Introducing Functional Programmers to Interactive Theorem Proving and Program Verification

Teaching Experience Report *

Ilya Sergey
IMDEA Software Institute
ilya.sergey@imdea.org

Aleksandar Nanevski
IMDEA Software Institute
aleks.nanevski@imdea.org

Abstract

We report on the design and preliminary evaluation of a short introductory course on interactive theorem proving and program verification using the Coq proof assistant, targeted at students with background in functional programming and software engineering.

The course builds on concepts familiar from functional programming to develop understanding of logic and mechanized proving by means of the Curry-Howard isomorphism. A particular emphasis is made of the *computational* nature of *decidable* properties of various data structures. This approach is of practical importance, as Coq's normalization can automatically simplify or discharge such properties, thus reducing the burden of constructing the proofs by hand. As a basis for teaching this style of mechanization, we use Gonthier *et al.*'s Ssreflect extension of Coq and its associated libraries.

In the course, we minimize the exposure to ad-hoc proof automation via tactics, and request that students develop proofs using only a small set of proof-building primitives that they should clearly understand. In addition to introducing logic as an application of functional programming, the topics covered by the course include: implementation of custom rewriting principles as instances of indexed type families, boolean reflection, implementation of algebraic structures and inheritance between them, and verification of imperative programs in separation logic and Hoare Type Theory.

1. Introduction

The concept of a rigorous mathematical proof has a long story as an educational topic. In our experience, however, a typical undergraduate curriculum in computer science teaches proofs from the point of view of logic, and ignores the fundamental connection between logic and programming. Thus, most undergraduates in computer science receive a strong background in programming, imperative as well as functional, but will seldom relate their programming intuition to logic and formal proofs.

This situation is unfortunate, as recent years have seen the completion of a number of impressive and ground-breaking project in the field of formal mathematics and program verification. Examples include the four color theorem [8], the odd order theorem [10], seL4 kernel [15] and CompCert compiler infrastructure [16]. These advances attest that proof assistants have matured to the level at which they are ready for use by the wider community in mathematics and computer science.

Mechanized formalization of a large mathematical theory significantly resembles development of a large programming project with a rich infrastructure and a set of libraries. In this light, it is unsurprising that an effective use of a proof assistant requires remarkable programming skills. This insight is especially true of proof assistants such as Coq [4] or Agda [22], which include a powerful dependently typed lambda calculus, in which one can write both programs as well as proofs, via the Curry-Howard isomorphism. However, the connection between formal mathematics and programming goes much deeper than the mere fact that proofs and programs can both be written in the lambda calculus. Indeed, even the way one defines the basic predicates of interest, implements algebraic objects and structures such as groups or rings, or organizes formal theories, bears a lot of similarity with programming in the large. In particular, programming concepts such as modules, functors, objects, abstract types and predicates, higher-order functions, inductive and coinductive types and families, overloading and inheritance, very quickly arise as natural mechanisms for structuring formal mathematical developments.

Most fascinatingly—and, perhaps, confusingly to a newcomer with pure mathematical background—there may be many different ways, employing many different programming mechanisms, to formulate and mechanize the same mathematical problem. Each choice may present a number of tradeoffs that are not typically encountered in mere programming. For instance, should one encode a particular property as a function or as an inductive datatype? It is important to know the tradeoffs well; good choices lead to reusable, maintainable and short proofs, just like they would in good programs. But, conversely, suboptimal or "hacky" choices typically lead to an explosion in complexity, much more quickly and severely than they do in programming. In the light of this observation, one may say that effective construction of formal proofs is not just a programming challenge, but a programming challenge of the highest order, in which one's coding skills, discipline and inventiveness can really shine, much more so than in ordinary programming (*i.e.*, in which proofs are not required).

This article describes an introductory course on interactive theorem proving and verification using Coq, which we designed to embrace the uniformity of the tradeoffs, that pertain to both pro-

* A part of this work has been carried out while the first author was giving lectures in a summer school on interactive theorem proving, which was sponsored by JetBrains and took place in August 2014 at Saint Petersburg State University.

gramming and proving. The considered issues appear frequently in our everyday formalization work, but because they have not been collected and emphasized in the existing introductory literature (rather, they are interspersed across research papers of many authors [6, 7, 9, 11, 17, 23]), we have frequently needed to devote time to introduce them to new students and collaborators (all with significant experience with ML and Haskell), individually. The idea for this course and its selection of topics arose out of such informal teaching experience. In a more formal setting, we report on the proof-of-concept evaluation of the course, which was delivered by the first author in a five-days summer school for a small group of students of Saint Petersburg State University (SPbSU), majoring in mathematics and software engineering and having necessary background in Haskell, thanks to the standard curriculum courses.

By necessity imposed by space limits, our presentation in this report assumes basic familiarity with Coq, though, of course, such familiarity is not assumed in the course itself, the associated lecture notes, slides, and code templates for hands-on classes [29]. However, we do not assume that the reader will be able to follow the ocasional Coq proofs we present. In such cases, we discuss the main ideas of the proof in prose. In the rest of the article, we present the broader outline of the course, focusing in more depth on a few specific topics that are not usually covered by the existing literature and courses on Coq.

2. Overview

Currently, there are several books by different authors on interactive theorem proving in Coq [1, 3, 26] (see Section 5 for the discussion). Although all these books have been successfully used in numerous introductory courses on Coq, we thought that there are still some topics, stemming from the proving-as-programming intuition and essential for effective and boilerplate-free mathematical reasoning via a proof assistant, that are left underrepresented. Our course is targeted to fill these gaps while still covering the common teaching material required to get the students off the ground. In particular, in our course we emphasized 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 [9]:

- Special treatment is given to the *computational* nature of reasoning about *decidable* propositions. In other words, many results about decidable properties can be *computed* as boolean values, rather than *proven* interactively. But to do so, one has to formulate the properties as computable recursive Coq functions with `bool` result, rather than as inductive predicates. The latter is more in the spirit of the traditional introductions to Coq.
- Instead of supplying the students with a large vocabulary of automated tactics necessary for everyday Coq hacking, we focus on a small but expressive set of primitives (about six in total), offered by Ssreflect or inherited from the vanilla Coq with notable enhancements.
- Reasoning by rewriting is presented from the perspective of Coq's definition of the propositional equality and followed by elaboration on the idea of using *index type families* as a tool to define client-specific conditional rewrite rules. The usual way indexed type families are presented in the related work is for use in programming and pattern matching, rather than rewriting. Conversely, we advocate parametrized (but not indexed) types, for programming and pattern matching.
- We provide a detailed explanation of the essentials of Ssreflect's *boolean reflection* between the sort `Prop` and the datatype `bool` as a particular case of conditional rewriting, following the computational approach to proving decidable properties.
- Formal encoding of familiar mathematical structures (*e.g.*, monoids, partial orders, *etc*) is carried out by means of *dependent records*; mathematical operations are overloaded using the mechanism of *canonical instances*, similar to Haskell type classes. This analogy illustrates how the programming ideas of overloading and inheritance come into formal proving [7].
- A novel (from a teaching perspective) case study is considered, introducing the students to the concepts of stateful program verification using separation logic and Hoare Type Theory [19].

2.1 Why teach with Ssreflect?

A significant part of our material is presented using the Ssreflect extension of Coq [9]. Ssreflect is developed as a part of the Mathematical Components project,[1] to facilitate automated reasoning and simplification in very large mathematical developments, in particular, the formalizations of the four color theorem [8] and the Feit-Thompson (odd order) theorem [10].

Ssreflect includes a small but expressive (and comfortable for practical work) set of primitives for proof construction, related to but different from the traditional set provided by Coq. It also comes with a large library of 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 proofs very concise.

Using Ssreflect for our development was not the goal by itself: a large part of the course could be presented using traditional Coq without any loss in the insights but, perhaps, some loss in brevity. However, having been developed as part of a very large formalization effort, Ssreflect's libraries are well-tested in the wild, and are an excellent example of good mechanization practice. In particular, they make it very easy for us to emphasize the above-listed mechanization aspects of the course. In fact, to the best of our knowledge, it was the work on Ssreflect and the Four color theorem [6–8] that first applied boolean reflection and canonical structures in Coq to a larger-scale formalization effort.

Last, but not least, Ssreflect comes with a much improved `Search` tool, compared to standard Coq. The `Search` tool is invaluable, given that a fair part of the time spent on mechanization is typically devoted to reading third-party code and lemma libraries.

3. Structure of the course

In this section we provide a detailed description of the structure of the course and refer the interested reader to the full syllabus and the accompanying exercises available online [29].

3.1 From functional programs to propositional logic

Assuming that the students are knowledgeable about the basic concepts of functional programming, such as algebraic datatypes, pattern matching and possibly higher-order functions, we begin the course by demonstrating the syntax for the same concepts in Coq, focusing solely on Coq's programming language component.

3.1.1 Functional programming in Coq

We start by presenting simple datatypes, such as `unit`, `bool`, `nat` and `empty` (*i.e.*, the type without constructors), proceeding shortly after to the definitions of functions on these datatypes. For example, addition on `nat` is defined by the program below, which introduces the students to Coq's syntax for pattern matching and its abbreviated Ssreflect variant `if-is`. The latter is useful for cases with only one non-default branch. `_.+1` is Ssreflect notation for successor.

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

[1] `http://www.msr-inria.fr/projects/mathematical-components-2/`

Next, we focus on the induction and recursion principles, generated by Coq automatically for each inductive definition. We explain these as higher-order functions, whose return types can depend on the value of the argument; hence, this gives us a way to introduce *dependent function types* as well. We show how the recursive function `my_plus` can be rewritten explicitly in terms of the recursion principles, and suggest a number of exercises intended to expose the students to working with higher-order dependently-typed functions. This lecture culminates with a short demonstration of custom dependently-typed functions, *e.g.*, a function returning `unit` on some inputs and `nat` on some others.

Elaborating futher on the types of generated recursion and induction principles, we draw the students' attention to the recursion principle generated for the datatype `empty`:

```
empty_rect : forall (P : empty -> Type) (e : empty), P e
```

A function with such a type allows one to obtain instance of *any* type, given an argument `e` of type `empty`, thanks to the argument `P`, which can be instantiated by a constant function returning a necessary type. A closer examination of the body of `empty_rect` by means of `Print` machinery leaves no doubts that it is well-typed. Therefore, an important conclusion follows: *assuming existence of a value that cannot be constructed, we are able to construct anything*. We next show that `empty` is not the only datatype that can have this "magical" property of allowing to construct a value of any type from it. In particular, we refer to the example with a type `strange`, taken from [1], which is also *non-inhabited*, even though it has a constructor:

```
Inductive strange := cs of strange.
```

While the students don't have all that much experience with the `empty` type, as it is not frequently encountered in programming, we point out that it will correspond to the empty set in mathematics, and to the false proposition in logic, and will thus have a significant role to play very soon in the course.

We continue by showing a number of familiar algebraic datatypes, such as pairs, sums, lists and trees as well as functions operating on these, suggesting a number of simple programming exercises. After successfully accomplishing a few of them, the students already feel relatively comfortable with Coq as a programming language and realize its main "limitation". In particular, they notice they can't define generally-recursive functions in the usual programming style, and are forced to rely on primitive recursion in order to convince Coq's type checker that their functions terminate.

3.1.2 Searching and structuring libraries

In practical programming, it is important to be able to search for appropriate procedures in someone else's libraries. The same holds for interactive proof development, where one needs to search for definitions and lemmas. When programming in Haskell, one may use search engines such as *Hoogle*[2] and *Hayoo!*.[3] This functionality is provided in Coq by the `Search` command, which is further enhanced by Ssreflect (see Chapter 10 of [9]). We decided to introduce the students to `Search` very early on. Even though by this moment in the course, they are not familiar with formal logic and proofs, they can already use `Search` to look for definitions and functions from the programming side, based on their types. For example, they can execute queries like the following one

```
Search _ ((?X -> ?Y) -> _ ?X -> _ ?Y).
```

which returns all `map`-like functions currently available in the imported libraries.

[2] `http://www.haskell.org/hoogle/`

[3] `http://hayoo.fh-wedel.de/`

Coq's extensible parser is a powerful tool when defining custom notations for mathematical theories. From our experience it is also a source of constant frustration for the Coq newcomers, who struggle to figure out whether some notation is defined in a library, or is Coq's native syntax. To avoid the pain of working with custom third-party notations, we teach them to use commands like `Locate` and `Print`. We also show how to "switch off" all the syntactic sugar from custom notations with the `Set Printing All` command. We also show how to define one's own custom notation, with a warning that the feature should be used carefully, as it is easy to abuse and make one's code unreadable.

At this point we also introduce the students to basic program structuring primitives of Coq, such as sections and modules. We explain the difference between the two (*e.g.*, modules provide hierarchical name spaces, while sections declare local variables that generalize over the section body), and illustrate how to combine the two to achieve a degree of "locality" for definitions and hiding of the internal details.

3.1.3 Introducing logical connectives

By this moment, the students are familiar with the basic principles of programming in Coq. In particular, they understand that values of inductive datatypes are constructed by applying constructors to the arguments and "destructed" by pattern matching. They know how to define functions of appropriate types, and that a function should be applied to get the value of its result type. They have even seen dependently-typed functions whose return type can vary depending on the value of the arguments. In other words, everything is set up for the introduction of constructive logic by means of the Curry-Howard analogy.

First of all, we define when a logical statement will be *true* for us: when its proof *can be built* from hypotheses and rules by referring to the hypotheses and applying the rules, correspondingly. This definition has two important implications, which we elaborate upon: (1) this notion of truth is *constructive*; something is only true if its proof can be constructed in a finite number of steps, and (2) the truth is *relative* with respect to the initial hypotheses and the rules of the logic. In our experience, this definition most often matches very well the intuition that the students have from their classical mathematical education. Even if they haven't been exposed to extensive study of mathematical logic and, hence, have only a very informal idea of what a formal proof is, they know how to write informal ones, and how to find flaws in them. The definition also makes it easy to introduce *falsehood* as a proposition whose proof cannot be constructed.

Next, we appeal to the analogy between the falsehood `False` and the datatype `empty`, defined previously, followed by the introduction of standard logical connectives by the Curry-Howard correspondence: conjunction corresponds to the product datatype `prod`, disjunction to `sum`, truth to a type with a single element (in Coq, the element is named `I`), implication to function type `A -> B`, universal quantification to dependent function types, and existential quantification to a specific dependent record type. This intuition gradually leads to the idea that proving a logical proposition corresponds to building a program that has the proposition as its type. In other words, we're trying to *inhabit* a type with a *proof term* element.[4]

Given this relation, it is natural to explain the usual inference rules in terms of functional programming. In particular, a function can be introduced by *assuming* its argument and constructing its body, in which the argument is used, which corresponds to the rules of implication introduction. The similar intuition holds

[4] While drawing the analogy of types/propositions as sets of proofs is useful for building intuition, we find it important to emphasize that, unlike sets in set theory, types in Coq are always *disjoint*; that is, an element cannot belong to more than one type.

for universal quantification. The introduction rules for other logical connectives correspond to application of a *constructor* of the corresponding logical connective (*e.g.*, `conj` in the case of conjunction, and similarly for existential quantification). As for elimination rules, most of the connectives (conjunction, disjunction, existentials) clearly correspond to case-analysis by means of Coq's `match-with` expression construct. The notable exceptions are the *modus ponens* rule for implication, and the `forall`-specialization rule, which both correspond to function application.

3.1.4 Interactive proof construction: first encounter

After concepts related to proofs have been introduced, we can proceed to interactive proof development. We point out that even though proofs are the same as programs, it's frequently easier to develop them not as explicit lambda-terms, but in a step-by-step manner using proof scripting with a small (but expressive) set of Ssreflect's primitives.

First, we show the simplest possible proof script, which just appeals to a known fact whose proof is constructed explicitly as a lambda-term. This construction is done by the `exact` primitive, as in the following proof of `True`, whose explicit proof is called `I`:

```
Theorem true_is_true : True.
Proof. exact: I. Qed.
```

Next, we demonstrate the machinery enabled by Ssreflect's `move` tactic placeholder and two *bookkeeping* tacticals, `=>` and `:`. These allow one to work with *assumptions*, moving them "bottom-up" (*i.e.*, from the goal to the context) and vice versa, respectively:

```
Theorem imp_trans (P Q R: Prop) :
        (P -> Q) -> (Q -> R) -> P -> R.
Proof. move => H1 H2 p; exact: (H2 (H1 p)). Qed.
```

We emphasize that the proof by `exact` is and instance of so-called *forward* reasoning style, in which assumptions are combined to eventually obtain the proof of the goal. In the above proof, `H1`, `H2`, `p` are proofs of `P->Q`, `Q->R`, and `p`, respectively, and the proof of `R` is obtained as an application `H2 (H1 p)`. We also exhibit the opposite—*backwards*—reasoning style, which replaces the goal by a number of obligations arising from the types of the applied function/hypothesis' arguments. In Ssreflect, backwards reasoning is implemented by the `apply:` tactic,[5] the use of which we immediately demonstrate on a number of simple examples:

```
Theorem all_imp_dist A (P Q: A -> Prop) :
  (forall x: A, P x -> Q x) -> (forall y, P y) ->
  forall z, Q z.
Proof. move => H1 H2 z; apply: H1; apply: H2. Qed.
```

We elaborate that constructing proofs by means of `apply:` essentially corresponds to reading logical introduction rules "bottom-up". In this way, constructors of logical connectives are essentially equated with hypotheses in context.

At this point, students are equipped with three main proof building primitives, and can already construct simple proofs. The main missing component is the primitive for case analysis (*i.e.*, pattern-matching), which we introduce next. As most of the Ssreflect tactics, `case` can be effectively combined with bookkeeping tacticals, which we make use of immediately.[6]

```
Theorem conj_comm P Q : P /\ Q -> Q /\ P.
Proof. case=> p q; apply: conj; [exact: q|exact: p]. Qed.
```

If no specific argument is provided, the tactics `move` and `case` by default always work with the *leftmost* assumption of the goal.

[5] Which is somewhat more powerful than Coq's native `apply` (without ":").

[6] The necessary syntax for working with multiple subgoals, *e.g.*, the `[||]` tactical, is introduced on the way by gradually "compressing" the proofs in front of the students during the hands-on recitations.

The four primitives introduced by now: `exact`, `move`, `case` and `apply`, together with Ssreflect's bookkeeping machinery, are sufficient for building proofs of arbitrary logical propositions. We next explain a number of "shortcut" tactics that simplify the work with particular logical connectives and predicates. Examples include: `split`, `exists`, `left`, `right`, *etc*. While explaining the shortcut tactics, we ask the students to express them through the four basic ones. We also explain Ssreflect's *terminators* `by` and `done`, which raise an error message if they fail to discharge the goal. Finally, we illustrate the tactic `intuition` on a few examples. This tactic attempts to prove propositions in intuitionistic logic automatically. In this case too, we ask the students to prove the same examples without automation, using only the four basic primitives.

We conclude this part of the course by providing a brief overview of non-constructive axioms from classical propositional logic, such as the *law of excluded middle*, *Peirce's law*, the *law of double negation*. Students are asked to to prove that these are all equivalent, following an excercise by Bertot and Castéran [1]. We also briefly elaborate on the impredicativity of the sort `Prop` of propositions in Coq, and provide a short explanation of the basics of Coq's hierarchy of universes.

3.2 Equality and rewriting

After explaining the basics of propositional logic, we proceed to address equational reasoning and the corresponding proof technique of *rewriting*. Propositional equality is defined in Coq by the following *indexed type family* (in this case, the index is the element of type `A` in the type signature `A -> Prop`):

```
Inductive eq (A : Type) (x : A) : A -> Prop :=
  eq_refl : eq x x.
Notation "x = y" := (eq x y) (at level 70).
```

A common way of understanding indexed type families such as the one above is to compare them to *generalized algebraic datatypes* (GADTs) in Haskell [24, 33]. The latter allow the programmer to refine and guide the process of (dependent) pattern matching by the type index of the scrutinee. Studying indexed type families from this point of view is a very active research area [5, 18]. However, not wanting to get bogged down with too many details of dependent pattern matching, we decided to explain indexed type families by another well-known analogy [23], specifically useful in Coq: indexed type families correspond to *custom* (*i.e.*, user-defined) *simultaneous conditional rewrite rules*.

The intuition about this is best built by examples. For instance, the proof of the equality's symmetry will look as follows:[7]

```
Lemma eq_sym A (x y: A) : x = y -> y = x.
Proof. by case. Qed.
```

The `case`-analysis on the assumption `x = y` essentially forces the unification mechanism of Coq to exploit the "implicit equality", encoded via the `eq`'s index (*i.e.*, the second argument), and *substitute* all occurrences of the index's actual value by the value declared in the constructor `eq_refl` first argument. In other words, case-analysing the assumption rewrites the occurrence of `y` by `x`, therefore leading to a subgoal `x = x` which trivially discharged by the terminator `by`. In a more pictorial analogy, one can think of the predicate `eq x y` as of the following "rewriting" table:

		y
eq_refl	x	x

In English, for the fixed value `x` (which is bound by the datatype's parameter), the instance of `eq x y` defines an "rewriting

[7] In fact, the case-analysis on Coq's predicate family `eq` is overloaded by Ssreflect, so this example is presented in the course using an equivalent, user-defined, equality predicate family `my_eq`.

table", whose top-right cell specifies *what* should be replaced (in this case, this is `y`, which is captured by the datatype's index). The bottom-right cell, thus, specifies what the actual value of the index, `y`, should be substituted *with*. Hence, case-analysis on such a rewriting table will replace all the occurrences of the columns' "headers" (values of indices in the actual instance) by the corresponding values in the "cells" (values of indices in the constructors). In this analogy, the "rows" of the table correspond to particular constructors and their arguments, and in the case of `eq` there is just one, `eq_refl`, with a single argument `x`.

This intuition is further strenghtened by more involved examples from the Ssreflect's library of natural numbers (`ssrnat`). In particular, we consider the following indexed family `nat_rels` and the companion lemma `ltngtP`:

```
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.

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

The parameters of `nat_rels` are the nats `m` and `n`. The definition also has three indexes, all boolean. Together, `nat_rels` and `ltngtP` encode the following rewriting table.

		`m < n`	`n < m`	`m == n`
`CompareNatLt`	`m < n`	`true`	`false`	`false`
`CompareNatGt`	`m > n`	`false`	`true`	`false`
`CompareNatEq`	`m = n`	`false`	`false`	`true`

The column headers specify the subexpression in the goal to be substituted by case analysing on a proof of `ltngtP` (*i.e.*, *what* is substituted). In this particular case, we want to simultanously substitute for three different boolean expressions: `m < n`, `n < m` and `m == n`.[8] The row headers specify when (*i.e.*, *under which conditions*) the substitution takes place. The value identified by a given row and column says what we substitute *with*. More concretely, case analysing on a proof of `ltngtP` will make Coq substitute the occurrences of `m == n` in a goal as follows (and similarly for `m < n` and `n < m`). It will generate three subgoals, in which `m == n` is replaced with `false`, `false`, and `true`, respectively. The subgoals will contain hypotheses as read off from the row headers; the first subgoal will be conditioned on `m < n`, the second on `m > n`, and the third on `m = n`. One can employ such simultaneous conditional rewriting to carry out very short and effective proofs. For example, the following proofs about `minn` and `maxn` functions are taken from the `ssrnat` library:

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

Lemma addn_min_max m n : minn m n + maxn m n = m + n.
Proof.
by rewrite /minn /maxn; case: ltngtP=>//; rewrite addnC.
Qed.
```

In the proof of `addn_min_max`, right after *unfolding* the definitions of `minn` and `maxn` by the rewrite command, the case-analysis on the table constructed by `ltngtP` with *simultaneous* rewritings and reducing `if`-expressions, produces a number of subgoals, most of which are discharged by Ssreflect's trailing tactical `//`, which can be expressed in somewhat standard Coq as `try done` for each of the subgoals. The observation about simultaneity of rewriting with respect to `m < n`, `n < m` and `m == n` is of essence. Had we tried instead to build the proof of `addn_min_max` based on the case-analysis over a lemma stating the totality of `<`,[9] we would first have to prove a series of auxiliary properties, such as that `n < m` is `false` when `m < n` is `true`, and vice versa. The latter increases the proof burden. The simultaneous rewriting via the `ltngtP` lemma "packages" these facts together in the truth table `nat_rels`.

While case analysis on lemmas over indexed families logically corresponds to rewriting, the above example readily illustrates that a separate `rewrite` primitive is also very useful, in particular with a number of enhancements provided by Ssreflect. The usage scenarious of a separate `rewrite` primitive include folding/unfolding definitions (by means of the `/` modality in Ssreflect), rewriting in assumptions and sub-expressions, and using occurrence switches to select specific subexpressions for rewriting (see Chapter 7 of [9]).

In the excercises for this module, we ask the students to practice with rewrite tactics on a number of examples involving natural numbers. The selection of exercises also familiarizes them with the `ssrnat` library, where most of the lemmas take the form of equality. One such lemma is `addnC` employed in the proof above, which states commutativity of addition. The students also get to practice with the `Search` tool to find lemmas that encode the familiar properties of natural numbers such as associativity of addition, neutrality of zero, *etc*.

3.3 Boolean reflection and views

One confusing point for students with programming background, when they are exposed to Coq, is the difference between the two-constructor enumeration type `bool` and Coq's sort of propositions `Prop`. At this point in the course, we explain the difference, but in a way that illustrates the trade-offs between the two concepts, when it comes to building interactive proofs. Following Ssreflect, we suggest that one should use boolean expressions to encode *decidable* properties, whereas the propositions from `Prop` should be used for undecidable properties, or properties whose decidability is not trivial to establish. Consequently, one obvious difference that can be immediately pointed out is that `Prop` allows quantification over infinite domains, while `bool` doesn't. But more importantly, `bool` and `Prop` can be related by an indexed family (i.e, a "rewriting table", as introduced in the previous section). This idea is at the core of Ssreflect and, if used properly, often leads to very effective and compact proofs.

3.3.1 Coercing `bool` into `Prop`

We start by pointing out that booleans can be naturally injected into propositions by equating their value to `true`, using propositional equality defined in the previous section. This injection is done in Ssreflect by the implicit *coercion*, defined in the `ssrbool` library:

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

We explain this coercion as an implicit type conversion, familiar to students from languages like Scala or Haskell, which Coq inserts automatically every time it "expects to see" a proposition, but instead encounters a boolean. We next show that, given a suitably formulated boolean expressions, reasoning with booleans can be surprisingly nifty, as demonstrated by the following example, involving Ssreflect's bool-returning function `prime`, which implements Erathostenes' sieve:

```
Goal prime (16 + 13). Proof. done. Qed.
```

Indeed, the proof would have been much longer had one used the usual definition of prime numbers as those with only trivial divisors.[10] This example, and a number of similar ones, convey the idea that decidable properties, if implemented as computable functions

[8] The ordering relations and the equality on natural numbers are implemented in Ssreflect as boolean functions, since they are decidable.

[9] This is the lemma `forall m n, m < n \/ m == n \/ m > n`.

[10] In fact, Ssreflect's library `prime` provides both definitions and proves them equivalent, so that the user can employ either one as needed.

returning a boolean result, allow Coq to perform automatic simplifications, and even obtain results by computation, without imposing any proving burden on the user.

3.3.2 Introducing the `reflect` datatype

Coercing booleans into propositions is easy enough by an implicit coercion, but going in the opposite direction is a somewhat more complicated. Ssreflect suggests a general methodology for implementing the equivalence between particular propositions and boolean expressions by means of the following `reflect` indexed family:

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

Indeed, following the "rewriting table" analogy (defined in § 3.2), one can think of `reflect` as a generic rewrite rule, parametrized with respect to "logically equivalent" proposition `P` and a boolean b_P.

		b_P
ReflectT	P	true
ReflectF	~ P	false

We illustrate the use of the `reflect` datatype by examples involving reflection of logic connectives, *e.g.*, conjunction

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

The students are asked to prove a number of similar lemmas, *e.g.*, for disjunction, in order to master the principles of providing custom instances of `reflect`.

We next explain the use of *views* and *view hints*, which are part of Ssreflect's bookkeeping machinery (see Chapter 9 of [9]). We omit the discussion here, but briefly note for the reader unfamiliar with Ssreflect, that views are simply `reflect` lemmas which can be employed to on-the-fly convert a boolean expression into the equivalent proposition. For example, the above lemma `andP` can be used as a view to transform `b1 && b2` into a conjunction `b1 /\ b2`. The latter form may be preferable at times; for example, the conjunction can be destructed into component conjuncts by the usual case analysis, whereas the similar destruction is much more cumbersome in the case of `b1 && b2`. Of course, the advantage of the `b1 && b2` form is that it may often simplify by computation, for special values of `b1` and `b2`, whereas those simplification won't be carried out automatically by Coq in the case of the `b1 /\ b2` form. A particularly useful view is the lemma `eqP` which relates boolean and propositional equality, in the special case of `eqTypes`, *i.e.*, types with decidable equality. We refer to such types as *equality types*, but postpone the definition of `eqType` until one of the future lectures.

```
Lemma eqP (A : eqType) (x y : A): reflect (x = y)(x == y).
```

We use the `eqP` lemma to illustrate the interoperatbility between the two different equalities. For example, the boolean equality `x == y` can be used in the scrutinee of conditionals, as the following example shows, whereas the propositional one cannot. However, in proofs, we may need to use `eqP` to switch between the two forms.

```
Definition foo (x y : nat) := if x == y then 1 else 0.
Goal forall x y, x = y -> foo x y = 1.
Proof. by rewrite /foo => x y /eqP ->. Qed.
```

To prove the goal, one first has to unfold `foo`. Then the view `/eqP` is used to covert the proposition `x = y` into the boolean `x == y`, or more precisely, into `(x == y) = true`. The later is subsequently used (by `->`), to rewrite the condition in `if-then-else` to `true`. By computation, this reduces the goal into `1 = 1`, which is then trivially discharged.

3.4 Proofs about inductive predicates and recursive functions

The next course module is dedicated to proofs by induction on custom datatypes and inductively-defined predicates.

Proof by induction in Ssreflect are usually done using the `elim` tactic. It has a number of enhancements over the standard Coq `induction`, and in particular, it keeps with the semantic of Ssreflect, already introduced with `move` and `case`, that the proof-scripting primitives by default always apply to the leftmost assumption in the goal, initiating the proof by induction on it.

We begin by pointing out that `elim` can be seen as a special case of `apply`, where the lemma chosen for application is the default induction principle generated automatically by Coq for the datatype in question. For example, inducting on natural numbers corresponds to applying `nat_ind` lemma, which expresses the well-known Peano induction schema. Similarly, eliminating falsehood (or, equivalently, the empty set) applies the lemma `empty_ind`.

We illustrate the usual patterns of inductive reasoning by working out a number of examples related to decidable properties of natural numbers. Here, we focus on evenness. We tie to the previous lecture by expressing the property in two ways: as a boolean expressions `evenb`, and as an inductive proposition `evenP` in `Prop`.[11]

```
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.
```

We consider the proofs of following two logically equivalent statements formulated using the two definitions given above.

```
Lemma evenP_contra n : evenP (n + 1 + n) -> False.
Proof.
elim: n=>[|n IH]; first by rewrite addn0 add0n; case.
rewrite addn1 addnS addnC !addnS.
rewrite addnC addn1 addnS in IH.
by case=>// m /eqP; rewrite !eqSS => /eqP <-.
Qed.

Lemma evenb_contra n: evenb (n + 1 + n) -> False.
Proof. by elim: n=>[//|n IH]; rewrite addSn addnS. Qed.
```

We do not expect the reader to understand the two proofs, but it can be noticed that the first proof is significantly more verbose and requires a number of rewritings. Pleasantly, in the second proof, the burden of rewritings is much smaller, thanks to the boolean and computational nature of `evenb`, which automatically performs simplification and partial evaluation in both the base and the inductive case. Intuitively, in the second proof, the boolean computation automatically discharges the base case (`//`), and produces a residual subgoal for the inductive case which is almost identical to the induction hypothesis. We only need a few simple commutations of "1", performed by the lemmas `addSn` and `addnS` from the `ssrnat` library, to finish the proof.

Sometimes, it may happen that a boolean property is less convenient than propositional ones. In the case of `evenb`, the problem arises because the property has an "orbit", which differs from the constructors of the underlying datatype. In the case of evenness, the orbit is 2 (*i.e.*, if a number n is even, then so is $n + 2$), but natural numbers are constructed by incrementing by 1. This makes the following lemma a bit more cumbersome to prove in the boolean case than in the propositional one.

```
Lemma even_add n m : evenb n -> evenb m -> evenb (n + m).
```

[11] In the definition of `evenP`, we make the equalities such as `n = m.+2` *explicit* in the constructor `EvenSS`. This can be avoided by switching to indexed families, but we reserve indexed families for implementing "rewriting tables" only, as shown in § 3.2. With explicit equalities, we can avoid the `inversion` tactic, whose behavior is not trival to explain.

In the propositional case, it turns out one can use induction either on the first or the second *assumption*. Contrarily, in the boolean version, it is only natural to induct on `n` and `m`. Either way, the inductive case is decremented by 1, but to exploit the orbit, we need to decrement by 2. In the course, we illustrate several ways in which this proof can be carried out. One is by generalizing the induction hypothesis in a reasonably simple way to quantify over all numbers smaller than n (or m), not just over the immediate predecessor. But we also illustrate that one can build and then use *custom induction principles* with `elim`. In this particular case, the following custom induction principle suffices.

```
Lemma nat2_ind (P: nat -> Prop) : P 0 -> P 1 ->
  (forall n, P n -> P n.+2) -> forall n, P n.
```

With this lemma, the proof of `even_add` reduces to a half-liner: `by elim/nat2_ind : n.`

The module concludes by summarizing the observed patterns of inductive definitions using predicates and recursive boolean functions, and enumerating the common practices of proving in both styles. A number of follow-up exercises in the hands-on recitations motivates the students to master both techniques, and familiarizes them with the basic Ssreflect libraries about the datatypes `nat`, `bool` and functional lists.

3.5 Programming with abstract algebraic structures

At this point in the course, the students are sufficiently proficient in proving statements in equational logic over natural numbers. We next introduce them to building their own theories of algebraic structures, familiar from the university classes on abstract algebra.

One can initiate the discussion of abstract algebraic structure with an intuition from programming. Indeed, an abstract algebraic structure is similar to the notion of class from object-oriented programming, module from Standard ML, or type class from Haskell [32]. In simply-typed languages, these mechanisms allow the programmer to aggregate, *i.e.*, *package* together, a number of operations over some datatype, potentially abstracting over the datatype implementation. In a dependently-typed language such as Coq, we can do a bit more: we can package the operations together with *proofs* of their important properties, such as commutativity or associativity, to obtain an abstract structure.

3.5.1 Defining abstract structures

As a running example, we consider partial commutative monoids (PCMs); an algebraic structure which recurrs in our current ongoing work on the verification of stateful and concurrent programs [21]. We implement PCMs using two of the Coq's native constructs: *dependent records* and *canonical structures*. We follow the estasblished Ssreflect design pattern of defining algebraic data structures by means of *mix-in* composition [6], whereby different dependent records formalize different algebraic properties, which can be combined using *packed classes* mechanism. The latter also defines the field resolution strategy [7] in a case of overlapping names. For instance, the *mix-in* defining PCMs is represented by the following dependent record:

```
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;
  _ : forall x y, valid_op (join_op x y) -> valid_op x;
  _ : valid_op unit_op }.
```

The type `T` is the *carrier type* of the structure. The field `valid_op` selects a subset of `T`, standing for the "defined" elements. The invalid (or "undefined") elements help model partiality: a partial function over `T` will return some invalid element on an input on which it is mathematically undefined. `join_op` is the binary operation of the PCM, and `unit_op` is the unit element. The remaining five unnamed fields enumerate the axioms that have to be satisfied by each PCM instance.

Next, the mix-in "interface" is packaged with a carrier type, into a dependent record type, which represents PCMs. We also introduce a coercion from the package to the underlying carrier type, so that the two can be conflated. This coercion essentially accounts for the delegation hierarchy from object-oriented languages.

```
Structure pcm : Type :=
    Pack {type : Type; _ : mixin_of type}.
Coercion type : pcm >-> Sortclass.
```

Next, we explain the mechanism of packaging all necessary definitions along with lemmas about data structures (such as *join*'s commutativity and associativity in the case of PCMs) into a single module that should be imported by the clients of the algebraic structure. For example, we introduce appropriate notation for the join operation, and specifically name and prove the lemmas that correspond to the PCM properties that we left unnamed in the mixin.

```
Notation x \+ y := (join_op x y).
Lemma joinC (U : pcm) (x y : U) : x \+ y = y \+ x.
Lemma joinA (U : pcm) (x y z : U) :
           x \+ y \+ z = x \+ (y \+ z).
```

The lemmas such as `joinC` and `joinA` are proved by destructing the package `U`, but notice how the coercion allows conflating `U` with its carrier type. Also notice how the notation `x \+ y` allows the PCM `U` to be ommitted from the equations themselves, as the typechecker can infer it from the context.

Algebraic structures can inherit the properties of other, more basic structures. Thus, we also require an analogue of object-oriented *inheritance*. We illustrate how this can be done in Coq, by defining an interface for a *cancellative* PCM, which inherits from an ordinary PCM. The cancellative PCM is defined as the following mix-in record:

```
Record mixin_of (U : pcm) := Mixin {
  _ : forall a b c: U, valid (a \+ b) ->
                       a \+ b = a \+ c -> b = c }.
```

Notice that the dependent record `mixin_of` in this case is parametrized via the carrier PCM `U`, which is used as a target for a coercion whenever an instance of a plain PCM or a carrier type `U` is required, since coercions a transitive.

3.5.2 Canonical instances of abstract structures

We proceed to show how to *instantiate* the definition of abstract structure with concrete datatypes. It turns out that it is insufficient to merely prove that a datatype satisfies the PCM axioms. To work comfortably with an algebraic structure in practice, one has to explicitly "register" the structure with the type inference engine.

We first show what goes wrong if one doesn't perform the "registration". For instance, assume we first define an instance of a PCM for `nat` with addition, by proving that `+` with `0` satisfies the PCM axioms. Then the following lemma which uses the generic notation `\+` for the PCM operation, is considered ill-formed by Coq. The reason is that Coq cannot figure that there is a PCM associated with `nat`, and that the generic notation `\+` should be resolved with addition. Indeed, we could have defined the PCM for `nat` via multiplication ($\times$) with 1, in which case `\+` should be resolved by $\times$.

```
Lemma add_perm (a b c : nat) :
      a \+ (b \+ c) = c \+ (b \+ a).
```

This is why for any given type such as `nat`, we need to register which structure should be considered as its "default" PCM. We do

so using *canonical instances* mechanism of Coq [17, 28]. In the above case, once a structure is registered as the default PCM for `nat`, the `add_perm` lemma can be proved by selective rewriting using the standard PCM properties.[12]

```
Proof. by rewrite joinA [c \+ _]joinC [b \+ _]joinC. Qed.
```

We conclude this module by presenting a few more examples of algebraic structures. We recal the structure `eqType` of types with decidable equality from (§ 3.3.2), and present Ssreflect's definition of a mix-in for `eqType` (defined in the `eqtype` library). The instances of this structure provide definitions of the boolean equality operation `==`, and proofs that `==` is a decidable version of propositional equality `=`. We exhibit a number of instances for standard datatypes such as `nat` and `bool`. As an exercise in the recitations, the students have to implement an interface for the *partially-ordered set* structure, and to provide a number of canonical instances for it.

3.6 Case study: verifying imperative programs using separation logic in Hoare Type Theory

The last module of the course presents a large case study, which employs all of the Coq programming and proving skills acquired by now: specification and verification of imperative programs in Hoare Type Theory (HTT) [19, 20]. HTT is an implementation of separation logic, formalized as a shallow embedding in Coq. In particular, types are used to implement specifications in the style of Hoare triples. The implementation provides a number of lemmas that correspond to the customary inference rules of separation logic, and are used to interactively establish that an imperative program satisfies a type that corresponds to a Hoare triple. The soundness of HTT is proved with respect to standard state-transformer denotational semantics, which is formalized in Coq.

3.6.1 Introducing Hoare triples using the types analogy

We expected that the students of the course would all have been exposed to Hoare logic in their university education. Nevertheless, to set up the stage, we begin the module by revisiting the main concepts, such as *partial correctness*, notations and rules. In particular, we stress a number of points to build an intuition that Hoare triples are a kind of "types in disguise". This is not a standard way of understanding Hoare logic, but it is reasonably accurate, especially in the case of separation logic. Indeed, an essential property of separation logic is *fault avoidance*—a property that verified programs are memory-safe. This insight can be summarized by the slogan "well-proved programs don't go wrong", which is a slight modification of Milner's motto about typability, generalized to a dependently-typed setting.

Another point relating Hoare triples to types is the rule of consequence, which allows for strengthening the precondition and weakening the postcondition.

$$\frac{P \implies P' \qquad \{P'\}\, c\, \{Q'\} \qquad Q' \implies Q}{\{P\}\, c\, \{Q\}} \text{ (CONSEQ)}$$

The rule features a very similar variance policy as the rule of subtyping for arrow types in the simply typed lambda-calculus with records [25, Chapter 15]:

$$\frac{P <: P' \qquad Q' <: Q}{P' \to Q' <: P \to Q} \text{ (S-ARROW)}$$

3.6.2 Basics of separation logic

After a short tour of Hoare logic, we demonstrate a number of well-known scalability problems that arise in the presence of pointers and aliasing. We proceed to introduce the main ideas behind separation logic [14, 27], focusing mainly on definition of heaps, and explicit heap disjointness.

We expected that our students will not be familiar with separation logic, as it is not typically a topic covered in standard university curricula. This fact embolded us to take a very non-standard approach in our presentation, and in particular, tailor the ideas behind separation logic to Coq-supported mechanized reasoning.

First of all, we ommitted *stack variables* from the presentation. As in functional programming, all our variables will be immutable. If mutation is required, it should be done via heap references. A consequence is that the effectful commands of our programs has to be able to return non-unit results, unlike in separation logic, where no results are returned. Second, we introduce higher-order functions and the fixed-point combinator, therefore removing `while`-loops (and loop invariants), as they can be expressed through recursion. Third, and perhaps most drastically, we avoid using the separation logic's standard *separating conjunction* connective $*$. In the way we will work with Coq, $*$ introduces a level of indirection; the first move in almost all the proofs is to unfold the definition of $*$, and reveal the existential quantification over the disjoint heaps. Instead, we base the specifications on the operation of disjoint heap union. The latter is well-known as the semantic foundation behind $*$ [2], but we found that it works well in practical mechanization too. Finally, we freely use heap-valued variables in our specs, which is not a particularly accepted practice in the world of separation logic, but (1) it works well in Coq, and (2) it also seems unavoidable in the absence of native logical infrastructure to reason about bunches [20].

Therefore, our separation logic specs have the following form, featuring explicitly-quantified initial/final heaps h and the result res, as illustrated by the rules for writing to a pointer x (the rule from reading $!x$ is similar)

$$\{h \mid h = x \mapsto -\}\ x ::= e\ \{\mathsf{res} : \mathtt{unit}, h \mid h = x \mapsto e\} \text{ (WRITE)}$$

and for a pointer allocation

$$\{h \mid h = \mathsf{empty}\}\ \mathtt{alloc}(v)\ \{\mathsf{res}, h \mid h = \mathsf{res} \mapsto v\} \text{ (ALLOC)}$$

Defining an appropriate form of a frame rule is possible, so working with explicit heap and result variables in assertions is pleasant, perhaps surprisingly so.

With only immutable variables allowed, sequential composition has to explicitly account for the results of commands, binding the result to a variable in the continuation.

$$\frac{\{h \mid P(h)\}\ c_1\ \{\mathsf{res}, h \mid Q(\mathsf{res}, h)\} \quad \{h \mid Q(x, h)\}\ c_2\ \{\mathsf{res}, h \mid R(\mathsf{res}, h)\}}{\{h \mid P(h)\}\ x \leftarrow c_1; c_2\ \{\mathsf{res}, h \mid R(\mathsf{res}, h)\}} \text{ (BIND)}$$

One can also *inject* a *pure* expression into a command by means of the ret-statement. The corresponding rule is an equivalent of Hoare-logic rule for variable assignment:

$$\{h \mid P(h)\}\ \mathsf{ret}\ e\ \{\mathsf{res}, h \mid P(h) \wedge \mathsf{res} = e\} \text{ (RETURN)}$$

3.6.3 Effectful computations as monads

A functional programmer will imediately notice that the rules (BIND) and (RETURN), shown in § 3.6.2, bear a lot of similarity with the monadic operations *bind* and *return*, familiar from the Haskell's parametrized type class `Monad`:

```
class Monad m where
  (>>=)            :: m a -> (a -> m b) -> m b
  return           :: a -> m a
```

More specifically, each command in HTT returns a result of some type, similarly to *monadic* programs. Furthermore, commands can be bound by means of the $x \leftarrow c_1; c_2$ syntax, and

[12] The *rewriting selectors*, *e.g.*, `[c + _]`, specify by means of regular expressions, in which subterms of the goal the rewriting should be performed.

the corresponding logic rule checks that the pre/postconditions of c_1 and c_2 agree with each other (modulo the rule of consequence), so they could be chained. Finally, similarly to how pure expressions in Haskell can be embedded into a monad using the `return` command, one can construct commands from expressions using the ret e syntax. All these observations illustrate that there is a correspondence in the style of the Curry-Howard isomorphism between monadic programs and inference rules in Hoare/separation logic, thereby lifting the Curry-Howard theme of the course to stateful programming as well.

3.6.4 HTT essentials

We can now present the main concepts of HTT, such as the notion of monadic *Hoare type* and the notation for writing effectful programs and their specifications in Coq. But before we do so, we remind the students that in general, one cannot implement effectful and potentially non-terminating general recursive computations as pure expressions in Coq. (They have already faced this issue *wrt.* recursion in one of the previous lectures). To encode such computations, similar as in Haskell, one has to encapsulate their side-effects by a monad. However, in the case of Coq and HTT, the monad will be more general than in Haskell. First, it will encapuslate recursion, in addition to the effects related to mutable state, whereas Haskell doesn't consider recursion to be an effect in this sense. Second, the type will include the precondition and the postcondition of the specified computation; thus such monadic types are called Hoare types.

A Hoare type is written using the notation of the form `{x1 x2 ...}, STsep (p, q)`. Here `p` and `q` are heap predicates, standing for the pre and postcondition. More specifically, the type of `p` is `heap -> Prop` and the type of `q` is `A -> heap -> Prop`, where `A` is the type of the result of the command being specified. The identifiers `x1`, `x2` *etc*. bind the logical variables that scope over both `p` and `q`, and are implicitly universally quantified, as customary in Hoare logics. For example, the allocation procedure, mentioned in § 3.6.2 is given the following Hoare type, which is a straightforward rephrasing of its Hoare-style specification with a small (*i.e.*, minimal) heap footprint:[13]

```
alloc : forall (A : Type) (v : A),
          STsep (fun h => h = Unit,
                 [vfun (res : ptr) h => h = res :-> v])
```

3.6.5 Verifying an imperative factorial implementation

We illustrate the HTT verification machinery on an imperative implementation of the factorial procedure (Figure 1). This program will be our main running example for the section. In the course, we first present a paper-and-pencil proof outline for this program (ommitted here). The proof outline lists the assertions that are valid at each program point, and how they have been justified by the inference rules of separation logic. This demonstration introduces the students to how separation logic is used in practice.

Next we show how the same reasoning is implemented in Coq. We start with providing a declarative definition `fact_pure` of a factorial by means of Coq's primitive recursion.

```
Fixpoint fact_pure n :=
 if n is n'.+1 then n * (fact_pure n') else 1.
```

Our goal will be to demonstrate that the procedure `fact` from Figure 1 computes `fact_pure` of its input value. We will also prove that `fact` does not *leak* memory.

To stress the compositionality of the verification, we will break the proof into two parts. We will first verify the recursive loop

[13] We omit the discussion on the distinction between `fun` and `vfun`; it has to do with the treatment of exceptions in HTT, but we don't consider exceptions in the course.

```
1  fun fact (N : nat) : nat = {
2      n   ← alloc(N);
3      acc ← alloc(1);
4      res ←
5        (fix loop (_ : unit).
6            a' ← !acc;
7            n' ← !n;
8            if n' is m' + 1 then
9               acc ::= a' * n';;
10              n   ::= m';;
11              loop(tt)
12           else ret a'
13       )(tt);
14     dealloc(n);;
15     dealloc(acc);;
16     ret res
17 }
```

Figure 1. A pseudocode implementation of an imperative factorial procedure with pointer allocation. The notation `;;` stands for sequential composition without result binding.

function on lines 5-13. After that, we refactor the factorial program so that it invokes the recursive function, rather than inline it. Then we verify that program, reusing the already developed proof for the recursive loop function.

The "loop invariant" of the recursive function is defined as a type `fact_tp` below. It constrains the heap inside the loop to consist of the two pointers: the counter `n` and the accumulator `acc`, storing values `n'` and `a'`, respectively. These facts are stated by using heap values `n :-> n'` and `acc :-> a'`. Heaps form a PCM under the operation of *disjoint union*, so we use the familiar operator `\+` from the lecture on PCMs (§ 3.5) to combine the two single-pointer heaps into a disjoint union. Thus, the verification in HTT in general, and of the factorial example in particular, will rely heavily on the PCM library that the students practiced with previously. Once the loop terminates, the postcondition of the type `fact_tp` says that the counter `n` is decremented to `0`, and the accumulator stores the return result, which equals `a' * fact_pure n'`.

```
Notation fact_tp n acc :=
 {n' a'},
   STsep (fun h => h = n :-> n' \+ acc :-> a',
          [vfun res h => h = n :-> 0 \+ acc :-> res /\
                         res = a' * fact_pure n']).

Program Definition fact_acc (n acc : ptr):
  fact_tp n acc :=
  Fix (fun (loop : unit -> fact_tp n acc) (_ : unit) =>
    Do (a' <-- read nat acc;
        n' <-- read nat n;
        if n' is m'.+1 then
          acc ::= a' * n';;
          n ::= m';;
          loop tt
        else ret a')) tt.
```

One can see that the implementation of `fact_acc` almost exactly matches the pseudocode of the lines 5–13 from Figure 1. The notations `Fix` and `Do` for effectful and generally-recursive computations are provided by HTT. The Coq's command `Program Definition` is similar to the standard definition, except that it allows the expression being defined to have uninstantiated components, which are left as obligations to be filled later [30]. In this particular case, what is omitted is the proof that `fact_acc` meets the type.

We next show how this proof is built. It essentially amounts to the application of the rule (CONSEQ) (§ 3.6.1) to demonstrate that the type automatically computed by Coq for the loop body (and containing the *weakest precondition* and *strongest postcondition* of

the loop body) can be weakened to the explicitly provided type `fact_tp n acc`. Application of (CONSEQ) issues a pair of implications, which can be discharged together by applying a number of helper lemmas that correspond to structural rules in separation logic. All in all, the proof script looks as follows.

```
Next Obligation.
apply: ghR=>_ [n' a'] /= -> _; heval.
case: n'=>[|m'] /=; rewrite ?muln1 ?mulnA; heval=>//.
by do 2![apply: (gh_ex _)]; apply: val_doR.
Qed.
```

While the script is cryptic, it is conceptually straightforward. The first line applies HTT's lemma `ghR`, which applies the rule of consequence, and "pulls out" all logical variables occurring in the specification (*i.e.*, `n'` and `a'` in this case). In Hoare logic parlance, this amounts to an iterated application of the rule of universal quantification (infinitary version of the rule of conjunction). The application of this lemma is followed by some bookkeeping of assumptions and use of HTT's `heval` tactic to symbolically evaluate the program; that is, automatically apply the rules for writing and reading from the pointers. The symbolic evaluation stops at the conditional. Next, `case`-analysis on `n'` considers the two branches of the conditionals: `then`-branch is when the counter `n'` has not reached zero; `else`-branch is when zero has been reached. Both branches are first simplified by rewriting with `mulnA` and `muln1` lemmas. These two lemmas are taken from the standard `ssrnat` library, and express the associativity of multiplication, and neutrality of `1`. After the simplification, the `else`-branch is trivially discharged by symbolic evaluation. The `then`-branch requires application of a procedure call, in this case a recursive call to `loop`. The fixed-point combinator tells us that `loop` has the type `fact_tp n acc`, but we need to instruct the system as to the values of logical variables `n'` and `a'` to be used in the recursive call. We do so by using the lemma `gh_ex` twice; once for `n'` and again for `a'`. In the particular case of `fact_acc`, we don't actually have to provide the values for `n'` and `acc'` explicitly, as the unification mechanism of Coq's will be able to infer them from the equations in the residual subgoals. Thus, we provide underscore `_` as an argument to `gh_ex`. Finally, we apply the lemma `val_doR`, which allows the system to verify that by making the function call to `loop`, under the assumption that `loop` has the type `fact_tp n acc`, as specified by the fixed-point combinator, we can reach the specification for `fact_acc`.

One can see that most of the proof script has to do with bookkeeping technicalities, such as starting symbolic evaluation and naming the variables that arise from case analysis. The only nontrivial insight was that we need to rewrite by `muln1` and `mulnA` at an appropriate place.

We can now specify and verify the full procedure `fact`, which wraps the allocation and deallocation of the around `fact_acc`.

```
Program Definition fact (N : nat) :
  STsep (fun h => h = Unit,
         [vfun res h => res = fact_pure N /\ h = Unit]) :=
  Do (n <-- alloc N;
      acc <-- alloc 1;
      res <-- fact_acc n acc;
      dealloc n;;
      dealloc acc;;
      ret res).
```

The equalities `h = Unit` in `fact`'s pre- and postconditions ensure that the procedure does not leak any memory. The proof of the specification is straightforward and mostly carried out by symbolic evaluation via the `heval` tactic. We omit it here, but it can be found in the lecture notes [29] and the code accompanying them.

3.6.6 Proving specifications with and without automation

Too much reliance on proof automation implemented by unspecified third-party tactics can easily obscure the arguments behind the proof. To avoid such situations in the case of HTT, this module asks the students to perform a few program verifications, without relying on the automation provided by `heval`. The students are supposed to figure out the lemmas applied by `heval` in the course of its run, and apply these lemmas by hand as appropriate.

As an example, we consider a simple program that swaps the natural values of two pointers, `x` and `y`.

```
Program Definition swap (x y : ptr):
 {a b : nat},
  STsep (fun h => h = x :-> a \+ y :-> b,
         [vfun _ h => h = x :-> b \+ y :-> a]) :=
    Do (vx <-- read nat x;
        vy <-- read nat y;
        x ::= vy;;
        y ::= vx).
```

The proof can be carried out by first "pulling out" the logical variables and applying automation by `heval`.

```
Next Obligation. by apply: ghR=> _ [a b]-> _; heval. Qed.
```

However, in a paper-and-pencil proof one would consistently apply a series of the rules, such as (BIND), (WRITE) *etc*. Therefore, we suggest the students to use the `Search` machinery to find the appropriate lemmas and apply them, which leads to the following proof of `swap`'s specification, exactly in the spirit of Hoare-style reasoning.

```
Next Obligation.
apply: ghR => _ [a b]-> _.
apply: bnd_seq; apply: val_read => _.
apply: bnd_seq; apply: val_readR => _.
apply: bnd_seq; apply: val_write => _.
by apply val_writeR.
Qed.
```

Indeed, each line of the proof decomposes a sequential composition by means of (BIND)-like lemma `bnd_seq` and then applies a necessary lemma for writing or reading a pointer, and the variant `val_readR` allows one to avoid making rewritings in the heap to bring it to the "right" shape.[14]

Our introduction to the imperative program verification and HTT is concluded by a series of exercises, including a proof of an imperative Fibonacci procedure and a verification of number of procedures operating on singly-linked lists.

4. Evaluation and experience

A preliminary proof-of-concept evaluation of the described course has been conducted by the first author in a form of a five-days summer school, which took place in Saint Petersburg State University (SPbSU) in August 2014. The school was advertised amongst the senior-years students specializing in mathematics and software engineering. Overall, five students enrolled for the course.

As one of the main prerequisites for the school, we listed good knowledge of functional programming (preferably, based on a statically-typed language, such as ML, Haskell or Scala), which the students of the software engineering chair obtain from the standard curriculum that includes, in particular, a course on Haskell. Passing familiarity with classical propositional logic was desired, and students at SPbSU take such a course during their second or third year. We didn't assume any knowledge of intuitionistic logic or Hoare logic, although simple Hoare logic is usually briefly mentioned in one of the introductory computer science courses at SPbSU.

[14] This problem can be avoided by means of lemma overloading, as demonstrated in [11], but we do not discuss this technique in the course.

Every day of the school featured four hours of lectures, which were mostly delivered in a hands-on mode, with the instructor presenting the material through interaction with Coq. The exceptions were the introductory lecture and parts of the last lecture on separation logic and HTT, which required some amount of theoretical background to be presented beforehand using slides. In the afternoon, four hours were dedicated to solving homework assignments under supervision of the lecturer. Every lecture started from the discussion on the solutions of the exercises from the previous day. The exercises on HTT on the very last day of the lectures were discussed immediately after the lab session.

4.1 Observations

Students had little trouble learning Coq's programming syntax. They quickly mastered writing simple recursive programs, and satisfying the requirements imposed by Coq's termination checker. Some of the students expressed a lot of eagerness to *test* their code. Before being introduced to formal proofs, they started writing "proofs" that correspond to unit tests, such as the one below.

```
Theorem test_plus_3_8 : my_plus 3 8 = 11.
Proof. reflexivity. Qed.
```

Surprisingly, such clever hackers experienced most troubles later on, when making transition from programming to proving, until they finally understood the semantics of proof primitives such as `move` and `case`.

In the process of explaining the basic concepts of the interactive proof (§ 3.1.4), we noticed that, when it comes to applying a tactic, the students often confuse *goals* and *assumptions*. Specifically, some of the student had trouble deciding whether they should use `split` or `case` when a conjunction is encountered in a goal. It helped when we repeated the explanation of the four basic primitives (`exact:`, `move`, `case` and `apply:`), and emphasize that all other proof constructions can be expressed through them.

Being dedicated hackers, about a half of the students often found themselves in a situation that, after some progress with the proof, they no longer understand the proof state. This situation typically occurs when doing proofs by induction (§ 3.4). In such cases, the students often went to the Coq manual in search of a powerful tactic that would solve the problem for them. We tried to prevent these situations by suggesting them to reflect on a problem and construct a paper-and-pencil proof first. For those who insisted on using some standard library tactics (*i.e.*, `inversion`, which usually resulted from the definitions following the traditional Coq style [3, 26] of using indexed type families as GADTs), we allowed them to do so once they were able to explain the outcome through the use of only the basic primitives.

The students found the way we introduced Hoare-style reasoning about imperative programs out of the type analogy (§ 3.6.1) to be quite natural. Moreover, right after the scalability problems of the traditional Hoare logic were listed (*i.e.*, the proof burden appearing when reasoning about aliasing), and before the introduction of separation logic, one of the students has immediately suggested to make the fact of heap disjointness to be *explicit* in the assertions, which is the crucial idea of the separation logic.

We were pleasantly surprised to find some of the students' proofs of the exercises (§ 3.2, § 3.4) were shorter and conceptually simpler than those we developed ourselves as model solutions.

4.2 Feedback from the participants

In the post-school anonymous evaluation, we received largely positive feedback. The course has been praised for a selection of topics with an emphasis on verification of imperative programs, which some of the students found to be an "impressive piece of real-world research project that can be taught in a comprehensible way".

One major criticism we got had to do with a way the proofs were presented in the files accompanying the lectures and used for the hands-on sessions. In the later lectures we have "compressed" some of the proof steps into non-atomic lines (*e.g.*, several rewritings followed by the bookkeeping machinery). Although not complex conceptually, some students found such proofs hard to follow without breaking the lines into more atomic steps. And while doing so, they would lose the pace of the lecture, and had to ask the lecturer to start the example over. We intend to revise the code supporting the course to avoid this problem in the future.

5. Related courses and future work

The Coq proof assistant has been in development since 1983, and by now there are a number of courses that provide introductions to Coq-powered proving and programming. The book *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions* by Yves Bertot and Pierre Castéran [1] is an exhaustive overview of Coq as a formal system and tool, covering both logical foundations, reasoning methodologies and automation, and offering large number of examples and exercises. Benjamin Pierce *et al.*'s *Software Foundations* electronic book [26] introduces Coq development from an angle of the basic research in programming languages, focusing primarily on formalization of language semantics and type systems, which serve both as main motivating examples of Coq usage and as a source of intuition for explaining Coq's logical foundations. The most recently published book, *Certified Programming with Dependent Types* by Adam Chlipala [3] provides an 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.

While designing the course, we have drawn a lot of inspiration from these books, from which we also borrowed a number of examples and exercises. However, we often had to redesign these examples and exercises in order to keep to the explicit but minimalistic Ssreflect style of proof, based on the small set of core tactics, and in order to emphasize the computational nature of properties being defined and verified.

Initially, we intended to adopt some of the teaching insights from Henz and Hobor's (H&H) experience report [12]. Alas, a majority of the practices described there are quite specific to the teaching program of National University of Singapore's School of Computing (in particular, some background on modal logic was assumed). The introduction into formal proofs in H&H's course is made through the Aristotles term logic, whereas we deliberately strived to employ the students' intuition acquired from the courses on functional programming. Both approaches have their strengths. In particular H&H's seems to be more approachable by students with no background in formal methods. On the other hand, our course is at freedom to employ a richer vocabulary of programming analogies, making it easier to explain the notions such as inductive proof, dependent records and Hoare types. Noteworthy, H&H notice the same anti-patterns, exhibited by the students during the learning process, in particular, the undue "hacking" with tactics.

In the future, we plan to extend the course with a discussion on co-fixpoints and proofs by coinduction, supported by recent advances on the mechanized proof construction in this area [13]. We also intend to enhance the course with a survey of proof automation techniques by means of of tactic engineering [31, 34] or lemma overloading via canonical structures [11, 17].

6. Conclusion

Programming and proving are the two sides of the same coin: construction of scalable and reusable formal proofs can be effectively taught basing on the knowledge of constructing scalable software systems. In this report, we have described the design of an experimental course on mechanized reasoning in the Coq proof assistant, which embraces the proving-as-programming insight. The particular cases of this duality, which we have instantiated as teaching principles in a class, are the computational approach to decidable properties, boolean reflection, implementation of algebraic structures as dependent records with inheritance, and reasoning about effectful programs in a Hoare-style program logic.

It is our belief that the proposed approach to teaching mechanized theorem proving and formal reasoning can be adopted in educational programs for students, who already have background in software engineering and program design, and we support this claim by reporting on our preliminary teaching experience, which is positive.

All the materials of the course, including the lecture notes, exercises, source files for the hands-on lectures, and necessary libraries, are available online [29].

Acknowledgments

We are grateful to Dmitri Boulytchev for his initiative to organize a summer school on dependent types at SPbSU. We also thank Anindya Banerjee, Olivier Danvy, Michael D. Ernst, Rémy Haemmerlé, José Francisco Morales and Éric Tanter for their comments on the lecture notes and on earlier drafts of this article.

References

[1] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Springer Verlag, 2004.

[2] C. Calcagno, P. W. O'Hearn, and H. Yang. Local action and abstract separation logic. In *LICS*. IEEE Computer Society, 2007.

[3] A. Chlipala. *Certified Programming with Dependent Types*. The MIT Press, 2013.

[4] Coq Development Team. The Coq Proof Assistant – Reference Manual, Version 8.4pl4, July 2014. Available at `http://coq.inria.fr/refman/`.

[5] P.-E. Dagand and C. McBride. Transporting functions across ornaments. In *ICFP*. ACM, 2012.

[6] F. Garillot. *Generic Proof Tools and Finite Group Theory*. PhD thesis, École Polytechnique, Palaiseau, France, 2011.

[7] F. Garillot, G. Gonthier, A. Mahboubi, and L. Rideau. Packaging mathematical structures. In *TPHOLs*, volume 5674 of *LNCS*. Springer, 2009.

[8] G. Gonthier. Formal Proof – The Four-Color Theorem. *Notices of the American Mathematical Society*, 55(11), Dec. 2008.

[9] G. Gonthier, A. Mahboubi, and E. Tassi. A Small Scale Reflection Extension for the Coq system. Technical Report 6455, Microsoft Research – Inria Joint Centre, 2009.

[10] G. Gonthier, A. Asperti, J. Avigad, Y. Bertot, C. Cohen, F. Garillot, S. Le Roux, A. Mahboubi, R. O'Connor, S. Ould Biha, I. Pasca, L. Rideau, A. Solovyev, E. Tassi, and L. Théry. A Machine-Checked Proof of the Odd Order Theorem. In *ITP*, volume 7998 of *LNCS*. Springer, 2013.

[11] G. Gonthier, B. Ziliani, A. Nanevski, and D. Dreyer. How to make ad hoc proof automation less ad hoc. *Journal of Functional Programming*, 23(4), 2013.

[12] M. Henz and A. Hobor. Teaching experience: Logic and formal methods with Coq. In *CPP*, volume 7086 of *LNCS*. Springer, 2011.

[13] C.-K. Hur, G. Neis, D. Dreyer, and V. Vafeiadis. The power of parameterization in coinductive proof. In *POPL*. ACM, 2013.

[14] S. S. Ishtiaq and P. W. O'Hearn. BI as an assertion language for mutable data structures. In *POPL*. ACM, 2001.

[15] G. Klein, J. Andronick, K. Elphinstone, T. Murray, T. Sewell, R. Kolanski, and G. Heiser. Comprehensive Formal Verification of an OS Microkernel. *ACM Transactions on Computer Systems*, 32(1): 2:1–2:70, 2014.

[16] X. Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *POPL*. ACM, 2006.

[17] A. Mahboubi and E. Tassi. Canonical Structures for the Working Coq User. In *ITP*, volume 7998 of *LNCS*. Springer, 2013.

[18] P. Morris, T. Altenkirch, and N. Ghani. A universe of strictly positive families. *International Journal of Foundations of Computer Science*, 20(1), 2009.

[19] A. Nanevski, J. G. Morrisett, and L. Birkedal. Hoare type theory, polymorphism and separation. *Journal of Functional Programming*, 18(5-6), 2008.

[20] A. Nanevski, V. Vafeiadis, and J. Berdine. Structuring the verification of heap-manipulating programs. In *POPL*. ACM, 2010.

[21] A. Nanevski, R. Ley-Wild, I. Sergey, and G. A. Delbianco. Communicating state transition systems for fine-grained concurrent resources. In *ESOP*, volume 8410 of *LNCS*. Springer, 2014.

[22] U. Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, September 2007.

[23] C. Paulin-Mohring. Inductive Definitions in the System Coq—Rules and Properties. In *TLCA*, volume 664 of *LNCS*. Springer-Verlag, 1993.

[24] S. L. Peyton Jones, D. Vytiniotis, S. Weirich, and G. Washburn. Simple unification-based type inference for GADTs. In *ICFP*. ACM, 2006.

[25] B. C. Pierce. *Types and Programming Languages*. The MIT Press, 2002.

[26] B. C. Pierce, C. Casinghino, M. Gaboardi, M. Greenberg, C. Hriţcu, V. Sjöberg, and B. Yorgey. *Software Foundations*. Electronic textbook, 2014. Available at `http://www.cis.upenn.edu/~bcpierce/sf`.

[27] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*. IEEE Computer Society, 2002.

[28] A. 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, Université Paris VI, Paris, France, 1999.

[29] I. Sergey. *Programs and Proofs: Mechanizing Mathematics with Dependent Types*. Lecture notes with exercises, 2014. Available at `http://ilyasergey.net/pnp`.

[30] M. Sozeau. Subset Coercions in Coq. In *TYPES*, volume 4502 of *LNCS*. Springer, 2006.

[31] A. Stampoulis and Z. Shao. VeriML: typed computation of logical terms inside a language with effects. In *ICFP*. ACM, 2010.

[32] P. Wadler and S. Blott. How to Make ad-hoc Polymorphism Less ad-hoc. In *POPL*. ACM Press, 1989.

[33] H. Xi, C. Chen, and G. Chen. Guarded recursive datatype constructors. In *POPL*. ACM, 2003.

[34] B. Ziliani, D. Dreyer, N. R. Krishnaswami, A. Nanevski, and V. Vafeiadis. Mtac: a monad for typed tactic programming in Coq. In *ICFP*. ACM, 2013.