CS 2800 Logic and Computation
Lecture Notes, Fall 2020

Stavros Tripakis

December 5, 2020

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 place Google's software at about 2 billion lines of code, and Microsoft's Windows operating system at about 50 million lines of code[1]; a pacemaker has about 100 thousand lines of code, the Boeing 787 airplane has more than 10 million, and a modern high-end car has 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 in mathematics.[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 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 key word 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? 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.

3 This Course

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 most programming courses you will focus on checking program correctness by *testing*. Testing is very important, but as Dijkstra 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 do 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 the meaning of love and politics, 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 love and politics, but also in engineering projects.

Specification and verification: In this course we will use logic for several things.[5] 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: This semester, 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. We found most helpful the instructions provided here: `https://leanprover-community.github.io/`.

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. WARNING: this game can become addictive!

[5] 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.

How to succeed in this course: You 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. 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. Etc. Try to experiment to see what goes wrong. For LEAN specific things, consult the LEAN documentation. Ask questions when you are deadlocked. *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. Their aim is not to be a comprehensive textbook, but rather to guide you in the course (like a map). The philosophy of the course is *learning by doing.* Is there any other way to learn really?

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. References to those will be provided as the course progresses.

These lecture notes will be permanently under construction. They will be updated regularly as we advance in the course. The latest version 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/

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.[6] 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

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.

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

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/2020/. 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.

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 [9].
- *Handbook of Practical Logic and Automated Reasoning*, by Harrison [6].
- *The Calculus of Computation - Decision Procedures with Applications to Verification*, by Bradley and Manna [2].

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

- *Model Checking*, by Clarke, Grumberg and Peled [3].
- *Principles of Model Checking*, by Baier and Katoen [1].
- Several books on the *SPIN Model Checker* by Holzmann [7, 8].
- 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) [10, 11].
- *Handbook of Model Checking*, by Clarke, Henzinger, Veith, Bloem [4].

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 [5].

6 Course Outline

6.1 Introduction (Lecture 1)

- Course goals and logistics.
- LEAN and other theorem provers (Coq, Isabelle, ACL2s).

6.2 Functional Programming with Types in LEAN (Lectures 2 - 6)

Programming in a functional language with types.

- 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`, and a word about termination.
- Anonymous functions (lambda abstraction).
- Booleans and functions on booleans.
- Product types and currying.

6.3 Testing as Proving (Lectures 6 - 7)

Writing 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.

6.4 Introduction to Specifications (Lecture 7)

- The type `Prop`.
- Properties and specifications.
- Informal and formal specifications.
- `example`, `lemma`, `theorem`.
- `sorry`.

6.5 Defining new types (Lectures 7 - 8)

- Defining our own types.
- Constructors.
- Enumerative types: the type `weekday`.
- Inductive data types.
- Defining the natural numbers: the type `mynat`.
- Defining recursive functions on inductive data types by pattern matching (*data-driven definitions*).
- Trees.
- Helper functions.

6.6 For-All Specifications (Lecture 9)

- Writing specifications with `forall` (∀).
- Formal specification and verification.
- Diving more into proofs.
- Proof tactics: `intro`.
- `try`.

6.7 Equational Reasoning and Introduction to Logic (Lectures 10 - 11)

- Equational reasoning.
- More forall specifications.
- Proof by cases.
- Introduction to logic: conjunction, disjunction, negation, implication.
- Higher-order logic.
- Proof tactics: `intro` (again, for implication), `intros`, `unfold`, `cases`.

6.8 Logic (Lectures 12 - 18)

- Review of propositional logic: syntax, semantics, boolean functions, satisfiability, validity, truth tables, ...
- Proving propositional logic tautologies in LEAN vs. by truth table.
- Negation.
- If-and-only-iff (iff).
- Exclusive-OR (xor).
- Propositions as types, theorems as functions.
- Modus ponens.

- Using lemmas and theorems.
- Constructive vs. classic logic.
- The axioms `classical.em` (law of excluded middle) and `classical.by_contradiction`.
- Proof tactics: `trivial`, `assumption`, `exact`, `left`, `right`, `cases` (again, for conjunctive and disjunctive hypotheses), `split`, `repeat`, `have`, `rewrite`, `simp`.

6.9 Exam 1 – Lecture 19

Taken online using Gradescope. Material: everything covered so far up to and including `have` (you won't need `rewrite` nor `simp` but OK to use them).

6.10 Induction and Functional Induction (Lectures 20 - 22, 24 - 28)

- Proofs by induction. Base case. Induction step. Induction hypothesis.
- Proof by induction vs proof by cases.
- Multiple base cases.
- Multiple induction steps.
- Multiple induction hypotheses.
- Induction on `nat`s, lists, trees, and other inductive data types.
- Effect of induction on hypotheses.
- Discovering, writing, and using lemmas.
- The power of generalization (see `30-generalization.lean`).
- Delaying introductions.
- Functional induction.
- Induction schemes generated by recursive functions.
- Dealing with tail-recursive functions.
- Notation: `local notation`.
- "Libraries": `import`.
- Proof tactics: `induction`.

6.11 Election Special (Lecture 23)

```
inductive freedom : Type
  | basic : right -> freedom
  | other : right -> freedom -> freedom
```

6.12 Termination (Lectures 29 - 34)

- Proving termination.
- Measure functions.
- Proving theorems automatically is hard.
- Checking termination automatically is hard.
- Alan Turing.
- Undecidability.
- Why program termination is important.
- Convincing LEAN that functions (and theorems) terminate.

6.13 Introduction to Proof Automation: SAT and SMT Solvers

SAT/SMT solving and their applications.

6.14 Reasoning About Imperative Programs

- Invariants.
- Inductive invariants.
- The invariant game.

6.15 Motivation: why are we doing all this?

- What is the science of software and why we need it.
- How this course fits into the science of software.
- Going further:
  - Courses:
    - *Formal Specification, Verification, and Synthesis*, CS 7430/4830, see:
      http://www.ccs.neu.edu/~stavros/fsvs20.html
      http://www.ccs.neu.edu/~stavros/ssvs19.html
  - Books: see section **References**.

7 Summary of Proof Tactics

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 $A = A$.

   Intuition: $A = A$ is an axiom of logic, called *reflexivity of equality*.

2. `intro`: eliminates $\forall$-quantified variable introducing it into the hypotheses; turns goal $P \to Q$ into Q introducing P in the hypotheses.

   Intuition: 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.

3. `intros`: repeatedly applies `intro`.

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

   Intuition: 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.[7]

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;

     Intuition: If I have to prove P assuming that x is of some inductive data type T, then it suffices to prove P in each of the possible cases 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: 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: 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. `trivial`: discharges the goal when either the goal is `true` (or "obviously true"), or one of the hypotheses is `false` (or "obviously false").

   Intuition: $H \to \mathsf{true}$ trivially holds for any H, and $\mathsf{false} \to G$ trivially holds for any G.

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

   Intuition: $G \to G$ trivially holds for any G.

8. `exact` H: discharges the goal when hypothesis H is identical to the goal.

   Intuition: $G \to G$ trivially holds for any G.

[7] 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.

9. `left`: when the goal is $P \vee Q$, transforms the goal into P.

   Intuition: to prove $P \vee Q$ it suffices to prove P.

10. `right`: when the goal is $P \vee Q$, transforms the goal into Q.

    Intuition: to prove $P \vee Q$ it suffices to prove Q.

11. `split`: when the goal is $P \wedge Q$, splits a proof/goal into two subproofs/subgoals, one for P and one for Q.

    Intuition: to prove $P \wedge Q$ it suffices to prove P and also prove Q.

12. `repeat { ... }` : repeats the sequence of tactics within `{ ... }` as many times as it can.

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: 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 new hypothesis that P holds.

14. `rewrite [<-] H`: rewrites the goal based on the equality or equivalence H. By default rewrites from left to write. If `<-` is added, rewrites from right to left. Abbreviated `rw`. If we add `at` A at the end, then the tactic applies to hypothesis A, instead of the goal.

    Intuition: if I know $A = B$, then in order to prove 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 <-`).

15. `simp [H]`: simplifies the goal according to H. H is optional: you can just issue `simp`.

    Similar to `unfold` and `rewrite`, but can simplify more. For instance, `simp` can simplify if-then-else statements, e.g., it simplifies `ite (tt = tt) A B` to `A`.

    H could be a function, a hypothesis, or a previously proven lemma/theorem. We can also add multiple simplification rules, like: `simp [H1,H2,...]`.

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

    Intuition: similar to those for `unfold` and `rewrite`. But `simp` is particularly useful in simplifying `if-then-else` expressions!

16. `induction x`: perform induction on `x`. Different sub-goals and induction hypotheses are generated depending on the type of `x`.

    Intuition: see lecture code `20-code.lean`.

8 Allowed LEAN Library Axioms/Theorems

In your proofs, you are allowed to appeal to the following from the LEAN library:[8]

```
#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


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

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 `ourlibrary24.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.

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

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

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

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

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

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

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

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

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

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