CS 2800 Logic and Computation
Lecture Notes, Fall 2022

Stavros Tripakis

December 6, 2022

1 Software

Our modern societies heavily depend on software, and this dependence is likely to grow. Software is important, and it is also beautifully complex. It is complex first because of its sheer size: estimates in 2015 placed Google's software at about 2 billion lines of code, and Microsoft's Windows operating system at about 50 million lines of code[1]; in 2017 a pacemaker had about 100 thousand lines of code, the Boeing 787 airplane had more than 10 million, and a high-end car had about 100 million[2]; some estimates place the size of new software produced every year to the hundreds of billions of lines of code[3].

But even very small programs can be extremely complex. The famous *Collatz conjecture* states that the following program terminates for all possible inputs:

```
n := input a natural number;
while (n > 1)
  if (n is even)
    n := n/2;
  else
    n := 3*n + 1;
```

The Collatz conjecture is an open problem.[4] It is a *conjecture* (i.e., something we believe is true), but not a *theorem* (i.e., not something we have proven). The fact that this 6-line program defies the understanding of even our best mathematicians tells us that there is something inherently complex and challenging about software. Software is the most complex artifact that humans have ever constructed. Understanding software is an important intellectual challenge for humanity.

2 Software science

Science is knowledge that helps us make predictions. The keyword is *predictions*. The stronger the science, the stronger the predictions it can make. Software science helps us make predictions about the programs that we write. Will my program terminate? Is my program correct? What does correct even mean? Will my program produce a correct output? When exactly is an output correct? Should the input satisfy any conditions in order for the output to be correct? Is my program secure? Is my program fair, or is it biased? Etc.

[1]According to https://www.wired.com/2015/09/google-2-billion-lines-codeand-one-place/.

[2]According to https://www.visualcapitalist.com/millions-lines-of-code/.

[3]According to https://cybersecurityventures.com/application-security-report-2017/.

[4]If you solve it, you will become famous. See also: https://www.quantamagazine.org/can-computers-solve-the-collatz-conjecture-20200826/ (thanks to Samuel Lowe for suggesting this article).

3 This course

Program correctness by testing and proving: This course is an introduction to the science of software. You have already written programs. You have taken and will take more courses that teach you how to program. In this course you will learn to *reason about programs*. In particular, you will learn to reason about program *correctness*.

In most programming courses you will focus on checking program correctness by *testing*. Testing is very important, but as Dijkstra[5] famously said: *"Program testing can be used to show the presence of bugs, but never to show their absence!"*

In this course we focus on *proving* program correctness. Proving is a stronger guarantee than testing. Testing checks only some inputs, whereas a proof is usually about all possible inputs. So proofs offer stronger predictions about our programs.

Logic: But in order to prove that a program is correct, we must first define what exactly we mean by *correct*. For that, we will use logic. Logic is first of all a language. Contrary to natural languages (English, Greek, etc.), logic is precise and unambiguous. We can debate endlessly about politics and the meaning of life, but the meaning of a logical formula is not a matter of opinion. It is mathematically defined. This is very important because it helps avoid errors of communication. Miscommunication can be catastrophic in life, but also in engineering projects.

Logic is also a reasoning device. It is a set of rules that allow us to say things like *if ABC is true, then XYZ must also be true*. In this course we will use logic both as a language and also as a reasoning device.[6]

Specification and verification: We will use logic to express properties of programs. Collectively these properties define what it means for a program to be correct: they *specify* the program. This is called program **specification**. We will also use the rules of logic to prove those properties. Proving that a program satisfies its specification is called program **verification**.

In this course we will learn:

- to read functional programs with types
- to write functional programs with types
- to read formal specifications written in logic
- to write formal specifications in logic
- to read proofs
- to write proofs.

LEAN: In this course, we will use the LEAN theorem prover: `https://leanprover.github.io/`. We will write programs in LEAN's programming language, we will write specifications in LEAN's logic, and we will write proofs using LEAN's proof system.

Install LEAN on your personal computer as soon as you read this:

IMPORTANT: YOU SHOULD INSTALL LEAN 3, NOT LEAN 4!!!

We found most helpful the instructions provided here: `https://leanprover-community.github.io/`.

[5] `https://en.wikipedia.org/wiki/Edsger_W._Dijkstra`

[6] Logic goes far beyond what we will see in this course. Logic is the foundation of mathematics. It is also the foundation of language, reason, and philosophy. It is also the foundation of truth. Logic is a fascinating subject, but this course is not really *about* logic, despite the title. This course is using logic but it is not studying logic itself. A more accurate title for this course might be *Introduction to Formal Specification and Verification*, or *Introduction to the Science of Software*, or something like that.

Other theorem provers: The goal of this course is *not* to teach you LEAN. The goal is to introduce you to the science of software, formal logic, formal specification, and formal verification. We are using LEAN as a tool and as a means to an end, rather than the end itself.[7] LEAN is just one of many tools that could be used for this purpose. Examples of other such tools are (in alphabetical order):

- ACL2s: `http://acl2s.ccs.neu.edu/`
- Agda: `https://wiki.portal.chalmers.se/agda/`
- Coq: `https://coq.inria.fr/`
- Idris: `https://www.idris-lang.org/`
- Isabelle: `https://isabelle.in.tum.de/`
- PVS: `https://pvs.csl.sri.com/`

The above list is by no means exhaustive. This is an active area of research, and new tools are being developed or new capabilities are added to existing tools all the time. Each tool has its own pros and cons, just like different programming languages and systems have their own pros and cons. Nevertheless, some basic concepts and principles are common to all these tools. It is these concepts and principles that we strive to teach you in this course, and it is these concepts and principles that you should strive to learn.

Having fun with proofs: Proving theorems with a tool like LEAN is a lot of fun. It's like playing a game. The goal of the game is to prove the theorem. This is like solving a puzzle, or finding our way out of a maze. We will learn which moves to make to help us find the exit of the maze (if such an exit exists!). WARNING: this game can become addictive!

How to succeed in this course: We learn by experimenting, asking questions, and making mistakes. Making mistakes is great (as long as they are not catastrophic mistakes, like drinking and driving and car crashing). Fortunately, computer science provides you with a very safe environment for making mistakes: the worst that can happen is that your program doesn't compile, or that it doesn't return the right result. No big deal!

In this course, what can go wrong? Maybe LEAN does not accept your function definition and you don't see why. Or maybe your function doesn't work as expected. Or you cannot complete a proof.[8] Etc. Try to experiment to see what goes wrong. For LEAN specific things, consult the LEAN documentation. Ask questions when you are deadlocked. *Asking questions is fine and expected of you!* Ask questions in class, on piazza, during office hours, etc. *There are no stupid questions.*

A good way to know whether you are learning what you are supposed to be learning is whether you are *able to do all the homework problems by yourself.* If you are, you will do well in the course. If you are not, you should be worried. Come to our office hours if you are worried.

4 These lecture notes

These lecture notes will be sparse. This is intentional. They are not meant to be a textbook, but rather a guide for the course (like a map). The philosophy of the course is *learning by doing.* Is there any other way to learn really?

[7]You cannot learn to bike without a bicycle. Once you learned to bike, you can use pretty much any bicycle. LEAN is our bicycle.

[8]Sometimes you may complete the proof and later find out that what you proved was not really what you meant or what we asked you to prove. This is fine. It's part of learning to write formal specifications and statements in logic. It's also about appreciating the need for precision and non-ambiguity.

In particular, these lecture notes are *not* about learning LEAN. There are many many good resources on LEAN freely available online: examples, tutorials, online textbooks, and many more. Some references to those are provided below.

These lecture notes will be permanently under construction. They will be updated regularly as we advance in the course. The latest version (available at the course web page) will serve as the reference point. Please look at the date of these notes, compare it to the date in your own copy, and use the latest version.

5 Other reading

Documentation on LEAN: There is a lot of documentation available on LEAN from the following web sites:

- `https://leanprover.github.io/`
- `https://leanprover-community.github.io/`
- `https://leanprover-community.github.io/papers.html`

Unfortunately, there is no single document that matches exactly what we present in this course, so you will have to collect information from multiple sources.[9] Also, much of the LEAN documentation is under construction and/or incomplete. We recommend starting with this (although there is a lot from the link below that we will *not* cover/emphasize in this course, like type theory, dependent types, etc., for instance):

- `https://leanprover.github.io/theorem_proving_in_lean`

You can also consult the reference manual (unfortunately the programming part is missing):

- `https://leanprover.github.io/reference/`

You can also look directly at the LEAN code, libraries, etc.:

- `https://github.com/leanprover/lean/tree/master/library/init`

For those interested in using LEAN for formal mathematics, here's a couple of links:

- `https://leanprover-community.github.io/mathematics_in_lean/index.html`
- `https://www.quantamagazine.org/building-the-mathematical-library-of-the-future-20201001/`, `https://www.quantamagazine.org/lean-computer-program-confirms-peter-scholze-proof-20210728/` (thanks to Samuel Lowe for suggesting these articles).
- *The Future of Mathematics?* talk by Kevin Buzzard: `https://www.youtube.com/watch?v=Dp-mQ3HxgDE` (thanks to William Schultz for suggesting this link).

Type Theory: LEAN is based on so-called *type theory* which studies *type systems*. LEAN has a type system, and many (typed) programming languages also have type systems. Type systems are fundamental in programming (languages), but also in logic and the foundation of mathematics. However, we will not study type systems nor type theory in this class, as our main focus is to learn how to do proofs by doing. Those interested in type theory can consult relevant courses in programming languages, or the references below:

- *Types and Programming Languages*, by Benjamin C. Pierce.
- *Advanced Topics in Types and Programming Languages*, by Benjamin C. Pierce, editor.
- A short introduction to LEAN's type system can be found here: `https://leanprover.github.io/theorem_proving_in_lean/dependent_type_theory.html`. More details on it can be found in Mario Carneiro's *The Type Theory of Lean*, available from `https://leanprover-community.github.io/papers.html`.

[9]This is also what you will have to do in your "real life" outside the university.

Software Foundations: https://softwarefoundations.cis.upenn.edu/. *Software Foundations* is a book series available online. It goes much further than we do in this course, but its first part (Volume 1) serves as good reading material for this course. *Software Foundations* uses a different theorem prover, called Coq. LEAN is quite similar to Coq, and you should be able to follow and re-do most of the things described in *Software Foundations* in LEAN. We often borrow exercises from *Software Foundations* and adapt them to our course. We thank the authors of *Software Foundations* for making the series freely available.

Other Courses: In addition to the *Software Foundations* online series, there is a number of courses available online which are related to our course. Here's a partial list for those interested:

- *Logic and Proof*, at CMU: https://leanprover.github.io/logic_and_proof/. This course is also based on LEAN.
- *Logical Verification*, at Vrije Universiteit Amsterdam: https://lean-forward.github.io/logical-verification/2021/. This course is also based on LEAN.
- *Semantics of Programming Languages*, at TU Munich: http://www21.in.tum.de/teaching/semantik/WS1920/. This course is based on another theorem prover called Isabelle.
- *Formal Reasoning About Programs*, at MIT: http://adam.chlipala.net/frap/. This course is based on Coq.

A formal methods course database is available here: https://fme-teaching.github.io/courses/

Summer Schools and Seminars: There are also regularly held summer schools and other seminars on logic and related formal techniques:

- See the *Speaking Logic* material by Natarajan Shankar (http://fm.csl.sri.com/SSFT21/speaklogicV10.pdf) as part of the Summer School on Formal Techniques: https://fm.csl.sri.com/SSFT22/.
- *Vistas in Verified Software*: https://www.newton.ac.uk/event/vs2w01/
- *World Logic Day 2022:* https://logicday.vcla.at/ – and you can sing *All you need is lo...*! See https://logicday.vcla.at/vienna-logic-day-lecture/ for list of Vienna Logic Day Lectures. Other related talks by Moshe Vardi: *From Aristotle to the iPhone* – https://www.youtube.com/watch?v=wOQuW6QFdos; *Technology Is Driving the Future, But Who Is Steering?* – https://www.youtube.com/watch?v=fL93WT3vy-0.
- *On the unusual effectiveness of logic in computer science:* see [8]. Slides and paper freely available online as PDFs.
- For more summer schools, see this list: http://user.it.uu.se/~bengt/Info/summer-schools.shtml.

Textbooks: *THERE IS NO REQUIRED TEXTBOOK FOR THIS COURSE.* For those interested in learning more about logic and its use in computer science in general and specification/verification in particular, here are some textbooks:

- *Logic in Computer Science: Modelling and reasoning about systems*, by Huth and Ryan [12].
- *Mathematical Logic for Computer Science, 3rd Edition*, by Mordechai Ben-Ari [2].
- *Handbook of Practical Logic and Automated Reasoning*, by Harrison [9].
- *The Calculus of Computation - Decision Procedures with Applications to Verification*, by Bradley and Manna [4].

For those interested in learning more about verification and formal methods:

- *Model Checking*, by Clarke, Grumberg and Peled [5].
- *Principles of Model Checking*, by Baier and Katoen [1].
- Several books on the *SPIN Model Checker*, by Holzmann [10, 11].
- Books by Manna and Pnueli: *The Temporal Logic of Reactive and Concurrent Systems: Specification*, *Temporal Verification of Reactive Systems: Safety*, and *Temporal Verification of Reactive Systems: Progress* (the third is available online as an unpublished draft) [14, 15].
- *Handbook of Model Checking*, by Clarke, Henzinger, Veith, Bloem [6].

Other relevant books are the following:

- *Computer-Aided Reasoning: An Approach* by Kaufmann, Manolios and Moore [13].
  See `https://www.cs.utexas.edu/users/moore/publications/acl2-books/car/index.html`.
- *Coq'Art* by Bertot and Castéran [3].
- *Certified Programming with Dependent Types* by Adam Chlipala.
  Available at `http://adam.chlipala.net/cpdt/`.
- *Formal Reasoning About Programs* by Adam Chlipala.
  Available at `http://adam.chlipala.net/frap/`.
- *Isabelle/HOL – A Proof Assistant for Higher-Order Logic* by Nipkow et al [16].
- *Concrete Semantics with Isabelle/HOL* by Nipkow and Klein.
  Available at `http://www.concrete-semantics.org/`.
- *Functional Algorithms, Verified!* by Nipkow et al.
  Available at `https://functional-algorithms-verified.org/`.

The history of logic, in comics: The following is a wonderful book on the history of logic and foundations of mathematics, written by famous computer scientist Christos Papadimitriou:

- *Logicomix: An Epic Search for Truth*, by Papadimitriou, Doxiadis and Papadatos [7].

An illustrated book of bad arguments: The following is a very nice book too: `https://bookofbadarguments.com/`.

6 Research

The core topics dealt with in this course are specification and verification. These are currently very active areas of research, with dozens of conferences and other events organized and hundreds of papers published every year. These are some of the main events (a very very partial list):

- CAV: `http://i-cav.org/`
- POPL: `https://popl22.sigplan.org/`
- TACAS: `https://etaps.org/2022/tacas`
- FMCAD: `https://fmcad.org/`

Competitions: Research in this area is aided by tool competitions. There are several competitions, depending on the specific problem ("sport") of interest. Here's some (again this is a partial list):

- The SAT competition: `http://satcompetition.org/`. See also: `http://www.satlive.org/`.
- The SMT competition: `https://smt-comp.github.io/`
- The Software Verification competition: `https://sv-comp.sosy-lab.org/`
- The Hardware Model Checking competition: `http://fmv.jku.at/hwmcc20/`
- VerifyThis: `https://www.pm.inf.ethz.ch/research/verifythis.html`
- The Verification of Neural Networks Competition: `https://sites.google.com/view/vnn2021/home`
- The Reactive Synthesis Competition: `http://www.syntcomp.org/`
- The Syntax-Guided Synthesis Competition: `https://sygus.org/`
- And many more! See: `https://alastairreid.github.io/verification-competitions/`

7 Course outline

Populated as we go along.

7.1 Introduction

Lecture 1. Module 01 on canvas. Slides: `01-intro.pdf`.

- Course goals and logistics.
- A glimpse into LEAN.
- Introductions.
- Homework 01: your first proof!

7.2 Functional programming with types in LEAN

Lectures 2-5. Modules 02 and 03 on canvas. Lecture code: `01-code.lean, 02-code.lean, 03-code.lean, 04-code.lean, 05-code.lean`.

- Basic expressions, predefined operations and types in LEAN.
- `#eval`, `#reduce`, `#check`, `#print`
- Defining simple non-recursive functions in LEAN.
- Strong typing, type errors, and function types as input-output contracts.
- Predefined types `bool`, `nat`, `int` and `list nat`.
- Defining functions using pattern-matching.
- Recursive functions on `nat` and `list nat`.
- A word about termination.
- Anonymous functions (lambda abstraction).

- Functions and types as first-class citizens.
- Homework 02.
- Product types and currying.
- Booleans and functions on booleans.
- Defining our own types.
- Inductive data types and constructors.
- Finite types: the type `weekday`.
- Infinite types: the type `mynat`.
- Defining recursive functions on inductive data types by pattern matching: redefining addition on `nat`s.

7.3 Testing as proving

Lecture 4. Module 03 on canvas. Lecture code: `04-code.lean`.

- Writing regression tests as "mini-theorems" using `example`.
- Introduction to proofs.
- The LEAN proof environment.
- The proof state, goals, and hypotheses.
- The `reflexivity` tactic.
- Tests = simple proofs.

7.4 Introduction to formal specifications

Lecture 5. Module 03 on canvas. Lecture code: `05-code.lean`.

- The type `Prop`.
- Properties and specifications.
- Informal and formal specifications.
- `example`, `lemma`, `theorem`.
- `sorry`.
- Writing specifications with the universal quantifier `forall` (∀).
- Formal verification and formal proofs.
- The `intro` tactic for ∀.
- `try`.
- Homework 03.

7.5 Proof by simplification, proof by cases, and introduction to logic

Lectures 6-7. Module 04 on canvas. Lecture code: `06-code.lean, ourlibrary06.lean, 07-code.lean`. Slides: `07-propositional-logic.pdf`.

- Proof by simplification (a.k.a. "equational reasoning").
- The `dunfold` tactic.
- The `rewrite` tactic.
- The `intros` tactic.
- What it means for a tactic to *apply* to a given proof state.
- Importing library files with `import`.
- More forall specifications.
- Things that we can and things that we cannot prove by simplification, and why.
- Proof by cases.
- The `cases` tactic.
- Nested cases.
- LEAN's `import`.
- Introduction to logic: a brief history of logic, and a review of propositional logic: boolean expressions, truth tables, satisfiability, validity, stronger, weaker, equivalent formulas.
- Logical connectives: negation, conjunction, disjunction, implication.
- The tactic `intro` again, this time for implication.
- The tactics `assumption` and `exact`.
- Homework 04.

7.6 Dealing with logical connectives; Calling lemmas and theorems

Lectures 8-9. Module 05 on canvas. Lecture code: `08-code.lean, 09-code.lean`.

- Recap: where we stand.
- The tactic `trivial`: anything implies `true` and `false` implies everything.
- Soundness and completeness.
- Proving disjunctions in the goal: the tactics `left` and `right`.
- Proving conjunctions in the goal: the tactic `split`.
- Dealing with disjunctions and conjunctions in the hypotheses: the tactic `cases`, again.
- Tactics as legal (but not always good) moves in a game.
- Negation is an implication.
- If-and-only-iff (iff) is a conjunction.

- Exclusive-OR (xor) is a disjunction.
- `repeat`.
- `bool` vs `Prop`, equality = vs iff ↔.
- Proving propositional logic tautologies in LEAN using `bool`s vs `Prop`s.
- Propositions are types! Theorems are functions that return proofs!
- Modus ponens: the tactic `have`.
- Calling lemmas and theorems.
- `rewrite` revisited: rewriting based on equalities or equivalences.
- Homework 05.

7.7 Constructive vs. classic logic; Provability vs Semantic truth; Homework review

Lectures 10 and 11. Module 06 on canvas. Lecture code: `10-code.lean, 11-code.lean, ourlibrary11.lean, ourlibrary12.lean`. Slides: `11-deductions.pdf`.

- Constructive vs. classic logic.
- The axioms `classical.em` (law of excluded middle) and `classical.by_contradiction`.
- `cases` on lists.
- `have` revisited: nested proofs.
- unfolding and rewriting `at` hypotheses.
- The power of `rewrite`.
- The dangers of overlapping.
- Simplifying equalities with constructors: the lemmas `succeq` and `listeq`.
- Tactics and deduction systems.
- Provability vs Semantic truth.
- Soundness and completeness revisited.
- Formal deductions by hand.
- Review: homeworks 03, 04, 05, 06.
- Homework 06: practice for the exam!
- Homework 07 (part): practice for the exam!

7.8 Exam 1 – 24 Oct 2022

Taken online using Gradescope. Material: everything covered so far (up to and including §7.7 above).

7.9 Induction, and Exam 1 and Homework Review

Lectures 12, 13, 14, 15, 16, 17, 18. Module 07 on canvas. Lecture code: `12-code.lean`, `13-code.lean`, `ourlibrary13.lean`, `14-code.lean`, `ourlibrary14.lean`, `15-code.lean`, `16-code.lean`, `ourlibraryite.lean`, `ourlibrary16.lean`, `17-code.lean`, `18-code.lean`.

- Proofs by induction. Base case. Induction step. Induction hypothesis.
- Expressing and using induction on natural numbers in LEAN: `nat_induction`.
- The `induction` tactic.
- Induction on lists.
- Homework 07 (part).
- Proof by induction vs proof by cases.
- Induction is only for inductive types.
- Induction for arbitrary inductive types.
- Multiple base cases.
- Multiple induction steps.
- Multiple induction hypotheses. Induction on trees.
- Lemma discovery.
- Generalization.
- Choosing the induction variable.
- Induction and hypotheses.
- Homework 08.
- Review of Exam 1.
- Simplifying `if-then-else`s: the lemmas `itetrue` and `itefalse`.
- Delaying introductions.
- The tactic `revert`.
- Homework 09.
- Review: homeworks 07, 08.
- How not to lose your `bool` cases.
- The tactic `clear`.
- LEAN's `notation`.
- Hints for homework 09 and beyond.

7.10 Functional Induction, Proof Automation, Undecidability, SAT Solvers

Lectures 19 and 20. Module 08 on canvas. Lecture code: `19-code.lean, 20-code.lean`.
Slides: `20-undecidability.pdf`. Papers: `terminator.pdf`, `MalikCACM.pdf`.

- Functional induction.
- The hardness of proving theorems.
- The hardness of proving software correctness.
- Alan Turing.
- Undecidability.
- Proof automation.
- The satisfiability problem.
- SAT solvers.
- Homework 10.

7.11 Proving Termination, Measure Functions

Lectures 21, 22, 23. Module 09 on canvas. Lecture code: `21-code.lean, 22-code.lean, 23-code.lean`.
Slides: `21-termination.pdf, 22-measures.pdf`.

- The hardness of checking termination.
- Proving termination.
- Measure functions.
- Homework 11.
- Review: homework 10.

7.12 Exam 2 – Wed 30 Nov 2022

Taken online using Gradescope. Material: everything up to and including measure functions.

7.13 Going further

Lectures 24, 25, 26. Module 10 on canvas. Lecture code: `24-code-kop.lean, 25-code.lean, 26-exists.lean`.

- INDUCTION REVISITED: jumping every two steps.
- Convincing LEAN that a function terminates.
- Dealing with the ∃ quantifier.
- Review: homework 11, exam 2.
- The science of software and other stories.

8 Summary of LEAN proof tactics and their justification

We say that a tactic *applies* to a given proof state S when LEAN returns no error when we issue that tactic at state S. The list of tactics given below also summarizes the conditions under which each tactic applies.

When a tactic applies to a certain proof state S_1, it transforms S_1 to a proof state S_2, such that *proving S_2 suffices (is enough) to prove S_1*. What this means is that if we can complete the proof from the new proof state S_2, then we have completed the proof from S_1 as well. To complete the proof means to reach a LEAN proof state that says **goals accomplished**.

Here's a summary of the proof tactics that we have learned so far in this course:

1. `reflexivity`, abbreviated `refl`: applies when the goal is of the form $A = A$ (or can be "easily" simplified/reduced to $A = A$) and transforms the proof state to 'goals accomplished'.

   Intuition/justification: "A is identical to A, for all A" is an axiom of logic, called *reflexivity of equality*.

   Sometimes `refl` also applies to goals that are of the form $A = B$, where A and B are not identical, but are such that one can be reduced to the other after performing 'computations' (*reductions*).

2. `intro`: applies when the goal is of the form $\forall x : T, ...$; eliminates $\forall$-quantified variable x from goal and introduces $x : T$ into the hypotheses.

   Also applies when goal is of the form $P \to Q$: turns goal into Q and introduces P in the hypotheses.

   Intuition/justification: If I have to prove something like $\forall x : T, P$, it suffices to prove P assuming x is an arbitrary element of type T. And if I have to prove $P \to Q$ then it suffices to prove Q assuming P holds.

   `intro` y renames the variable into y.

   `intros`: repeatedly applies `intro`. Can also be called as `intros` $x\ y\ z$ ..., to give the desired names to the introduced objects.

3. `dunfold` f: simplify/reduce function applications of the form $(f\ e)$ based on the definition of the given function f. If we add `at` H at the end, then the tactic applies to hypothesis H, instead of the goal.

   Intuition/justification: If I have to prove P, and $(f\ e)$ appears somewhere in P, and $(f\ e) = g$ for some g, then it suffices to prove P where $(f\ e)$ is replaced by g.[10]

4. `rewrite [<-] H`: rewrites the goal based on the equality or equivalence H. H could be a function, a hypothesis, or a previously proven lemma/theorem.

   By default rewrites from left to write. If `<-` is added, rewrites from right to left. Abbreviated `rw`.

   If we add `at` D at the end, then the tactic applies to hypothesis D, instead of the goal.

   Intuition/justification: If I rewrite based on a proven equality $A = B$, then in order to prove goal G it suffices to prove G' which is obtained from G by substituting any occurence of A with B (or vice versa, of B with A, for `rewrite <-`). If I rewrite based on a proven equivalence $G \iff G'$ then I can replace goal G with G'. If I rewrite function f and by definition of f I know that $(fe) = g$, then similar to the intuition/justification of `unfold/dunfold`.

5. `cases` x:

   - if x is an element of a certain data type such as `bool` or `nat`, splits a proof/goal into several subproofs/subgoals depending on the type of x;

[10] The phrase "replaced by g" is a bit simplistic, as the rules of substitution are not as trivial as they might seem at first glance. Luckily, we don't have to worry about defining precisely what the rules of substitution are, going over all its subtleties (free vs. bound variables, etc), in this course. The reason is that LEAN is watching over us and performs substitutions correctly on our behalf.

Intuition/justification: If I have to prove P assuming that x is of some inductive data type T, then it suffices to prove P for each of the possible objects that x could be, based on the constructors of T.

- if x is a hypothesis of the form $P \vee Q$, splits a proof/goal into two subproofs/subgoals, one where P is assumed, and another where Q is assumed;

  Intuition/justification: If I have to prove G assuming that $P \vee Q$ holds, then it suffices to prove G in each of the two cases: Case (1): P holds, and Case (2): Q holds.
- if x is a hypothesis of the form $P \wedge Q$, replaces x with two hypotheses, one stating that P holds, the other stating that Q holds.

  Intuition/justification: If I have to prove G assuming that $P \wedge Q$ holds, then it suffices to prove G assuming that both P holds and Q holds.

If we add `with ...` at the end, then we can rename the variables or labels in the various cases. Otherwise, LEAN picks the names for us.

6. `assumption`: discharges the goal when one of the hypotheses is identical to the goal.

   Intuition/justification: $G \to G$ trivially holds for any G.
7. `exact` H: discharges the goal when hypothesis H is identical to the goal.

   Intuition/justification: $G \to G$ trivially holds for any G.
8. `trivial`: discharges the goal when either the goal is `true` (or "obviously true"), or one of the hypotheses is `false` (or "obviously false").

   Intuition/justification: $A \to$ `true` trivially holds for any A, and `false` $\to G$ trivially holds for any G.
9. `repeat { ... }` : repeats the sequence of tactics within `{ ... }` as many times as it can.
10. `left`: when the goal is $P \vee Q$, transforms the goal into P.

    Intuition/justification: to prove $P \vee Q$ it suffices to prove P.
11. `right`: when the goal is $P \vee Q$, transforms the goal into Q.

    Intuition/justification: to prove $P \vee Q$ it suffices to prove Q.
12. `split`: when the goal is $P \wedge Q$, splits the proof/goal into two subproofs/subgoals, one for P and one for Q.

    Intuition/justification: to prove $P \wedge Q$ it suffices to prove P and Q separately.
13. `have H : P := ...`: creates the new hypothesis H that P holds. We must then prove P, by filling in the `...` with a proof.

    Intuition/justification: to prove G from hypotheses $H_1, H_2, ...$, it suffices to (1) prove a new goal P from hypotheses $H_1, H_2, ...$, and then (2) prove G using the existing hypotheses $H_1, H_2, ...$ plus the newly proved result H that P holds.
14. `induction x`: if `x` is an element of a certain inductive data type `T`, perform induction on `x`. Generates several proof obligations and the corresponding induction hypotheses (if any) depending on the constructors of `T`.

    Intuition/justification: If I have to prove P assuming that `x:T`, then it suffices to prove P for each of the possible objects that `x` could be, based on the constructors of `T`. In doing so, I can assume that P holds for all previously/already constructed objects of type `T` (induction hypotheses), in order to prove P for a newly constructed object of type `T` (induction step).
15. `revert x`: if `x:T` is a variable of some data type `T` like `nat`, `bool`, etc, puts `x` back into the goal as `∀x...` ; if `x:P` is a hypothesis that some proposition `P` holds, puts `P` back into the goal as `P -> ...`.
16. `clear h`: removes (no longer needed) hypothesis `h` from the proof state (to reduce clutter).

9 Allowed LEAN Library Axioms/Theorems

In your proofs, you are allowed to appeal to the following from the LEAN library:[11]

```
#check band_self
#check bor_self
#check band_ff
#check band_tt
#check bor_ff
#check bor_tt
#check ff_band
#check ff_bor
#check tt_band
#check tt_bor

#check band_eq_false_eq_eq_ff_or_eq_ff
#check band_eq_true_eq_eq_tt_and_eq_tt
#check bor_eq_false_eq_eq_ff_and_eq_ff
#check bor_eq_true_eq_eq_tt_or_eq_tt

#check and_self
#check or_self
#check and_comm
#check or_comm
#check or_false
#check false_or
#check or_true
#check true_or
#check and_true
#check true_and
#check and_false
#check false_and
```

In addition to the above results from the LEAN library, you are also allowed to use *any* result previously proven in class, including in lectures, labs, homeworks, etc. For instance, you are allowed to use anything in given `ourlibraryXYZ.lean` and all lecture and homework files uploaded on canvas. You are also allowed to copy your own solutions from past homeworks, define your own helper functions, define and prove your own theorems and lemmas, etc.

10 Specifications for software transparency and ethics

These days there is a lot of talk about computer science ethics: bias and fairness in AI and other systems, and the like. Does this course have anything to contribute to that debate? I believe so. This course talks about software *specification*, and specification is a description of *what* the software is supposed to do, and not *how* exactly it does it. Specification can be the basis for software *transparency*. Without devulging their intellectual property secrets (the *how*), companies can still reveal the *what*: what properties does their software have? what is their software actually supposed to do? Then users of the software can make initial judgements about the ethics of the software based on its specification.

[11]If there is a result missing from the list that you think is reasonably basic and should be included, please let Stavros know.

References

[1] C. Baier and J.-P. Katoen. *Principles of Model Checking*. MIT Press, 2008.

[2] Mordechai Ben-Ari. *Mathematical Logic for Computer Science, 3rd Edition*. Springer, 2012.

[3] Yves Bertot and Pierre Casteran. *Interactive Theorem Proving and Program Development – Coq'Art: The Calculus of Inductive Constructions*. Springer, 2004.

[4] A. R. Bradley and Z. Manna. *The calculus of computation - decision procedures with applications to verification*. Springer, 2007.

[5] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 2000.

[6] Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, editors. *Handbook of Model Checking*. Springer, 2018.

[7] A. Doxiadis, C.H. Papadimitriou, A. Papadatos, and A. Di Donna. *Logicomix: An Epic Search for Truth*. Bloomsbury USA, 2009.

[8] Joseph Y. Halpern, Robert Harper, Neil Immerman, Phokion G. Kolaitis, Moshe Y. Vardi, and Victor Vianu. On the unusual effectiveness of logic in computer science. *The Bulletin of Symbolic Logic*, 7(2):213–236, 2001.

[9] John Harrison. *Handbook of Practical Logic and Automated Reasoning*. Cambridge University Press, 2009.

[10] G. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, 1991.

[11] G. Holzmann. *The Spin Model Checker*. Addison-Wesley, 2003.

[12] M. Huth and M. Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems*. Cambridge University Press, 2004.

[13] Matt Kaufmann, Panagiotis Manolios, and J Strother Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, 2000.

[14] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, New York, 1991.

[15] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety*. Springer-Verlag, New York, 1995.

[16] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL – A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.