness with respect to the "pure" function fib_pure (which you should define in plain Coq) as well as the fact that it starts and ends in an empty heap.
Hint: What is the loop invariant of the recursive computation defined by means of the *loop* function?
Hint: Try to decompose the reasoning into verification of several code pieces as in the factorial example and then composing them together in the "main" function.

```
fun fib (N : nat): nat = {
  if N == 0 then ret 0
   else if N == 1 then ret 1
   else n <-- alloc 2;
        x <-- alloc 1;
        y <-- alloc 1;
        res <--
          (fix loop (_ : unit).
             n' <-- !n;
             y' <-- !y;
             if n' == N then ret y'
             else tmp <-- !x;
                  x ::= y';;
                  x' <-- !x;
                  y ::= x' + tmp;;
                  n ::= n' + 1;;
                  loop(tt))(tt).
        dealloc n;;
        dealloc x;;
        dealloc y;;
        ret res
}
```

Figure 8.1: An imperative procedure computing the Nth Fibonacci number.

8.5 On shallow and deep embeddings

A noteworthy trait of HTT's approach to verification of effectful programs is its use of *shallow embedding* of the imperative language into the programming language of Coq. In fact, the imperative programs that we have written, e.g., the factorial procedure, are mere Coq programs, written in Coq syntax with a number of HTT-specific notations. Moreover, the Hoare triples, by means of which we have provided the specifications to the heap-manipulating programs are nothing but specific types defined in Coq. This is what makes the way effectful programs encoded *shallow*: the new programming language of imperative programs and their Hoare-style specifications has been defined as a subset of Coq programming language, so most of the Coq's infrastructure for parsing, type-checking, name binding and computations could be reused off the shelf. In particular, shallow embedding made it possible to represent the variables in imperative programs as Coq's variables, make use of Coq's conditional operator and provide specifications to higher-order procedures without going into the need to design a higher-order version of a separation logic first (since the specifications in HTT are just types of monadically-typed expressions). Furthermore, shallow embedding provided us with a benefit of reusing Coq's name binding machinery, so we could avoid the problem of *name capturing* by means using the approach known as *Higher-Order Abstract Syntax* [51], representing immutable variables by Coq's native variables (disguised by the binding notation $<--$). To summarize, shallow embedding is an approach of implementing programming languages

(not necessarily in Coq) characterized by representation of the language of interest (usually called a *domain-specific language* or DSL) as a subset of another general-purpose *host* language, so the programs in the former one are simply the programs in the latter one. The idea of shallow embedding originated in early '60s with the beginning of the era of the Lisp programming language [26], which, thanks to its macro-expansion system, serves as a powerful platform to implement DSLs by means of shallow embedding (such DSLs are sometimes called *internal* or *embedded*). Shallow embedding in the world of practical programming is advocated for a high speed of language prototyping and the ability to re-use most of the host language infrastructure. An alternative approach of implementing and encoding programming languages in general and in Coq in particular is called *deep embedding*, and amounts to the implementation of a language of interest from scratch, essentially, writing its parser, interpreter and type-checker in a general-purpose language. In practice, deep embedding is preferable when the overall performance of the implemented language runtime is of more interest than the speed of DSL implementation, since then a lot of intermediate abstractions, which are artefacts of the host language, can be avoided. In the world of mechanized program verification, both approaches, deep and shallow embedding, have their own strengths and weaknesses. Although implementations of deeply embedded languages and calculi naturally tend to be more verbose, design choices in them are usually simpler to explain and motivate. Moreover, the deep embedding approach makes the problem of name binding to be explicit, so it would be appreciated as an important aspect in the design and reasoning about programming languages [2, 6, 65]. We believe, these are the reasons why this approach is typically chosen as a preferable one when teaching program specification and verification in Coq [53]. Importantly, deep embedding gives the programming language implementor the *full control* over its syntax and semantics.[13] In particular, the expressivity limits of a defined logic or a type system are not limited by expressivity of Coq's (or any other host language's) type system. Deep embedding makes it much more straightforward to reason about *pairs* of programs by means of defining the relations as propositions on pairs of syntactic trees, which are implemented as elements of corresponding datatypes. This point, which we deliberately chose not to discuss in detail in this course, becomes crucial when one needs to reason about the correctness of program transformations and optimizing compilers [1]. In contrast, the choice of shallow embedding, while sparing one the labor of implementing the parser, name binder and type checker, may limit the expressivity of the logical calculus or a type system to be defined. In the case of HTT, for instance, it amounts to the impossibility of specifying programs that store *effectful functions* and their specifications into a heap.[14] In the past decade Coq has been used in a large number of projects targeting formalization of logics and type systems of various programming languages and proving their soundness, with most of them preferring the deep embedding approach to the shallow one. We believe that the explanation of this phenomenon is the fact that it is much more straightforward to define semantics of a deeply-embedded "featherweight" calculus [33] and prove soundness of its type system or program logic, given that it is the *ultimate goal* of the research project. However, in order to use the implemented framework to specify and verify realistic programs, a significant implementation effort is required to extend the deep implementation beyond the "core

[13] This observation is reminiscent to the reasons of using deep embedding in the practical world.
[14] This limitation can be, however, overcome by postulating necessary *axioms* on top of CIC.

language", which makes shallow embedding more preferable in this case—a reason why this way has been chosen by HTT.

8.6 Soundness of Hoare Type Theory

Because of shallow embedding, every valid Coq program is also a valid HTT program. However, as it has been hinted at the beginning of Section 8.4, imperative programs written in HTT cannot be simply executed, as, due to presence of general loops and recursion, they simply may not terminate. At this point, a reader may wonder, what good is verification of programs that cannot be run and what is it that we have verified? To answer this question, let us revise how the *soundness* of a Hoare logic is defined. HTT takes definition of a Hoare triple (or, rather, a Hoare type, since in HTT specs are types) from page 112 literally but implements it not via an operational semantics, i.e., defining how a program *should be run*, but using a denotational semantics [67, Chapter 5], i.e., defining what a program *is*. The HTT library comes with a module `stmod` that defines denotational semantics of HTT commands[15] and Hoare triples, defined as types. Each command is represented by a function, which is sometimes referred to as a *state transformer*, in the sense that it takes a particular heap and transforms it to another heap, also returning some result. The denotational semantics of HTT commands in terms of state-transforming functions makes it also possible to define what is a semantics of a program resulting from the use of the `Fix` operator (Section 8.4.3).[16] The semantics of Hoare types $\{h \mid P(h)\} - \{\mathsf{res}, h \mid Q(\mathsf{res}, h)\}$ is defined as *sets* of state transforming functions, taking a heap satisfying P to the result and heap satisfying Q. Therefore, the semantic account of the verification (which is implemented by means of type-checking in Coq) is checking that semantics of a particular HTT program (i.e., a state-transforming function) lies *within* the semantics of its type as a set. If execution of programs verified in HTT is of interest, it can be implemented by means of *extraction* of HTT commands into programs in an external language, which supports general recursion natively (e.g., Haskell). In fact, such extraction has been implemented in the first release of HTT [42], but was not ported to the latest release.

8.7 Specifying and verifying programs with linked lists

We conclude this chapter with a *tour de force* of separation logic in HTT by considering specification and verification of programs operating with single-linked lists. Unlike the factorial example, an implementation of single-linked lists truly relies on pointers, and specifying such datatypes and programs is an area where separation logic shines. On the surface, a single-linked list can be represented by a pointer, which points to its head.

[15] I.e., monadic values constructed by means of the write/alloc/dealloc/read/return commands and standard Coq connectives, such as conditional expression or pattern matching.

[16] In fact, a standard construction from the domain theory is used, namely, employing Knaster-Tarski theorem on a lattice of monotone functions. This subject is, however, outside of the scope of this course, so we redirect the reader to the relevant literature: Glynn Winskel's book for the theoretical construction [67, Chapters 8–10] or Adam Chlipala's manuscript covering a similar implementation [7, § 7.2].

```
Definition llist (T : Type) := ptr.

Section LList.
Variable T : Type.
Notation llist := (llist T).
```

However, in order to specify and prove interesting facts about imperative lists, similarly to the previous examples, we need to establish a connection between what is stored in a list heap and a purely mathematical sequence of elements. This is done using the *recursive predicate* lseg, which relates two pointers, p and q, pointing correspondingly to the head and to the tail of the list and a logical sequence xs of elements stored in the list.

```
Fixpoint lseg (p q : ptr) (xs : seq T): Pred heap :=
  if xs is x::xt then
    [Pred h | exists r h',
        h = p :-> x \+ (p .+ 1 :-> r \+ h') ∧ h' \In lseg r q xt]
  else [Pred h | p = q ∧ h = Unit].
```

The notation [Pred h | ...] is just an abbreviation for a function of type heap → Prop, where h is assumed to be of the type heap. The notation h \In f is a synonym for f h assuming f is a predicate of type heap → Prop. The following lemma lseg_null states a fact, which is almost obvious: given that the heap h, corresponding to a linked list, is a valid one (according to its notion of validity as a PCM) and the head pointer of a list structure is null, then its tail pointer is null as well, and the overall list is empty.

```
Lemma lseg_null xs q h :
          valid h → h \In lseg null q xs →
          [/\ q = null, xs = [::] & h = Unit].
Proof.
case: xs=>[|x xs] D /= H; first by case: H=><- →.
case: H D⇒r [h'][->] _.
```

```
  ...
  r : ptr
  h' : heap
  ============================
   valid (null :-> x \+ (null.+1 :-> r \+ h')) →
   [/\ q = null, x :: xs = [::] & null :-> x \+ (null.+1 :-> r \+ h') = Unit]
```

In the process of the proof we are forced to use the validity of a heap in order to derive a contradiction. In the case of heap's validity, one of the requirements is that every pointer in it is not null. We can make it explicit by rewriting the top assumption with one of the numerous HTT lemmas about heap validity (use the Search machinery to find the others).

```
rewrite validPtUn.
```

```
  ...
  ============================
   [&& null != null, valid (null.+1 :-> r \+ h')
```

```
    & null \notin dom (null.+1 :-> r \+ h')] →
  [/\ q = null, x :: xs = [::] & null :-> x \+ (null.+1 :-> r \+ h') = Unit]
```

The conjunct null != null in the top assumption is enough to complete the proof by implicit discrimination.

```
done.
Qed.
```

We can now define a particular case of linked lists—*null-terminating* lists and prove the specification of a simple insertion program, which allocates a new memory cell for an element x and makes it to be a new head of a list pointed by p. The allocation is performed via the primitive allocb, which allocates a number of subsequent heap pointers (two in this case, as defined by its second argument) and sets all of them to point to the value provided.

```
Definition lseq p := lseq p null.

Program Definition insert p x :
  {xs}, STsep (lseq p xs, [vfun y ⇒ lseq y (x::xs)]) :=
  Do (q <-- allocb p 2;
      q ::= x;;
      ret q).
Next Obligation.
apply: ghR⇒i xs H _; heval⇒x1; rewrite unitR -joinA; heval.
Qed.
```

Next, we are going to give a specification to the list "beheading"—removing the head element of a list. For this, we will need a couple of auxiliary lemmas involving the list heap predicate *lseq_neq*. The first one, lseq_null is just a specialization of the previously proved lseg_null.

```
Lemma lseq_null xs h : valid h → h \In lseq null xs → xs = [::] ∧ h = Unit.
Proof. by move⇒D; case/(lseg_null D)=>_ →. Qed.
```

The next lemma, lseq_pos, states that if p is a head of a linked list, defined by a heap h, is not null, then it can be "beheaded". That is, there will exist a head value x, a "next" r and a residual heap *h'*, such that the heap *h'* corresponds to the list's tail, which is expressed by Ssreflect's behead function.

```
Lemma lseq_pos xs p h :
        p != null → h \In lseq p xs →
        exists x r h',
          [/\ xs = x :: behead xs,
              p :-> x \+ (p .+ 1 :-> r \+ h') = h & h' \In lseq r (behead xs)].
Proof.
case: xs=>[|x xs] /= H []; first by move⇒E; rewrite E eq_refl in H.
by move⇒y [h'][->] H1; heval.
Qed.
```

We can finally define and specify the HTT procedure remove, which removes the current head of the list and returns the pointer to its next element or null if the list is empty.

```
Program Definition
```

```
remove p : {xs}, STsep (lseq p xs, [vfun y ⇒ lseq y (behead xs)]) :=
  Do (if p == null then ret p
      else pnext <-- !(p .+ 1);
           dealloc p;;
           dealloc p .+ 1;;
           ret pnext).
```

The proof is straightforward and employs both lemmas: lseq_null to prove the "null" case and lseq_pos for the case when the list has at least one element.

```
Next Obligation.
apply: ghR⇒i xs H V; case: ifP H⇒H1.
- by rewrite (eqP H1); case/(lseq_null V)=>->->; heval.
case/(lseq_pos (negbT H1))=>x [q][h][->] ← /= H2.
by heval; rewrite 2!unitL.
Qed.
```

Exercise 8.4. Define and verify function remove_val which is similar to remove, but also returns the *value* of the last "head" of the list before removal, in addition to the "next" pointer. Use Coq's *option* type to account for the possibility of an empty list in the result.

```
End LList.
```

Exercise 8.5 (Imperative in-place map)**.** Define, specify and verify the imperative higher-order function list_map that takes as arguments two types, S and T, a function $f : T \to S$ and a head p of a single-linked list, described by a predicate lseq, and changes the list in place by applying f to each of its elements, while preserving the list's structure. The specification should reflect the fact that the new "logical" contents of the single-linked list are an f map-image of the old content.
Hint: The lemmas lseq_null and lseq_pos, proved previously, might be useful in the proof of the established specification.
Hint: A tail-recursive call can be verified via HTT's val_do lemma, reminiscent to the rule (App). However, the heap it operates with should be "massaged" appropriately via PCM's lemmas joinC and joinA.

Exercise 8.6 (In-place list reversal)**.** Let us define the following auxiliary predicates, where shape_rev splits the heap into two disjoint linked lists (by means of the separating conjunction #).

```
Definition shape_rev T p s := [Pred h | h \In @lseq T p.1 s.1 # @lseq T p.2 s.2].
```

Then the in-place list reversal is implemented by means of the recursive function reverse with a loop invariant expressed using the type revT.

```
Definition revT T : Type :=
  ∀ p, {ps}, STsep (@shape_rev T p ps, [vfun y ⇒ lseq y (rev ps.1 ++ ps.2)]).

Program Definition
reverse T p : {xs}, STsep (@lseq T p xs, [vfun y ⇒ lseq y (rev xs)]) :=
```

```
Do (let: reverse := Fix (fun (reverse : revT T) p ⇒
                         Do (if p.1 == null then ret p.2
                             else xnext <-- !p.1 .+ 1;
                                  p.1 .+ 1 ::= p.2;;
                                  reverse (xnext, p.1)))
    in reverse (p, null)).
```

We invite the reader to conduct the verification of reverse, proving that it satisfies the given specification.

Hint: It might be a good idea to make use of the previously proved lemmas lseq_null and lseq_pos.

Hint: Be careful with the logical values of variables passed to the gh_ex lemma before verifying a recursive call of reverse.

Hint: A verification goal to a function defined via Fix can be reduced via the val_doR lemma or similar ones.

Hint: The shape_rev predicate is in fact an existential in disguise: it can be proved by providing appropriate witnesses.

Hint: Rewriting rev_cons, cat_rcons and cats0 from the seq library will be useful for establishing equalities between lists.

9 Conclusion

The goal of this course was to introduce a reader with a background in programming and abstract algebra to interactive theorem proving in the Coq proof assistant.

Starting from the concepts, familiar from the world of functional programming, such as higher-order functions, algebraic datatypes and recursion, we have first considered Coq as a programming language in **Chapter 2**. The programming language intuition helped us to move further into the realm of propositional logic and comprehend the way of encoding and proving propositions constructively in **Chapter 3**. At that point a number of familiar logical connectives came in the new light of Curry-Howard correspondence with respect to the familiar datatypes. Introducing universal and existential quantification, though, required to appeal to the dependently-typed part of Coq as a programming language, which moved us beyond a simple propositional logic, so we could make statements over arbitrary entities, not just propositions. At the same point we had the first encounter with Coq's proof construction machinery. To unleash the full power of the mathematical reasoning, in **Chapter 4** we learned about the way equality is defined in Coq and how it is used for proofs by rewriting. In the process we have learned that equality is just one way to encode a rewriting principle and seen how custom rewriting principles can be encoded in Coq. It turned out that one of the most useful rewriting principles is the ability to "switch" in the reasoning between the constructive and computable formulation of decidable propositions—a trick that working mathematicians perform on the fly in their minds. In **Chapter 5**, we have seen how the same switch can be implemented seamlessly in Coq using the boolean reflection machinery. With the introduction of boolean reflection, our journey in the world of interactive theorem proving took a path, paved by Gonthier's et al.'s Ssreflect extension, embracing and leveraging the computational power of Coq as a programming language to the extreme. The motto "let Coq compute a part of the proof for you, since it's a programming language after all!", witnessed by formulation of boolean functions instead of decidable inductive predicates, has been supplied by a number of examples in **Chapter 6**, in which we have also exercised in proofs by induction of different flavours. Mathematics is a science of abstract structures and facts, and formalizing such structures and facts is an important component of rigorous reasoning. In **Chapter 7** we have learned how the concepts of records and type classes, familiar from languages like C and Haskell, can be directly employed, powered by Coq's dependent types, to encode a variety of mathematical structures from the course of abstract algebra. **Chapter 8** brought all of the presented programming and proving concepts together and made them to work in concert to tackle a large case study—specifying and verifying imperative programs in the framework of Hoare Type Theory.

Future directions

I hope that this short and by all means not exhaustive course on mechanized mathematics in Coq was helpful in reconciling the reader's programming expertise and mathematical background in one solid picture. So, what now?

In the author's personal experience, fluency in Coq and the ability to operate with a large vocabulary of facts, lemmas and tactics is a volatile skill, which fades out at an alarming rate without regular practice in proving and mechanized reasoning. On the bright side, this is also a skill, which can fill one with a feeling of excitement from a progressive reasoning process and the rewarding sense of achievement that few human activities bring.

With this respect, it seems natural to advise the reader to pick a project on her own and put it to the rails of machine-assisted proving. Sadly, formalizing things just for the sake of formalization is rarely a pleasant experience, and re-doing someone's results in Coq just to "have them in Coq at any price" is not a glorious goal by itself. What is less obvious is that setting up mathematical reasoning in Coq usually brings some *brand new* insights that usually come from directions no one expected. Such insights might be due to exploding proofs, which are repetitive and full of boilerplate code (seems like a refactoring opportunity in someone's math?) or because of the lack of abstraction in a supposedly abstract concept, which overwhelms its clients with proof obligations, once being applied to something its designer mathematician didn't foresee (a case of leaky abstraction?). Coq combines programming and mathematics in a single framework. I believe, this must be the point, at which several decades of mastering the humanity's programming expertise should pay back and start being useful for producing the genuine *understanding* of formally stated facts and proofs about them.

Bibliography

[1] Andrew W. Appel, Robert Dockins, Aquinas Hobor, Lennart Beringer, Josiah Dodds, Gordon Stewart, Sandrine Blazy, and Xavier Leroy. *Program Logics for Certified Compilers*. Cambridge University Press, 2014 (cit. on pp. 7, 137).

[2] Brian E. Aydemir, Arthur Charguéraud, Benjamin C. Pierce, Randy Pollack, and Stephanie Weirich. "Engineering formal metatheory". In: *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '08)*. San Francisco, California, USA: ACM, 2008, pp. 3–15 (cit. on pp. 7, 137).

[3] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer Verlag, 2004 (cit. on pp. 5, 17, 21, 29, 32, 44, 47).

[4] Richard Bird. *Pearls of Functional Algorithm Design*. Cambridge University Press, 2010 (cit. on p. 111).

[5] Hongxu Cai, Zhong Shao, and Alexander Vaynberg. "Certified self-modifying code". In: *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI '07)*. San Diego, California, USA: ACM, 2007, pp. 66–77 (cit. on p. 7).

[6] Arthur Charguéraud. "The Locally Nameless Representation". In: *Journal of Automated Reasoning* 49.3 (2012), pp. 363–408 (cit. on p. 137).

[7] Adam Chlipala. *Certified Programming with Dependent Types*. The MIT Press, 2013 (cit. on pp. 6–7, 17, 20, 46, 138).

[8] Adam Chlipala. "Mostly-automated verification of low-level programs in computational separation logic". In: *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11)*. San Jose, California, USA: ACM, 2011, pp. 234–245 (cit. on p. 7).

[9] Alonzo Church. "A Formulation of the Simple Theory of Types". In: *The Journal of Symbolic Logic* 5.2 (1940), pp. 56–68 (cit. on p. 30).

[10] The Coq Development Team. *The Coq Proof Assistant – Reference Manual, Version 8.4pl4*. Available at http://coq.inria.fr/refman/. 2015 (cit. on pp. 5, 8, 26, 102).

[11] Thierry Coquand. "An Analysis of Girard's Paradox". In: *Proceedings of the Symposium on Logic in Computer Science*. Cambridge, Massachusetts, USA: IEEE Computer Society, 1986, pp. 227–236 (cit. on p. 48).

[12] Thierry Coquand and Gérard P. Huet. "Constructions: A Higher Order Proof System for Mechanizing Mathematics". In: *Proceedings of European Conference on Computer Algebra (EUROCAL '85), Volume 1: Invited Lectures*. Vol. 203. Lecture Notes in Computer Science. Linz, Austria: Springer, 1985, pp. 151–184 (cit. on p. 48).

[13] Thierry Coquand and Gérard P. Huet. "The Calculus of Constructions". In: *Information and Computation* 76.2/3 (1988), pp. 95–120 (cit. on p. 29).

[14] Haskell B. Curry. "Functionality in combinatory logic". In: *Proceedings of the National Academy of Sciences of the United States of America* 20.11 (1934), pp. 584–590 (cit. on p. 29).

[15] Olivier Danvy. *Three Steps for the CPS Transformation*. Tech. rep. CIS-92-2. Manhattan, Kansas, USA: Kansas State University, 1992 (cit. on p. 120).

[16] Robert Dockins and Aquinas Hobor. "A theory of termination via indirection". In: *Proceedings of the Dagstuhl Seminar on Modelling, Controlling and Reasoning about State*. Vol. 10351. Dagstuhl, Germany, 2010, pp. 166–177 (cit. on p. 113).

[17] Xinyu Feng, Zhong Shao, Alexander Vaynberg, Sen Xiang, and Zhaozhong Ni. "Modular verification of assembly code with stack-based control abstractions". In: *Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation (PLDI '06)*. Ottawa, Ontario, Canada: ACM, 2006, pp. 401–414 (cit. on p. 7).

[18] Robert W. Floyd. "Assigning meanings to programs". In: *Proceedings of the Symposium on Applied Mathematics*. Vol. 19. 1967, pp. 19–31 (cit. on p. 112).

[19] François Garillot. "Generic Proof Tools and Finite Group Theory". PhD thesis. Palaiseau, France: École Polytechnique, 2011 (cit. on p. 107).

[20] François Garillot, Georges Gonthier, Assia Mahboubi, and Laurence Rideau. "Packaging Mathematical Structures". In: *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2009)*. Vol. 5674. Munich, Germany: Springer, 2009, pp. 327–342 (cit. on pp. 99, 101).

[21] Jean-Yves Girard. "Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur". PhD thesis. Université Paris 7, 1972 (cit. on pp. 20, 30, 48).

[22] Georges Gonthier. "Formal Proof — The Four-Color Theorem". In: *Notices of the American Mathematical Society* 55.11 (Dec. 2008), pp. 1382–1393 (cit. on p. 8).

[23] Georges Gonthier, Assia Mahboubi, and Enrico Tassi. *A Small Scale Reflection Extension for the Coq system*. Tech. rep. 6455. Microsoft Research – Inria Joint Centre, 2009 (cit. on pp. 6–8, 24, 43, 62, 70, 81).

[24] 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: *Proceedings of the 4th International Conference on Interactive Theorem Proving (ITP 2013)*. Vol. 7998. Lecture Notes in Computer Science. Rennes, France: Springer, 2013, pp. 163–179 (cit. on p. 8).

[25] Georges Gonthier, Beta Ziliani, Aleksandar Nanevski, and Derek Dreyer. "How to make ad hoc proof automation less ad hoc". In: *Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming (ICFP '11)*. Tokyo, Japan: ACM, 2011, pp. 163–175 (cit. on pp. 7, 60, 107).

[26] Paul Graham. *ANSI Common Lisp*. Prentice Hall Press, 1996 (cit. on p. 137).

[27] Fritz Henglein. "Type Inference with Polymorphic Recursion". In: *ACM Transactions on Programming Languages and Systems* 15.2 (1993), pp. 253–289 (cit. on p. 116).

[28] Manuel V. Hermenegildo, Francisco Bueno, Manuel Carro, Pedro López-García, Edison Mera, José F. Morales, and Germán Puebla. "An overview of Ciao and its design philosophy". In: *Theory and Practice of Logic Programming* 12.1-2 (2012), pp. 219–252 (cit. on p. 111).

[29] C. A. R. Hoare. "An Axiomatic Basis for Computer Programming". In: *Communications of the ACM* 12.10 (1969), pp. 576–580 (cit. on p. 112).

[30] William A. Howard. "The formulae-as-types notion of construction". In: *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*. Ed. by Jonathan P. Seldin and J. Roger Hindley. Original paper manuscript from 1969. Academic Press, 1980, pp. 479–490 (cit. on p. 29).

[31] Paul Hudak, Simon L. Peyton Jones, Philip Wadler, Brian Boutel, Jon Fairbairn, Joseph H. Fasel, María M. Guzmán, Kevin Hammond, John Hughes, Thomas Johnsson, Richard B. Kieburtz, Rishiyur S. Nikhil, Will Partain, and John Peterson. "Report on the Programming Language Haskell, A Non-strict, Purely Functional Language". In: *SIGPLAN Notices* 27.5 (1992), pp. 1– (cit. on p. 8).

[32] Chung-Kil Hur, Georg Neis, Derek Dreyer, and Viktor Vafeiadis. "The power of parameterization in coinductive proof". In: *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '13)*. Rome, Italy: ACM, 2013, pp. 193–206 (cit. on p. 7).

[33] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. "Featherweight Java: a minimal core calculus for Java and GJ". In: *ACM Trans. Program. Lang. Syst.* 23.3 (2001), pp. 396–450 (cit. on p. 137).

[34] Xavier Leroy and Hervé Grall. "Coinductive big-step operational semantics". In: *Information and Computation* 207.2 (2009), pp. 284–304 (cit. on p. 7).

[35] Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. *The OCaml system, release 4.01. Documentation and user's manual*. Tech. rep. Available at http://caml.inria.fr/pub/docs/manual-ocaml/. INRIA, 2013 (cit. on p. 8).

[36] John W. Lloyd. *Foundations of Logic Programming, 2nd Edition*. Springer, 1987 (cit. on p. 111).

[37] Assia Mahboubi and Enrico Tassi. "Canonical Structures for the Working Coq User". In: *Proceedings of the 4th International Conference on Interactive Theorem Proving (ITP 2013)*. Vol. 7998. Lecture Notes in Computer Science. Rennes, France: Springer, 2013, pp. 19–34 (cit. on pp. 60, 107).

[38] Per Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis, 1984 (cit. on pp. 29, 48).

[39] Robin Milner, Mads Tofte, and David Macqueen. *The Definition of Standard ML*. Cambridge, MA, USA: MIT Press, 1997 (cit. on p. 8).

[40] Eugenio Moggi. "Notions of Computation and Monads". In: *Information and Computation* 93.1 (1991), pp. 55–92 (cit. on p. 124).

[41] Aleksandar Nanevski, Greg Morrisett, and Lars Birkedal. "Polymorphism and separation in Hoare Type Theory". In: *Proceedings of the 11th ACM SIGPLAN International Conference on Functional Programming (ICFP '06)*. Portland, Oregon, USA: ACM, 2006, pp. 62–73 (cit. on pp. 7, 127).

[42] Aleksandar Nanevski, J. Gregory Morrisett, and Lars Birkedal. "Hoare type theory, polymorphism and separation". In: *Journal of Functional Programming* 18.5-6 (2008), pp. 865–911 (cit. on pp. 7, 127, 129, 138).

[43] Aleksandar Nanevski, Viktor Vafeiadis, and Josh Berdine. "Structuring the verification of heap-manipulating programs". In: *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '10)*. Madrid, Spain: ACM, 2010, pp. 261–274 (cit. on pp. 98, 127, 129).

[44] Aleksandar Nanevski, Ruy Ley-Wild, Ilya Sergey, and Germán Andrés Delbianco. "Communicating State Transition Systems for Fine-Grained Concurrent Resources". In: *Proceedings of the 23rd European Symposium on Programming (ESOP 2014)*. Vol. 8410. Lecture Notes in Computer Science. Grenoble, France: Springer, 2014, pp. 290–310 (cit. on pp. 7, 98).

[45] Ulf Norell. "Towards a practical programming language based on dependent type theory". PhD thesis. SE-412 96 Göteborg, Sweden: Department of Computer Science and Engineering, Chalmers University of Technology, 2007 (cit. on p. 48).

[46] Martin Odersky. *The Scala Language Specification, version 2.9*. Tech. rep. Lausanne, Switzerland: Programming Methods Laboratory, EPFL, 2014 (cit. on pp. 8, 121, 124).

[47] Martin Odersky and Matthias Zenger. "Scalable Component Abstractions". In: *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '05)*. San Diego, California, USA: ACM, 2005, pp. 41–57 (cit. on p. 102).

[48] Simon L. Peyton Jones. "Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell". In: *Proceedings of the 2000 Marktoberdorf Summer School on Engineering theories of software construction*. Marktoberdorf, Germany: IOS Press, 2001, pp. 47–96 (cit. on p. 125).

[49] Simon L. Peyton Jones and Philip Wadler. "Imperative Functional Programming". In: *Proceedings of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '93)*. Charleston, South Carolina, USA: ACM Press, 1993, pp. 71–84 (cit. on pp. 112, 125).

[50] Simon L. Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. "Simple unification-based type inference for GADTs". In: *Proceedings of the 11th ACM SIGPLAN International Conference on Functional Programming (ICFP '06)*. Portland, Oregon, USA: ACM, 2006, pp. 50–61 (cit. on p. 54).

[51] Frank Pfenning and Conal Elliott. "Higher-Order Abstract Syntax". In: *Proceedings of the ACM SIGPLAN'88 Conference on Programming Language Design and Implementation (PLDI '88)*. Atlanta, Georgia, USA: ACM, 1988, pp. 199–208 (cit. on p. 136).

[52] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 2002 (cit. on pp. 20, 30, 44, 50, 115, 117, 120).

[53] Benjamin C. Pierce, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, and Brent Yorgey. *Software Foundations*. Available at http://www.cis.upenn.edu/~bcpierce/sf. Electronic textbook, 2014 (cit. on pp. 6–7, 89, 94, 114, 137).

[54] John C. Reynolds. "Definitional Interpreters for Higher-Order Programming Languages". In: *Proceedings of 25th ACM National Conference*. Reprinted in Higher-Order and Symbolic Computation 11(4):363–397, 1998. Boston, Massachusetts, USA: ACM, 1972, pp. 717–740 (cit. on p. 120).

[55] John C. Reynolds. "Separation Logic: A Logic for Shared Mutable Data Structures". In: *Proceedings of the 17th IEEE Symposium on Logic in Computer Science (LICS 2002)*. Copenhagen, Denmark: IEEE Computer Society, 2002, pp. 55–74 (cit. on p. 118).

[56] John C. Reynolds. "Towards a theory of type structure". In: *Symposium on Programming*. Vol. 19. LNCS. Paris, France: Springer, 1974, pp. 408–423 (cit. on pp. 20, 30).

[57] Amokrane Saïbi. "Outils Génériques de Modélisation et de Démonstration pour la Formalisation des Mathématiques en Théorie des Types: application à la Théorie des Catégories". PhD thesis. Paris, France: Université Paris VI, 1999 (cit. on p. 106).

[58] Matthieu Sozeau. "Subset Coercions in Coq". In: *Proceedings of the International Workshop on Types for Proofs and Programs*. Vol. 4502. Lecture Notes in Computer Science. Nottingham, United Kingdom: Springer, 2007, pp. 237–252 (cit. on p. 128).

[59] Matthieu Sozeau and Nicolas Oury. "First-Class Type Classes". In: *Proceedings of the 21st International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2008)*. Vol. 5170. Lecture Notes in Computer Science. Montreal, Canada: Springer, 2008, pp. 278–293 (cit. on p. 107).

[60] Matthieu Sozeau and Nicolas Tabareau. "Universe Polymorphism in Coq". In: *Proceedings of the 5th International Conference on Interactive Theorem Proving (ITP 2014)*. Vol. 8558. Lecture Notes in Computer Science. Vienna, Austria: Springer, 2014, pp. 499–514 (cit. on p. 50).

[61] Antonis Stampoulis and Zhong Shao. "Static and user-extensible proof checking". In: *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '12)*. Philadelphia, Pennsylvania, USA: ACM, 2012, pp. 273–284 (cit. on p. 7).

[62] Antonis Stampoulis and Zhong Shao. "VeriML: typed computation of logical terms inside a language with effects". In: *Proceeding of the 15th ACM SIGPLAN international conference on Functional programming (ICFP '10)*. Baltimore, Maryland, USA: ACM, 2010, pp. 333–344 (cit. on pp. 7, 83).

[63] Philip Wadler. "The Essence of Functional Programming". In: *Proceedings of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '92)*. Albuquerque, New Mexico, USA: ACM Press, 1992, pp. 1–14 (cit. on p. 124).

[64] Philip Wadler and Stephen Blott. "How to Make ad-hoc Polymorphism Less ad-hoc". In: *Proceedings of the Sixteenth Annual ACM Symposium on Principles of Programming Languages (POPL '89)*. Austin, Texas, USA: ACM Press, 1989, pp. 60–76 (cit. on pp. 97, 107).

[65] Stephanie Weirich, Brent A. Yorgey, and Tim Sheard. "Binders unbound". In: *Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming (ICFP '11)*. Tokyo, Japan: ACM, 2011, pp. 333–345 (cit. on p. 137).

[66] Benjamin Werner. "Sets in Types, Types in Sets". In: *Proceedings of the Third International Symposium on Theoretical Aspects of Computer Software (TACS '97)*. Vol. 1281. Lecture Notes in Computer Science. Sendai, Japan: Springer, 1997, pp. 530–346 (cit. on p. 47).

[67] Glynn Winskel. *The formal semantics of programming languages — an introduction*. Foundation of computing series. MIT Press, 1993 (cit. on pp. 116, 138).

[68] Andrew K. Wright and Matthias Felleisen. "A Syntactic Approach to Type Soundness". In: *Information and Computation* 115.1 (1994), pp. 38–94 (cit. on p. 117).

[69] Hongwei Xi, Chiyan Chen, and Gang Chen. "Guarded recursive datatype constructors". In: *Proceedings of the 30th SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '03)*. New Orleans, Louisisana, USA: ACM, 2003, pp. 224–235 (cit. on p. 54).

[70] Beta Ziliani, Derek Dreyer, Neelakantan R. Krishnaswami, Aleksandar Nanevski, and Viktor Vafeiadis. "Mtac: a monad for typed tactic programming in Coq". In: *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming (ICFP '13)*. Boston, Massachusetts, USA: ACM, 2013, pp. 87–100 (cit. on pp. 7, 83).

Index