Logic and Mechanized Reasoning

Release 0.1

Jeremy Avigad
Marijn J. H. Heule
Wojciech Nawrocki

Nov 24, 2021

CONTENTS:

CHAPTER

ONE

INTRODUCTION

1.1 Historical background

In the thirteenth century, Ramon Lull, an eccentric Franciscan monk on the Island of Majorca, wrote a work called the *Ars Magna*, which contains mechanical devices designed to aid reasoning. For example, Lull claimed that the reason that infidels did not accept the Christian god was that they failed to appreciate the multiplicity of God's attributes, so he designed nested paper circles with letters and words signifying these attributes. By rotating the circles, one obtained various combinations of the words and symbols, signifying compound attributes. For example, in one configuration, one might read off the result that God's greatness is true and good, and that his power is wise and great. Other devices and diagrams would assist in reasoning about virtues and vices, and so on.

Today, this sounds silly, but the work was based on three fundamental ideas that are still of central importance.

- First, we can use symbols, or tokens, to stand for ideas or concepts.
- Second, compound ideas and concepts are formed by putting together simpler ones.
- And, third, mechanical devices—even as simple a concentric rotating wheels—can serve as aids to reasoning.

The first two ideas go back to ancient times and can be found in the work of Aristotle. The third, however, is usually attributed to Lull, marking his work as the origin of mechanized reasoning.

Four centuries later, Lull's work resonated with Gottfied Leibniz, who invented calculus around the same time that Isaac Newton did so independently. Leibniz was also impressed by the possibility of symbolic representations of concepts and rules for reasoning. He spoke of developing a *characteristica universalis*—a universal language of thought—and a *calculus ratiocinator*—a calculus for reasoning. In a famous passage, he wrote:

> If controversies were to arise, there would be no more need of disputation between two philosophers than between two calculators. For it would suffice for them to take their pencils in their hands and to sit down at the abacus, and say to each other (and if they so wish also to a friend called to help): Let us calculate.

The last phrase—*calculemus!* in the original Latin—has become a motto of computer scientists and computationally-minded mathematicians today.

The development of modern logic in the late eighteenth and early nineteen centuries began to bring Leibniz' vision to fruition. In 1931, Kurt Gödel wrote:

> The development of mathematics towards greater precision has led, as is well known, to the formalization of large tracts of it, so that one can prove any theorem using nothing but a few mechanical rules.

It is notable that the use of the word "mechanical" here—*mechanischen* in the original German—predates the modern computer by a decade or so.

What logicians from the time of Aristotle to the present day have in common is that they are all at least slightly crazy. They are driven by the view that knowledge is rooted in language and that the key to knowledge lies in having just the right symbolic representations of language and rules of use. But often it's the crazy people that change the world. The logical view of language and knowledge lies at the heart of computer science and provides the foundation for some of our most valuable technologies today, including programming languages, automated reasoning and AI, and databases.

That's what this course is about: the logician's view of the world, the power of symbolic representations of language, and the way those representations facilitate the mechanization of reasoning and the acquisition of knowledge.

The logicians' view complements the view from statistics and machine learning, where representations of knowledge tend to be very large, approximate, and hard to represent is succinct symbolic terms. Such methods have had stunning successes in recent years, but there are still branches of computer science and AI where symbolic methods are paramount. It is an important open question as to the best way to combine logical, statistical, and machine learning methods in the years to come.

1.2 An overview of this course

This course is designed to teach you the mathematical theory behind symbolic logic, with an eye towards putting it to good use. An interesting aspect of the course is that it develops three interacting strands in parallel:

- *Theory.* We will teach you the syntax and semantics of propositional and first-order logic. If time allows, we will give you a brief overview of related topics, like simple type theory and higher-order logic.
- *Implementation.* We will teach you how to implement logical syntax—terms and formulas—in a functional programming language called *Lean*. We will also teach you how to carry out fundamental operations and transformations on these objects.
- *Application.* We will show you how to use logic-based automated reasoning tools to solve interesting and difficult problems. In particular, we will use a SAT solver called CaDiCaL, an SMT solver called Z3, and a first-order theorem prover called Vampire (and by then you will understand what all these terms mean).

The first strand will be an instance of pure mathematics. We will build on the skills you have learned in Mathematical Foundations of Computer Science (15-151). The goal is to teach you to think about and talk about logic in a mathematically rigorous way.

The second strand will give you an opportunity to code up some of what you have learned and put it to good use. Our goal is to provide a foundation for you to use logic-based computational methods in the future, whether you choose to make use of them in small or large ways. In the third strand, for illustrative purposes, we will focus mainly on solving puzzles and combinatorial problems. This will give you a sense of how the tools can also be used on proof and constraint satisfaction problems that come up in fields like program verification, discrete optimization, and AI.

1.3 Acknowledgments

We are grateful to Seulkee Baek for implementing the link between Lean and a SAT solver.

CHAPTER

TWO

MATHEMATICAL BACKGROUND

2.1 Induction and recursion on the natural numbers

In its most basic form, the principle of induction on the natural numbers says that if you want to prove that every natural number has some property, it suffices to show that zero has the property, and that whenever some number n has the property, so does $n + 1$. Here is an example.

Theorem

For every natural number n, $\sum_{i \leq n} i = n(n+1)/2$.

Proof

Use induction on n. In the base case, we have $\sum_{i \leq 0} i = 0 = 0(0+1)/2$. For the induction step, assuming $\sum_{i \leq n} i = n(n+1)/2$, we have

$$\begin{aligned} \sum_{i \leq n+1} i &= \sum_{i \leq n} i + (n+1) \\ &= n(n+1)/2 + 2(n+1)/2 \\ &= (n+1)(n+2)/2 \end{aligned}$$

The story is often told that Gauss, as a schoolchild, discovered this formula by writing

$$\begin{aligned} S &= 1 + \ldots + n \\ S &= n + \ldots + 1 \end{aligned}$$

and then adding the two rows and dividing by two. The proof by induction doesn't provide insight as to how one might *discover* the theorem, but once you have guessed it, it provides a short and effective means for establishing that it is true.

In a similar vein, you might notice that an initial segment of the odd numbers yields a perfect square. For example, we have $1 + 3 + 5 + 7 + 9 = 25$. Here is a proof of the general fact:

Theorem

For every natural number n, $\sum_{i \leq n} (2i + 1) = (n+1)^2$.

Proof

The base case is easy, and assuming the inductive hypothesis, we have

$$\begin{aligned}\sum_{i \leq n+1} (2i+1) &= \sum_{i \leq n} (2i+1) + 2(n+1) + 1 \\ &= (n+1)^2 + 2n + 3 \\ &= n^2 + 4n + 4 \\ &= (n+2)^2.\end{aligned}$$

A close companion to induction is the principle of *recursion*. Recursion enables us to define functions on the natural numbers, and induction allows us to prove things about them. For example, let $g : \mathbb{N} \to \mathbb{N}$ be the function defined by

$$\begin{aligned}g(0) &= 1 \\ g(n+1) &= (n+1) \cdot g(n)\end{aligned}$$

Then g is what is known as the *factorial* function, whereby $g(n)$ is conventionally written $n!$. The point is that if you don't know what the factorial function is, the two equations above provide a complete specification. There is exactly one function, defined on the natural numbers, that meets that description.

Here is an identity involving the factorial function:

Theorem

$\sum_{i \leq n} i \cdot i! = (n+1)! - 1$.

Proof

The base case is easy. Assuming the claim holds for n, we have

$$\begin{aligned}\sum_{i \leq n+1} i \cdot i! &= \sum_{i \leq n} i \cdot i! + (n+1) \cdot (n+1)! \\ &= (n+1)! + (n+1) \cdot (n+1)! - 1 \\ &= (n+1)!(1 + (n+1)) - 1 \\ &= (n+2)! - 1\end{aligned}$$

This is a pattern found throughout mathematics and computer science: define functions and operations using recursion, and then use induction to prove things about them.

The *Towers of Hanoi* puzzle provides a textbook example of a problem that can be solved recursively. The puzzle consists of three pegs and disks of different diameters that slide onto the pegs. The initial configuration has n disks stacked on one of the pegs in decreasing order, with the largest one at the bottom and the smallest one at the top. Suppose the pegs are numbered 1, 2, and 3, with the disks starting on peg 1. The required task is to move all the disks from peg 1 to peg 2, one at a time, with the constraint that a larger disk is never placed on top of a smaller one.

```
To move n disks from peg A to peg B with auxiliary peg C:
    if n = 0
        return
    else
        move n - 1 disks from peg A to peg C using auxiliary peg B
        move 1 disk from peg A to peg B
        move n - 1 disks from peg C to peg B using auxiliary peg A
```

We will show in class that this requires $2^n - 1$ moves. The exercises below ask you to show that *any* solution requires at least this many moves.

2.2 Complete induction

As we have described it, the principle of induction is pretty rigid: in the inductive step, to show that $n + 1$ has some property, we can only use the corresponding property of n. The principle of *complete* induction is much more flexible.

Principle of complete induction

To show that every natural number n has some property, show that n has that property whenever all smaller numbers do.

As an exercise, we ask you to prove the principle of complete induction using the ordinary principle of induction. Remember that a natural number greater than or equal to 2 is *composite* if it can be written as a product of two smaller numbers, and *prime* otherwise.

Theorem

Every number greater than two can be factored into primes.

Proof

Let n be any natural number greater than or equal to 2. If n is prime, we are done. Otherwise, write $n = m \cdot k$, where m and k are smaller than n (and hence greater than 1). By the inductive hypothesis, m and k can each be factored into prime numbers, and combining these yields a factorization of n.

Here is another example we will discuss in class:

Theorem

For any $n \geq 3$, the sum of the angles in any n-gon is $180(n - 2)$.

The companion to complete induction on the natural numbers is a form of recursion known as course-of-values recursion, which allows you to define a function f by giving the value of $f(n)$ in terms of the value of f at arbitrary smaller values of n. For example, we can define the sequence of Fibonacci numbers as follows:

$$\begin{aligned} F_0 &= 0 \\ F_1 &= 1 \\ F_{n+2} &= F_{n+1} + F_n \end{aligned}$$

The fibonacci numbers satisfy lots of interesting identities, some of which are given in the exercises.

In fact, you can define a function by recursion as long as *some* associated measure decreases with each recursive call. Define a function $f(n, k)$ for $k \leq n$ by

$$f(n,k) = \begin{cases} 1 & \text{if } k = 0 \text{ or } k = n \\ f(n-1,k) + f(n-1,k-1) & \text{otherwise} \end{cases}$$

Here it is the first argument that decreases. In class, we'll discuss a proof that this defines the function

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

which is simultaneously equal to number of ways of choosing k objects out of n without repetition.

Finally, here is a recursive description of the greatest common divisor of two nonnegative integers:

$$\mathit{gcd}(x, y) = \begin{cases} x & \text{if } y = 0 \\ \mathit{gcd}(y, \mathit{mod}(x, y)) & \text{otherwise} \end{cases}$$

where $\mathit{mod}(x, y)$ is the remainder when dividing x by y.

2.3 Generalized induction and recursion

The natural numbers are characterized inductively by the following clauses:

- 0 is a natural number.
- If x is a natural number, so is $\mathit{succ}(x)$.

Here the function $\mathit{succ}(x)$ is known as the *successor* function, namely, the function that, given any number, returns the next one in the sequence. The natural numbers structure is also sometimes said to be *freely generated* by this data. The fact that it is *generated* by 0 and $\mathit{succ}(x)$ means that it is the *smallest* set that contains 0 and is closed under $\mathit{succ}(x)$; in other words, any set of natural numbers that contains 0 and is closed under $\mathit{succ}(x)$ contains all of them. This is just the principle of induction in disguise. The fact that it is generated *freely* by these elements means that there is no confusion between them: 0 is not a successor, and if $\mathit{succ}(x) = \mathit{succ}(y)$, then $x = y$. Intuitively, being generated by 0 and $\mathit{succ}(x)$ means that any number can be represented by an expression built up from these, and being generated freely means that the representation is unique.

The natural numbers are an example of an *inductively defined structure*. These come up often in logic and computer science. It is often useful to define functions by recursion on such structures, and to use induction to prove things about them. We will describe the general schema here with some examples that often come up in computer science.

Let α be any data type. The set of all *lists* of elements of α, which we will write as $\mathit{List}(\alpha)$, is defined inductively as follows:

- The element *nil* is an element of $\mathit{List}(\alpha)$.
- If a is an element of α and ℓ is an element of $\mathit{List}(\alpha)$, then the element $\mathit{cons}(a, \ell)$ is an element of $\mathit{List}(\alpha)$.

Here *nil* is intended to describe the empty list, $[]$, and $\mathit{cons}(a, \ell)$ is intended to describe the result of adding a to the beginning of ℓ. So, for example, the list of natural numbers $[1, 2, 3]$ would be written $\mathit{cons}(1, \mathit{cons}(2, \mathit{cons}(3, \mathit{nil})))$. Think of $\mathit{List}(\alpha)$ as having a constructor $\mathit{cons}(a, \cdot)$ for each a. Then, in the terminology above, $\mathit{List}(\alpha)$ is generated inductively by *nil* and those constructors.

Henceforth, for clarity, we'll use the notation $[]$ for *nil* and $a :: \ell$ for $\mathit{cons}(a, \ell)$. More generally, we can take $[a, b, c, \ldots]$ to be an abbreviation for $a :: (b :: (c :: \ldots []))$.

Saying that $\mathit{List}(\alpha)$ is inductively defined means that we principles of recursion and induction on it. For example, the following concatenates two lists:

$$\begin{aligned} \mathit{append}([], m) &= m \\ \mathit{append}((a :: \ell), m) &= a :: (\mathit{append}(\ell, m)) \end{aligned}$$

Here the recursion is on the first argument. As with the natural numbers, the recursive definition specifies what to do for each of the constructors. We'll use the notation $\ell +\!\!+ m$ for $\mathit{append}(\ell, m)$, and with this notation, the two defining clauses read as follows:

$$\begin{aligned} [] +\!\!+ m &= m \\ (a :: \ell) +\!\!+ m &= a :: (\ell +\!\!+ m) \end{aligned}$$

From the definition, we have $[] +\!\!+ \ell = \ell$ for every ℓ, but $m +\!\!+ [] = m$ is something we have to prove.

Proposition

For every m, we have $m +\!\!+ [] = m$.

Proof

We use induction on m. In the base case, we have $[] +\!\!+ [] = []$ from the definition of *append*. For the induction step, suppose we have $m +\!\!+ [] = m$. Then we also have

$$\begin{aligned}(a :: m) +\!\!+ [] &= a :: (m +\!\!+ []) \\ &= a :: m.\end{aligned}$$

The definition of the append function is an example of *structural recursion*, called that because the definition proceeds by recursion on the structure of the inductively defined type. In particular, there is a clause of the definition corresponding to each constructor. The proof we have just seen is an instance of *structural induction*, called that because, once again, there is part of the proof for each constructor. The base case, for *nil*, is straightforward, because that constructor has no arguments. The inductive step, for *cons*, comes with an inductive hypothesis because the *cons* constructor has a recursive argument. In class, we'll do a similar proof that the *append* operation is associative.

The following function (sometimes called *snoc*) appends a single element at the end:

$$\begin{aligned}\mathit{append1}([], a) &= \mathit{cons}(a, \mathit{nil}) \\ \mathit{append1}(\mathit{cons}(b, \ell), a) &= \mathit{cons}(b, \mathit{append1}(\ell, a))\end{aligned}$$

An easy induction on ℓ shows that, as you would expect, $\mathit{append1}(\ell, a)$ is equal to $\ell +\!\!+ [a]$.

The following function reverses a list:

$$\begin{aligned}\mathit{reverse}([]) &= [] \\ \mathit{reverse}(\mathit{cons}(a, \ell)) &= \mathit{append1}(\mathit{reverse}(\ell), a)\end{aligned}$$

In class, or for homework, we'll work through proofs that that the following holds for every pair of lists ℓ and m:

$$\mathit{reverse}(\ell +\!\!+ m) = \mathit{reverse}(m) +\!\!+ \mathit{reverse}(\ell)$$

Here is another example of a property that can be proved by induction:

$$\mathit{reverse}(\mathit{reverse}(\ell)) = \ell$$

From a mathematical point of view, this definition of the reverse function above is as good as any other, since it specifies the function we want unambiguously. But in Chapter 3 we will see that such a definition can also be interpreted as executable code in a functional programming language such as Lean. In this case, the execution is quadratic in the length of the list (think about why). The following definition is more efficient in that sense:

$$\begin{aligned}\mathit{reverseAux}([], m) &= m \\ \mathit{reverseAux}((a :: \ell), m) &= \mathit{reverseAux}(\ell, (a :: m))\end{aligned}$$

$$\mathit{reverse}'(\ell) = \mathit{reverseAux}(\ell, [])$$

The idea is that *reverseAux* adds all the elements of the first argument to the second one in reverse order. So the second arguments acts as an *accumulator*. In fact, because it is a tail recursive description, the code generated by Lean is quite efficient. In class, we'll discuss an inductive proof that $\mathit{reverse}(\ell) = \mathit{reverse}'(\ell)$ for every ℓ.

It is worth mentioning that structural induction is not the only way to prove things about lists, and structural recursion is not the only way to define functions by recursion. Generally speaking, we can assign any complexity measure to a data type, and do induction on complexity, as long as the measure is well founded. (This will be the case, for example, for measures that take values in the natural numbers, with the usual ordering on size.) For example, we can define the length of a list as follows:

$$\begin{aligned} \mathit{length}([]) &= 0 \\ \mathit{length}(a :: \ell) &= \mathit{length}(\ell) + 1 \end{aligned}$$

Then we can define a function f on lists by giving the value of $f(\ell)$ in terms of the value of f on smaller lists, and we can prove a property of lists using the fact that the property holds of all smaller lists as an inductive hypothesis. These are ordinary instances of recursion and induction on the natural numbers.

As another example, we consider the type of finite binary trees, defined inductively as follows:

- The element *empty* is a binary tree.
- If s and t are finite binary trees, so is the $\mathit{node}(s, t)$.

In this definition, *empty* is intended to denote the empty tree, and $\mathit{node}(s, t)$ is intended to denote the binary tree that consists of a node at the top and has s and t as the left and right subtrees, respectively.

Be careful: it is more common to take the set of binary trees to consist of only the *nonempty* trees, in which case, what we have defined here are called the *extended* binary trees. Adding the empty tree results in a nice inductive characterization. If we started with a one-node tree as the base case, we would have to allow for three types of compound tree: one type with a node and a subtree to the left, one with a node and a subtree to the right, and one with a node with both left and right subtrees.

We can count the number of nodes in an extended binary tree with the following recursive definition:

$$\begin{aligned} \mathit{size}(\mathit{empty}) &= 0 \\ \mathit{size}(\mathit{node}(s, t)) &= 1 + \mathit{size}(s) + \mathit{size}(t) \end{aligned}$$

We can compute the depth of an extended binary tree as follows:

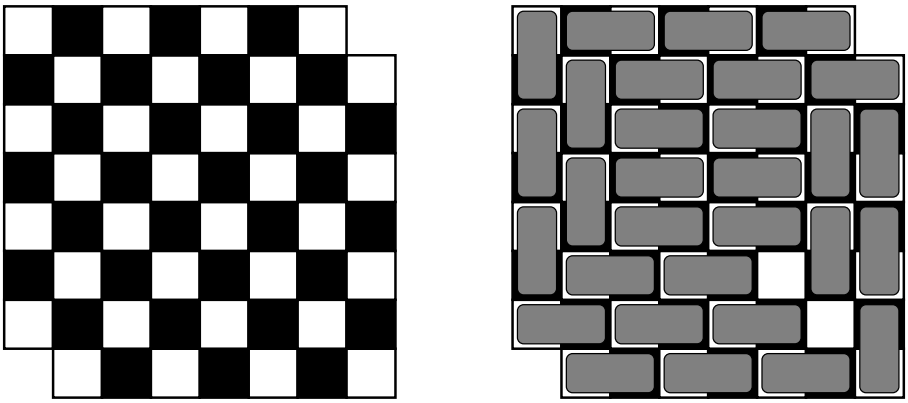
$$\begin{aligned} \mathit{depth}(\mathit{empty}) &= 0 \\ \mathit{depth}(\mathit{node}(s, t)) &= 1 + \max(\mathit{depth}(s), \mathit{depth}(t)) \end{aligned}$$

Again, be careful: many authors take the depth of a tree to be the length of the longest path from the root to a leaf, in which case, what we have defined here computes the depth *plus one* for nonempty trees.

2.4 Invariants

The *mutilated chessboard* problems involves an 8×8 chessboard with the top right and bottom left corners removed. Imagine you are given a set of dominoes, each of which can cover exactly two squares. It is possible to cover all the squares of the mutilated chessboard using dominoes, so that each square is covered by exactly one domino?

A moment's reflection shows that the answer is no. If you imagine the chessboard squares colored white and black in the usual way, you'll notice that the two squares we removed have the same color, say, black. That means that there are more white squares than black squares. On the other hand, every domino covers exactly one square of each color. So no matter how many dominoes we put down, we'll never have them color more white squares than black squares.



The fact that any way of putting down dominoes covers the same number of white and black squares is an instance of an *invariant*, which is a powerful idea in both mathematics and computer science. An invariant is something—a quantity, or a property—that doesn't change as something else does (in this case, the number of dominoes).

Often the natural way to establish an invariant uses induction. In this case, it is obvious that putting down one domino doesn't change the difference between the number of white and black squares covered, since each domino covers one of each. By induction on n, putting down n dominoes doesn't change the difference either.

The following puzzle, called the *MU puzzle*, comes from the book *Gödel, Escher, Bach* by Douglas Hofstadter. It concerns strings consisting of the letters *M*, *I*, and *U*. Starting with the string *MI*, we are allowed to apply any of the following rules:

1. Replace *sI* by *sIU*, that is, add a *U* to the end of any string that ends with *I*.
2. Replace *Ms* by *Mss*, that is, double the string after the initial *M*.
3. Replace *sIIIt* by *sUt*, that is, replace any three consecutive *Is* with a *U*.
4. Replace *sUUt* by *st*, that is, delete any consecutive pair of *Us*.

The puzzle asks whether it is possible to derive the string *MU*. The answer is no: it turns out that a string is derivable if and only if it consists of an *M* followed by any number of *Is* and *Us*, as long as the number of *Is* is not divisible by 3. In class, we'll prove the "only if" part of this equivalence. Try the "if" part if you like a challenge.

As a final example, in class we'll discuss the Golomb *tromino theorem*. A *tromino* is an L-shaped configuration of three squares. Golomb's theorem says that any $2^n \times 2^n$ chessboard with one square removed can be tiled with trominoes. We'll prove this together in class.

2.5 Exercises

1. Prove the formula for the sum of a geometric series:

$$\sum_{i<n} ar^i = \frac{a(r^n - 1)}{r - 1}$$

2. Prove that for every $n > 4$, $n! > 2^n$.
3. Show that the solution to the towers of Hanoi given in Section 2.1 is optimal: for every n, it takes at least *2^n - 1* moves to move all the disks from one peg to another.
4. Consider the variation on the towers of Hanoi problem in which you can only move a disk to an *adjacent* peg. In other words, you can move a disk from peg 1 to peg 2, from peg 2 to peg 1, from peg 2 to peg 3, or from peg 3 to peg 2, but not from peg 1 to peg 3 or from peg 3 to peg 1.

Describe a recursive procedure for solving this problem, and show that is requires $3^n - 1$ moves. If you are ambitious, show that this is optimal, and that it goes through all the 3^n valid positions.

5. Consider the variation on the towers of Hanoi in which pegs can be moved cyclicly: from peg 1 to peg 2, from peg 2 to peg 3, or from peg 3 to peg 1. Describe a recursive procedure to solve this problem.
6. Use the ordinary principle of induction to prove the principle of complete induction.
7. Let $F_0, F_1, F_2, \ldots$ be the sequence of Fibonacci numbers.
   1. Let α and β be the two roots of the equation $x^2 = x + 1$. Show that for every n, $F_n = (\alpha^n - \beta^n)/\sqrt{5}$.
   2. Show $\sum_{i<n} F_i = F_{n+1} - 1$.
   3. Show $\sum_{i\leq n} F_i^2 = F_n F_{n+1}$.
8. Show that with *n* straight lines we can divide the plane into at most $n^2 + n + 2$ regions, and that this is sharp.
9. Show that the recursive description of $\mathit{gcd}(x, y)$ presented in Section 2.3 correctly computes the greatest common divisor of x and y, where we define $\mathit{gcd}(0, 0)$ equal to 0. You can restrict attention to nonnegative values of x and y. (Hint: you can use the fact that for for every y not equal to 0, we can write $x = \mathit{div}(x, y) \cdot y + \mathit{mod}(x, y)$, where $\mathit{div}(x, y)$ is the integer part of x divided by y. First show that for every k, $\mathit{gcd}(x, y) = \mathit{gcd}(x + ky, y)$, and use that fact.
10. Use structural induction to prove

$$\mathit{reverse}(\ell +\!\!+ m) = \mathit{reverse}(m) +\!\!+ \mathit{reverse}(\ell).$$

11. Use structural induction to prove

$$\mathit{reverse}(\mathit{reverse}(\ell)) = \ell.$$

12. Prove that for every ℓ we have

$$\mathit{reverse}'(\ell) = \mathit{reverse}(\ell).$$

13. Prove that for every ℓ and m we have

$$\mathit{length}(\ell +\!\!+ m) = \mathit{length}(\ell) + \mathit{length}(m).$$

14. How many binary trees of depth n are there? Prove your answer is correct.
15. Show that a string is derivable in the MU puzzle if and only if it consists of an M followed by any number of Is and Us, as long as the number of Is is not divisible by 3.

CHAPTER

THREE

LEAN AS A PROGRAMMING LANGUAGE

3.1 About Lean

Lean is a new programming language and interactive proof assistant being developed at Microsoft Research. It is currently in an experimental, development stage, which makes it a risky choice for this course. But in many ways it is an ideal system for working with logical syntax and putting logic to use. Lean is an exciting project, and the system fun to use. So please bear with us. Using Lean puts us out on the frontier, but if you adopt a pioneering attitude, you will be in a good position to enjoy all the cool things that Lean has to offer.

You can learn more about Lean on the Lean home page, on the Lean community home page, and by asking questions on the Lean Zulip chat, which you are heartily encouraged to join. To be more precise, there are currently two versions of Lean:

- Lean 3 is reasonably stable, and primarily an interactive proof assistant. It has a very large mathematical library, known as mathlib.
- Lean 4 is being designed as a performant programming language, and it is still under development. It can also be used as a proof assistant, though it does not yet have a substantial library. Its language and syntax are similar to that of Lean 3, but it is not backward compatible.

In this course, we will use Lean 4, even though it is still under development. It has the rough beginnings of a user manual and there is a tutorial on the underlying foundation. As we will see, Lean has a lot of features that make that worthwhile. In particular, Lean 4 is designed to be an ideal language for implementing powerful logic-based systems, as evidenced by the fact that most of Lean 4 is implemented in Lean 4 itself.

The goal of this section is to give you a better sense of what Lean is, how it can possibly be a programming language and proof assistant at the same time, and why that makes sense. The rest of the introduction will give you a quick tour of some of its features, and we will learn more about them as the course progresses.

At the core, Lean is an implementation of a formal logical foundation known as *type theory*. More specifically, it is an implementation of *dependent type theory*, and even more specifically than that, it implements a version of the *Calculus of Inductive Constructions*. Saying that it implements a formal logic foundation means that there is a precise grammar for writing expressions, and precise rules for using them. In Lean, every well-formed expression has a type.

```
#check 2 + 2
#check -5
#check [1, 2, 3]
#check #[1, 2, 3]
#check (1, 2, 3)
#check "hello world"
#check true
#check fun x => x + 1
#check fun x => if x = 1 then "yes" else "no"
```

You can find this example in the file *using_lean_as_a_programming_language/examples1.lean* in the *LAMR/Examples* folder of the course repository. We recommend copying that entire folder in the *User* folder, so you can edit the files and try examples of your own. You can always find the original file in the folder *LAMR/Examples*, which you should not edit.

If you hover over the *#check* statements or move your cursor to one of these lines and check the information window, Lean reports the result of the command. It tells you that *2 + 2* has type *Nat*, *-5* has type *Int*, and so on. In fact, in the formal foundation, types are expressions as well. The types of all the expressions above are listed below:

```
#check Nat
#check Int
#check List Nat
#check Array Nat
#check Nat × Nat × Nat
#check String
#check Bool
#check Nat → Nat
#check Nat → String
```

Now Lean tells you each of these has type *Type*, indicating that they are all data types. If you know the type of an expression, you can ask Lean to confirm it:

```
#check (2 + 2 : Nat)
#check ([1, 2, 3] : List Nat)
```

Lean will report an error if it cannot construe the expression as having the indicated type.

In Lean, you can define new objects with the *def* command. The new definition becomes part of the *environment*: the defined expression is associated with the identifier that appears after the word *def*.

```
def four : Nat := 2 + 2

def isOne (x : Nat) : String := if x = 1 then "yes" else "no"

#check four
#print four

#check isOne
#print isOne
```

The type annotations indicate the intended types of the arguments and the result, but they can be omitted when Lean can infer them from the context:

```
def four' := 2 + 2

def isOne' x := if x = 1 then "yes" else "no"
```

So far, so good: in Lean, we can define expressions and check their types. What makes Lean into a programming language is that the logical foundation has a computational semantics, under which expressions can be *evaluated*.

```
#eval four
#eval isOne 3
#eval isOne 1
```

The *#eval* command evaluates the expression and then displays the return value. Evaluation can also have *side effects*, which are generally related to system IO. For example, displaying the string "Hello, world!" is a side effect of the following evaluation:

```
#eval IO.println "Hello, world!"
```

Theoretical computer scientists are used to thinking about programs as expressions and identifying the act of running the program with the act of evaluating the expression. In Lean, this view is made manifest, and the expressions are defined in a formal system with a precise specification.

But what makes Lean into a proof assistant? To start with, some expressions in the proof system express propositions:

```
#check 2 + 2 = 4
#check 2 + 2 < 5
#check isOne 3 = "no"
#check 2 + 2 < 5 ∧ isOne 3 = "no"
```

Lean confirms that each of these is a proposition by reporting that each of them has type *Prop*. Notice that they do not all express *true* propositions; theorem proving is about certifying the ones that are. But the language of Lean is flexible enough to express just about any meaningful mathematical statement at all. For example, here is the statement of Fermat's last theorem:

```
def Fermat_statement : Prop :=
∀ a b c n : Nat, a * b * c ≠ 0 ∧ n > 2 → a^n + b^n ≠ c^n
```

In Lean's formal system, data types are expressions of type *Type*, and if T is a type, an expression of type T denotes an object of that type. We have also seen that propositions are expressions of type *Prop*. In the formal system, if P is a proposition, a proof of P is just an expression of type P. This is the final piece of the puzzle: we use Lean as a proof assistant by writing down a proposition P, writing down an expression p, and asking Lean to confirm that p has type P. The fact that $2 + 2 = 4$ has an easy proof, that we will explain later:

```
theorem two_plus_two_is_four : 2 + 2 = 4 := rfl
```

In contrast, proving Fermat's last theorem is considerably harder.

```
theorem Fermat_last_theorem : Fermat_statement := sorry
```

Lean knows that *sorry* is not a real proof, and it flags a warning there. If you manage to replace *sorry* by a real Lean expression, please let us know. We will be very impressed.

So, in Lean, one can write programs and execute them, and one can state propositions and prove them. In fact, one can state propositions about programs and then prove those statements as well. This is known as *software verification*; it is a means of obtaining a strong guarantee that a computer program behaves as intended, something that is important, say, if you are using the software to control a nuclear reactor or fly an airplane.

This course is not about software verification. We will be using Lean 4 primarily as a programming language, one in which we can easily define logical expressions and manipulate them. To a small extent, we will also write some simple proofs in Lean. This will help us think about proof systems and rules, and understand how they work. Taken together, these two activities embody the general vision that animates this course: knowing how to work with formally specified expressions and rules opens up a world of opportunity. It is the key to unlocking the secrets of the universe.

3.2 Using Lean as a functional programming language

There is a preliminary user's manual for Lean, still a work in progress, here. The fact that Lean is a functional programming language means that instead of presenting a program as a list of instructions, you simply *define* functions and ask Lean to evaluate them.

```
def foo n := 3 * n + 7

#eval foo 3
#eval foo (foo 3)

def bar n := foo (foo n) + 3

#eval bar 3
#eval bar (bar 3)
```

There is no global state: any value a function can act on is passed as an explicit argument and is never changed. For that reason, functional programming languages are amenable to parallelization.

Nonetheless, Lean can do handle system IO using the *IO monad*, and can accommodate an imperative style of programming using *do notation*.

```
def printExample : IO Unit:= do
  IO.println "hello"
  IO.println "world"

#eval printExample
```

Recursive definitions are built into Lean.

```
def factorial : Nat → Nat
  | 0       => 1
  | (n + 1) => (n + 1) * factorial n

#eval factorial 10
#eval factorial 100
```

Here is a solution to the Towers of Hanoi problem:

```
def hanoi (numPegs start finish aux : Nat) : IO Unit :=
  match numPegs with
  | 0     => pure ()
  | n + 1 => do
      hanoi n start aux finish
      IO.println s!"Move disk {n + 1} from peg {start} to peg {finish}"
      hanoi n aux finish start

#eval hanoi 7 1 2 3
```

You can also define things by recursion on lists:

```
def addNums : List Nat → Nat
  | []    => 0
  | a::as => a + addNums as

#eval addNums [0, 1, 2, 3, 4, 5, 6]
```

In fact, there are a number of useful functions built into Lean's library. The function *List.range n* returns the list *[0, 1, ..., n-1]*, and the functions *List.map* and *List.foldl* and *List.foldr* implement the usual map and fold functions for lists. By opening the *List* namespace, we can refer to these as *range*, *map*, *foldl*, and *foldr*. In the examples below, the dollar sign has the same effect as putting parentheses around everything that appears afterward.

```
#eval List.range 7

section
open List

#eval range 7
#eval addNums $ range 7
#eval map (fun x => x + 3) $ range 7
#eval foldl (. + .) 0 $ range 7

end
```

The scope of the *open* command is limited to the section, and the cryptic inscription *(. + .)* is notation for the addition function. Lean also supports projection notation that is useful when the corresponding namespace is not open:

```
def myRange := List.range 7
#eval myRange.map fun x => x + 3
```

Because *myRange* has type *List Nat*, Lean interprets *myrange.map fun x => x + 3* as *List.map (fun x => x + 3) myrange*. In other words, it automatically interprets *map* as being in the *List* namespace, and then it interprets *myrange* as the first *List* argument.

This course assumes you have some familiarity with functional programming. There is a free online textbook, Learn You a Haskell for Great Good that you might find helpful; porting some of the examples there to Lean is a good exercise. We will all suffer from the fact that documentation for Lean 4 barely exists at the moment, but we will do our best to provide you with enough examples for you to be able to figure out how to do what you need to do. One trick is to nose around the Lean code base itself. If you ctrl-click on the name of a function in the Lean library, the editor will jump to the definition, and you can look around and see what else is there. Another strategy is simply to ask us, ask each other, or ask questions on the Lean Zulip chat. We are all in this together.

When working with a functional programming language, there are often clever tricks for doing things that you may be more comfortable doing in an imperative programming language. For example, as explained in Section 2.3, here are Lean's definitions of the *reverse* and *append* functions for lists:

```
namespace hidden

def reverseAux : List α → List α → List α
  | [],   r => r
  | a::l, r => reverseAux l (a::r)

def reverse (as : List α) :List α :=
  reverseAux as []

protected def append (as bs : List α) : List α :=
  reverseAux as.reverse bs

end hidden
```

The function *reverseAux l r* reverses the elements of list *l* and adds them to the front of *r*. When called from *reverse l*, the argument *r* acts as an *accumulator*, storing the partial result. Because *reverseAux* is tail recursive, Lean's compiler can implement it efficiently as a loop rather than a recursive function. We have defined these functions in a namespace named *hidden* so that they don't conflict with the ones in Lean's library if you open the *List* namespace.

In Lean's foundation, every function is totally defined. In particular, every function that Lean computes has to terminates (in principle) on every input. Lean 4 will eventually support arbitrary recursive definitions in which the arguments in a recursive call decrease by some measure, but some work is needed to justify these calls in the underlying foundation. In the meanwhile, we can always cheat by using the *partial* keyword, which will let us perform arbitrary recursive calls.

```
partial def gcd m n :=
  if n = 0 then m else gcd n (m % n)

#eval gcd 45 30
#eval gcd 37252 49824
```

Using *partial* takes us outside the formal foundation; Lean will not let us prove anything about *gcd* when we define it this way. Using *partial* also makes it easy for us to shoot ourselves in the foot:

```
partial def bad (n : Nat) : Nat := bad (n + 1)
```

On homework exercises, you should try to use structural recursion when you can, but don't hesitate to use *partial* whenever Lean complains about a recursive definition. We will not penalize you for it.

The following definition of the Fibonacci numbers does not require the *partial* keyword:

```
def fib' : Nat → Nat
  | 0 => 0
  | 1 => 1
  | n + 2 => fib' (n + 1) + fib' n
```

But it is inefficient; you should convince yourself that the natural evaluation strategy requires exponential time. The following definition avoids that.

```
def fibAux : Nat → Nat × Nat
  | 0     => (0, 1)
  | n + 1 => let p := fibAux n
             (p.2, p.1 + p.2)

def fib n := (fibAux n).1

#eval (List.range 20).map fib
```

Producing a *list* of Fibonacci numbers, however, as we have done here is inefficient; you should convince yourself that the running time is quadratic. In the exercises, we ask you to define a function that computes a list of Fibonacci values with running time linear in the length of the list.

3.3 Inductive data types in Lean

One reason that computer scientists and logicians tend to like functional programming languages is that they often provide good support for defining inductive data types and then using structural recursion on such types. For example, here is a Lean definition of the extended binary trees that we defined in mathematical terms in Section 2.3:

```
import Init

inductive BinTree
  | empty : BinTree
  | node  : BinTree → BinTree → BinTree
  deriving Repr, DecidableEq, Inhabited

open BinTree
```

The command *import Init* imports a part of the initial library for us to use. The command *open BinTree* allows us to write *empty* and *node* instead of *BinTree.empty* and *BinTree.node*. Note the Lean convention of capitalizing the names of data types.

The last line of the definition, the one that begins with the word *deriving*, is boilerplate. It tells Lean to automatically generate a few additional functions that are useful. The directive *deriving Repr* tells Lean to define an internal function that can be used to represent any *BinTree* as a string. This is the string that is printed out by any *#eval* command whose argument evaluates to a *BinTree*. Adding *DecidableEq* defines a function that tests whether two *BinTrees* are equal, and adding *Inhabited* defines an arbitrary value of the data type to serve as a default value for function that need one. The following illustrates their use.

```
#eval node empty (node empty empty)

#eval empty == node empty empty  -- evaluates to false

#eval (arbitrary : BinTree)  -- BinTree.empty
```

We can now define the functions *size* and *depth* by structural recursion:

```
def size : BinTree → Nat
  | empty    => 0
  | node a b => 1 + size a + size b

def depth : BinTree → Nat
  | empty    => 0
  | node a b => 1 + Nat.max (depth a) (depth b)

def example_tree := node (node empty empty) (node empty (node empty empty))

#eval size example_tree
#eval depth example_tree
```

In fact, the *List* data type is also inductively defined.

```
#print List
```

You should try writing the inductive definition on your own. Call it *MyList*, and then try *#print MyList* to see how it compares.

3.4 Using Lean as an imperative programming language

The fact that Lean is a functional programming language means that there is no global notion of *state*. Functions take values as input and return values as output; there are no global or even local variables that are changed by the result of a function call.

But one of the interesting features of Lean is a functional programming language is that it incorporates features that make it *feel* like an imperative programming language. The following example shows how to print out, for each value i less than 100, the the sum of the numbers up to i.

```
def showSums : IO Unit := do
  let mut sum := 0
  for i in [0:100] do
    sum := sum + i
    IO.println s!"i: {i}, sum: {sum}"

#eval showSums
```

You can use a loop not just to print values, but also to compute values. The following is a boolean test for primality:

```
def isPrime (n : Nat) : Bool := do
  if n < 2 then false else
    for i in [2:n] do
      if n % i = 0 then
        return false
      if i * i > n then
        return true
    true
```

You can use such a function with the list primitives to construct a list of the first 10,000 prime numbers.

```
#eval (List.range 10000).filter isPrime
```

But you can also use it with Lean's support for *arrays*. Within the formal foundation these are modeled as lists, but the compiler implements them as dynamic arrays, and for efficiency it will modify values rather than copy them whenever the old value is not referred to by another part of an expression.

```
def primes (n : Nat) : Array Nat := do
  let mut result := #[]
  for i in [2:n] do
    if isPrime i then
      result := result.push i
  result

#eval (primes 10000).size
```

Notice the notation: *#[]* denotes a fresh array (Lean infers the type from context), and the *Array.push* function adds a new element at the end of the array.

The following example shows how to compute a two-dimensional array, a ten by ten multiplication table.

```
def mulTable : Array (Array Nat) := do
  let mut table := #[]
  for i in [:10] do
    let mut row := #[]
    for j in [:10] do
      row := row.push ((i + 1) * (j + 1))
    table := table.push row
  table

#eval mulTable
```

Alternatively, you can use the function *Array.mkArray* to initialize an array (in this case, to the values 0), and then use the *Array.set!* function to replace the elements later one.

```
def mulTable' : Array (Array Nat) := do
  let mut s : Array (Array Nat) := mkArray 10 (mkArray 10 0)
  for i in [:10] do
    for j in [:10] do
      s := s.set! i $ s[i].set! j ((i + 1) * (j + 1))
  s
```

Here we replace the ith row by the previous ith row, with the jth column updated.

The following snippet prints out the table. The idiom *show T from t* is a way of telling Lean that term *t* should have type *T*. Writing *@id T t* has a similar effect, as does writing *(t : T)*. (A difference is that the first two expressions have type *T* exactly, whereas *(t : T)* ensures that *t* has a type that Lean recognizes as being equivalent to *T*.)

```
#eval show IO Unit from do
  for i in [:10] do
    for j in [:10] do
      let numstr := toString mulTable[i][j]
      -- print 1-3 spaces
      IO.print $ " ".pushn ' ' (3 - numstr.length)
      IO.print numstr
    IO.println ""
```

3.5 Exercises

1. Using operations on *List*, write a Lean function that for every n returns the list of all the divisors of n that are less than n.
2. A natural number n is *perfect* if it is equal to the sum of the divisors less than n. Write a Lean function (with return type *Bool*) that determines whether a number n is perfect. Use it to find all the perfect numbers less than 1,000.
3. Define a recursive function *sublists*(ℓ) that, for every list ℓ, returns a list of all the sublists of ℓ. For example, given the list $[1, 2, 3]$, it should compute the list

$$[[], [1], [2], [3], [1, 2], [1, 3], [2, 3], [1, 2, 3]].$$

   The elements need not be listed in that same order.
4. Prove in Lean that the length of *sublists*(ℓ) is $2^{\mathit{length}(\ell)}$.
5. Define a function *permutations*(ℓ) that returns a list of all the permutations of ℓ.
6. Prove in Lean that the length of *permutations*(ℓ) is *factorial*(*length*(ℓ)).
7. Define in Lean a function that, assuming ℓ is a list of lists representing an $n \times n$ array, returns a list of lists representing the transpose of that array.
8. Write a program that solves the Tower of Hanoi problem with n disks on the assumption that disks can only be moved to an *adjacent* peg. (See Section 2.5.)
9. Write a program that solves the Tower of Hanoi problem with n disks on the assumption that disks can only be moved clockwise. (See Section 2.5.)
10. Define a Lean data type of binary trees in which every node is numbered by a label. Define a Lean function to compute the sum of the nodes in such a tree. Also write functions to list the elements in a preorder, postorder, and inorder traversal.

CHAPTER

FOUR

PROPOSITIONAL LOGIC

We are finally ready to turn to the proper subject matter of this course, logic. We will see that although propositional logic has limited expressive power, it can be used to carry out useful combinatorial reasoning in a wide range of applications.

4.1 Syntax

We start with a stock of variables $p_0, p_1, p_2, \ldots$ that we take to range over propositions, like "the sky is blue" or "2 + 2 = 5". We'll make the interpretation of propositional logic explicit in the next section, but, intuitively, propositions are things that can be either true or false. (More precisely, this is the *classical* interpretation of propositional logic, which is the one we will focus on in this course.) Each propositional variable is a *formula*, and we also include symbols $\top$ and $\bot$ for "true" and "false" respectively. We also provide means for building new formulas from old ones. The following is a paradigm instance of an inductive definition.

Definition

The set of propositional formulas is generated inductively as follows:

- Each variable p_i is a formula.
- $\top$ and $\bot$ are formulas.
- If A is a formula, so is $\neg A$ ("not A").
- If A and B are formulas, so are
  - $A \wedge B$ ("A and B"),
  - $A \vee B$ ("A or B"),
  - $A \to B$ ("A implies B"), and
  - $A \leftrightarrow B$ ("A if and only if B").

We will see later that there is some redundancy here; we could get by with fewer connectives and define the others in terms of those. Conversely, there are other connectives that can be defined in terms of these. But the ones we have included form a *complete* set of connectives, which is to say, any conceivable connective can be defined in terms of these, in a sense we will clarify later.

The fact that the set is generated inductively means that we can use recursion to define functions on the set of propositional

formulas, as follows:

$$\begin{aligned}
\textit{complexity}(p_i) &= 0 \\
\textit{complexity}(\top) &= 0 \\
\textit{complexity}(\bot) &= 0 \\
\textit{complexity}(\neg A) &= \textit{complexity}(A) + 1 \\
\textit{complexity}(A \wedge B) &= \textit{complexity}(A) + \textit{complexity}(B) + 1 \\
\textit{complexity}(A \vee B) &= \textit{complexity}(A) + \textit{complexity}(B) + 1 \\
\textit{complexity}(A \to B) &= \textit{complexity}(A) + \textit{complexity}(B) + 1 \\
\textit{complexity}(A \leftrightarrow B) &= \textit{complexity}(A) + \textit{complexity}(B) + 1
\end{aligned}$$

The function $\textit{complexity}(A)$ counts the number of connectives. The function $\textit{depth}(A)$, defined in a similar way, computes the depth of the parse tree.

$$\begin{aligned}
\textit{depth}(p_i) &= 0 \\
\textit{depth}(\top) &= 0 \\
\textit{depth}(\bot) &= 0 \\
\textit{depth}(\neg A) &= \textit{depth}(A) + 1 \\
\textit{depth}(A \wedge B) &= \max(\textit{depth}(A), \textit{depth}(B)) + 1 \\
\textit{depth}(A \vee B) &= \max(\textit{depth}(A), \textit{depth}(B)) + 1 \\
\textit{depth}(A \to B) &= \max(\textit{depth}(A), \textit{depth}(B)) + 1 \\
\textit{depth}(A \leftrightarrow B) &= \max(\textit{depth}(A), \textit{depth}(B)) + 1
\end{aligned}$$

Here's an example of a proof by induction:

Theorem

For every formula A, we have $\textit{complexity}(A) \leq 2^{\textit{depth}(A)} - 1$.

Proof

In the base case, we have

$$\textit{complexity}(p_i) = 0 = 2^0 - 1 = 2^{\textit{depth}(p_i)} - 1,$$

and similarly for $\top$ and $\bot$. In the case for negation, assuming the claim holds of A, we have

$$\begin{aligned}
\textit{complexity}(\neg A) &= \textit{complexity}(A) + 1 \\
&\leq 2^{\textit{depth}(A)} - 1 + 1 \\
&\leq 2^{\textit{depth}(A)} + 2^{\textit{depth}(A)} - 1 \\
&\leq 2^{\textit{depth}(A)+1} - 1 \\
&= 2^{\textit{depth}(\neg A)} - 1.
\end{aligned}$$

Finally, assuming the claim holds of A and B, we have

$$\begin{aligned}
\textit{complexity}(A \wedge B) &= \textit{complexity}(A) + \textit{complexity}(B) + 1 \\
&\leq 2^{\textit{depth}(A)} - 1 + 2^{\textit{depth}(B)} - 1 + 1 \\
&\leq 2 \cdot 2^{\max(\textit{depth}(A), \textit{depth}(B))} - 1 \\
&= 2^{\max(\textit{depth}(A), \textit{depth}(B))+1} - 1 \\
&= 2^{\textit{depth}(A \wedge B)} - 1,
\end{aligned}$$

and similarly for the other binary connectives.

In our metatheory, we will use variables $p, q, r, \ldots$ to range over propositional variables, and $A, B, C, \ldots$ to range over propositional formulas. The formulas p_i, $\top$, and $\bot$ are called *atomic* formulas. If A is a formula, B is a *subformula* of A if B occurs somewhere in A. We can make this precise by defining the set of subformulas of any formula A inductively as follows:

$$\begin{aligned} \textit{subformulas}(A) &= \{A\} \quad \text{if } A \text{ is atomic} \\ \textit{subformulas}(\neg A) &= \{\neg A\} \cup \textit{subformulas}(A) \\ \textit{subformulas}(A \star B) &= \{A \star B\} \cup \textit{subformulas}(A) \cup \textit{subformulas}(B) \end{aligned}$$

In the last clause, the star is supposed to represent any binary connective.

If A and B are formulas and p is a propositional variable, the notation $A[B/p]$ denotes the result of substituting B for p in A. Beware: the notation for this varies widely; $A[p \mapsto B]$ is also becoming common in computer science. The meaning is once again given by a recursive definition:

$$\begin{aligned} p_i[B/p] &= \begin{cases} B & \text{if } p \text{ is } p_i \\ p_i & \text{otherwise} \end{cases} \\ (\neg C)[B/p] &= \neg(C[B/p]) \\ (C \star D)[B/p] &= C[B/p] \star D[B/p] \end{aligned}$$

4.2 Semantics

Consider the formula $p \wedge (\neg q \vee r)$. Is it true? Well, that depends on the propositions p, q, and r. More precisely, it depends on whether they are true — and, in fact, that is all it depends on. In other words, once we specify which of p, q, and r are true and which are false, the truth value of $p \wedge (\neg q \vee r)$ is completely determined.

To make this last claim precise, we will use the set $\{\top, \bot\}$ to represent the truth values *true* and *false*. It doesn't really matter what sorts of mathematical objects those are, as long as they are distinct. You can take them to be the corresponding propositional formulas, or you can take $\top$ to be the number 1 and $\bot$ to be the number 0. A *truth assignment* is a function from propositional variables to the set $\{\top, \bot\}$, that is, a function which assigns a value of true or false to each propositional variable. Any truth assignment v extends to a function $\bar{v}$ that assigns a value of $\top$ or $\bot$ to any propositional formula. It is defined recursively as follows:

$$\begin{aligned} \bar{v}(p_i) &= v(p_i) \\ \bar{v}(\top) &= \top \\ \bar{v}(\bot) &= \bot \\ \bar{v}(\neg A) &= \begin{cases} \top & \text{if } \bar{v}(A) = \bot \\ \bot & \text{otherwise} \end{cases} \\ \bar{v}(A \wedge B) &= \begin{cases} \top & \text{if } \bar{v}(A) = \top \text{ and } \bar{v}(B) = \top \\ \bot & \text{otherwise} \end{cases} \\ \bar{v}(A \vee B) &= \begin{cases} \top & \text{if } \bar{v}(A) = \top \text{ or } \bar{v}(B) = \top \\ \bot & \text{otherwise} \end{cases} \\ \bar{v}(A \to B) &= \begin{cases} \top & \text{if } \bar{v}(A) = \bot \text{ or } \bar{v}(B) = \top \\ \bot & \text{otherwise} \end{cases} \\ \bar{v}(A \leftrightarrow B) &= \begin{cases} \top & \text{if } \bar{v}(A) = \bar{v}(B) \\ \bot & \text{otherwise} \end{cases} \end{aligned}$$

It is common to write $[\![A]\!]_v$ instead of $\bar{v}(A)$. Double-square brackets like these are often used to denote a semantic value that is assigned to a syntactic object. Think of $[\![A]\!]_v$ as giving the *meaning* of A with respect to the interpretation given by v. In this case, variables are interpreted as standing for truth values and the meaning of the formula is the resulting truth value, but in the chapters to come we will come across other semantic interpretations of this sort.

The following definitions are now fundamental to logic. Make sure you are clear on the terminology and know how to use it. If you can use these terms correctly, you can pass as a logician. If you get the terminology wrong, you'll be frowned upon.

- If $[\![A]\!]_v = \top$, we say that A is *satisfied* by v, or that v is a *satisfying assignment* for A. We also sometimes write $\models_v A$.
- A formula A is *satisfiable* if there is some truth assignment that satisfies it. A formula A is *unsatisfiable* if it is not satisfiable.
- A formula A is *valid*, or a *tautology* if it is satisfied by *every* truth assignment. In other words, A is valid if $[\![A]\!]_v = \top$ for every truth assignment v.
- If Γ is a set of propositional formulas, we say that Γ is *satisfied by* v if every formula in Γ is satisfied by v. In other words, Γ is satisfied by v if $[\![A]\!]_v = \top$ for every A in Γ.
- A set of formulas Γ is *satisfiable* if it is satisfied by some truth assignment v. Otherwise, it is *unsatisfiable*.
- If Γ is a set of propositional formulas and A is a propositional formula, we say Γ entails A if every truth assignment that satisfies Γ also satisfies A. Roughly speaking, this says that whenever the formulas in Γ are true, then A is also true. In this case, A is also said to be a *logical consequence* of Γ.
- Two formulas A and B are *logically equivalent* if each one entails the other, that is, we have $\{A\} \models B$ and $\{B\} \models A$. When that happens, we write $A \equiv B$.

There is a lot to digest here, but it is important that you become comfortable with these definitions. The mathematical analysis of truth and logical consequence is one of the crowning achievements of modern logic, and this basic framework for reasoning about expressions and their meaning has been applied to countless other settings in logic and computer science.

You should also get used to using semantic notions in proofs. For example:

Theorem

A propositional formula A is valid if and only if $\neg A$ is unsatisfiable.

Proof

A is valid if and only if $[\![A]\!]_v = \top$ for every truth assignment v. By the definition of $[\![\neg A]\!]_v$, this happens if and only if $[\![\neg A]\!]_v = \bot$ for every v, which is the same as saying that $\neg A$ is unsatisfiable.

4.3 Calculating with propositions

Remember that Leibniz imagined that one day we would be able to calculate with propositions. What he had noticed is that propositions, like numbers, obey algebraic laws. Here are some of them:

- $A \vee \neg A \equiv \top$
- $A \wedge \neg A \equiv \bot$
- $\neg\neg A \equiv A$

- $A \vee A \equiv A$
- $A \wedge A \equiv A$
- $A \vee \bot \equiv A$
- $A \wedge \bot \equiv \bot$
- $A \vee \top \equiv \top$
- $A \wedge \top \equiv A$
- $A \vee B \equiv B \vee A$
- $A \wedge B \equiv B \wedge A$
- $(A \vee B) \vee C \equiv A \vee (B \vee C)$
- $(A \wedge B) \wedge C \equiv A \wedge (B \wedge C)$
- $\neg(A \wedge B) \equiv \neg A \vee \neg B$
- $\neg(A \vee B) \equiv \neg A \wedge \neg B$
- $A \wedge (B \vee C) \equiv (A \wedge B) \vee (A \wedge C)$
- $A \vee (B \wedge C) \equiv (A \vee B) \wedge (A \vee C)$
- $A \wedge (A \vee B) \equiv A$
- $A \vee (A \wedge B) \equiv A$

The equivalences $\neg(A \wedge B) \equiv \neg A \vee \neg B$ and $\neg(A \vee B) \equiv \neg A \wedge \neg B$ are known as *De Morgan's laws*. It is not hard to show that all the logical connectives respect equivalence, and hence substituting equivalent formulas for a variable in a formula results in equivalent formulas. This means that, as Leibniz imagined, we can prove that a Boolean formula is valid by calculating to show that it is equivalent to $\top$. Here is an example.

Theorem

For any propositional formulas A and B, we have $(A \wedge \neg B) \vee B \equiv A \vee B$.

Proof

$$\begin{aligned}(A \wedge \neg B) \vee B &\equiv (A \vee B) \wedge (\neg B \vee B) \\ &\equiv (A \vee B) \wedge \top \\ &\equiv (A \vee B).\end{aligned}$$

Mathematicians have a trick, called *quotienting*, for turning an equivalence relation into an equality. If we interpret A, B, and C as *equivalence classes* of formulas instead of formulas, the equivalences listed above become identities. The resulting algebraic structure is known as a *Boolean algebra*, and we can view the preceding proof as establishing an identity that holds in any Boolean algebra. The same trick is used, for example, to interpret an equivalence between numbers modulo 12, like $5 + 9 \equiv 2$ as an identity on the structure $\mathbb{Z}/12\mathbb{Z}$.

4.4 Complete sets of connectives

You may have noticed that our choice of connectives is redundant. For example, the following equivalences show that we can get by with $\neg$, $\vee$, and $\bot$ alone:

$$\begin{aligned} A \wedge B &\equiv \neg(\neg A \vee \neg B) \\ A \to B &\equiv \neg A \vee B \\ A \leftrightarrow B &\equiv (A \to B) \wedge (B \to A) \\ \top &\equiv \neg\bot \end{aligned}$$

We can even define $\bot$ as $P \wedge \neg P$ for any propositional variable P, though that has the sometimes annoying consequence that we cannot express the constants $\top$ and $\bot$ without using a propositional variable.

Let $f(x_0, \ldots, x_{n-1})$ be a function that takes n truth values and returns a truth value. A formula A with variables $p_0, \ldots, p_{n-1}$ is said to *represent* f if for every truth assignment *v*,

$$[\![A]\!]_v = f(v(p_0), \ldots, v(p_{n-1})).$$

If you think of f as a truth table, this says that the truth table of A is f.

A set of connectives is said to be *complete* if every function f is represented by some formula A involving those connectives. In class, we'll discuss how to prove that $\{\wedge, \neg\}$ is a complete set of connectives in that sense.

It is now straightforward to show that a certain set of connectives is complete: just show how to define $\vee$ and $\neg$ in terms of them. Showing that a set of connectives is *not* complete typically requires some more ingenuity. One idea, as suggested in Section 2.4, is to look for some invariant property of the formulas that *are* represented.

4.5 Normal forms

For both theoretical reasons and practical reasons, it is often useful to know that formulas can be expressed in particularly simple or convenient forms.

Definition

An *atomic* formula is a variable or one of the constants $\top$ or $\bot$. A *literal* is an atomic formula or a negated atomic formula.

Definition

The set of propositional formulas in *negation normal form* (NNF) is generated inductively as follows:

- Each literal is in negation normal form.
- If A and B are in negation normal form, then so are $A \wedge B$ and $A \vee B$.

More concisely, the set of formulas in negation normal form is the smallest set of formulas containing the literals and closed under conjunction and disjunction. If we identify $\top$ with $\neg\bot$ and $\neg\top$ with $\bot$, we can alternatively characterize the formulas in negation normal form as the smallest set of formulas containing $\top$, $\bot$, variables, and their negations, and closed under conjunction and disjunction.

Proposition

Every propositional formula is equivalent to one in negation normal form.

Proof

First use the identities $A \leftrightarrow B \equiv (A \to B) \wedge (B \to A)$ and $A \to B \equiv \neg A \vee B$ to get rid of $\leftrightarrow$ and $\to$. Then use De Morgan's laws together with $\neg\neg A \equiv A$ to push negations down to the atomic formulas.

More formally, we can prove by induction on propositional formulas A that both A and $\neg A$ are equivalent to formulas in negation normal form. (You should try to write that proof down carefully.) Putting a formula in negation normal form is reasonably efficient. You should convince yourself that if A is in negation normal form, then putting $\neg A$ in negation normal form amounts to switching all the following in A:

- $\top$ with $\bot$
- variables p_i with their negations $\neg p_i$
- $\wedge$ with $\vee$.

We will see that *conjunctive normal form* (CNF) and *disjunctive normal form* (DNF) are also important representations of propositional formulas. A formula is in conjunctive normal form if it can be written as conjunction of disjunctions of literals, in other words, if it can be written as a big "and" of "or" expressions:

$$\bigwedge_{i<n} \left(\bigvee_{j<m_i} \pm \ell_j \right).$$

where each ℓ_j is a literal. Here is an example:

$$(p \vee \neg q \vee r) \wedge (\neg p \vee s) \wedge (\neg r \vee s \vee \neg t).$$

We can think of $\bot$ as the empty disjunction (because a disjunction is true only when one of the disjuncts is true) and we can think of $\top$ as the empty conjunction (because a conjunction is true when all of its conjuncts are true, which happens trivially when there aren't any).

Dually, a formula is in disjunctive normal form if it is an "or" and "and" expressions:

$$\bigvee_{i<n} \left(\bigwedge_{j<m_i} \pm \ell_j \right).$$

If you switch $\wedge$ and $\vee$ in the previous example, you have a formula in disjunctive normal form.

It is pretty clear that if you take the conjunction of two formulas in CNF the result is a CNF formula (modulo associating parentheses), and, similarly, the disjunction of two formulas in DNF is again DNF. The following is less obvious:

Lemma

The disjunction of two CNF formulas is equivalent to a CNF formula, and dually for DNF formulas.

Proof

For the first claim, we use the equivalence $A \vee (B \wedge C) \equiv (A \vee B) \wedge (A \vee C)$. By induction on n, we have that for every sequence of formulas $B_0, \ldots, B_{n-1}$ we have $A \vee \bigwedge_{i<n} B_i \equiv \bigwedge_{i<n}(A \vee B_i)$. Then by induction on n' we have $\bigwedge_{i'<n'} A_{i'} \vee \bigwedge_{i<n} B_i \equiv \bigwedge_{i'<n'} \bigwedge_{i<n}(A_{i'} \vee B_i)$. Since each $A_{i'}$ and each B_i is a disjunction of literals, this yields the result.

The second claim is proved similarly, switching $\wedge$ and $\vee$.

Proposition

Every propositional formula is equivalent to one in conjunctive normal form, and also to one in disjunctive normal form.

Proof

Since we already know that every formula is equivalent to one in negation normal form, we can use induction on that set of formulas. The claim is clearly true of $\top$, $\bot$, p_i, and $\neg p_i$. By the previous lemma, whenever it is true of A and B, it is also true of $A \wedge B$ and $A \vee B$.

In contrast to putting formulas in negation normal form, the exercises below show that the smallest CNF or DNF equivalent of a formula A can be exponentially longer than A.

We will see that conjunctive normal form is commonly used in automated reasoning. Notice that if a disjunction of literals contains a duplicated literal, deleting the duplicate results in an equivalent formula. We can similarly delete any occurrence of $\bot$. A disjunction of literals is called a *clause*. Since the order of the disjuncts and repetitions don't matter, we generally identify clauses with the corresponding set of literals; for example, the clause $p \vee \neg q \vee r$ is associated with the set $\{p, \neg q, r\}$. If a clause contains a pair p_i and $\neg p_i$, or if it contains $\top$, it is equivalent to $\top$. If Γ is a set of clauses, we think of Γ as saying that all the clauses in Γ are true. With this identification, every formula in conjunctive normal form is equivalent to a set of clauses. An empty clause corresponds to $\bot$, and an empty set of clauses corresponds to $\top$.

The dual notion to a clause is a conjunction like $\neg p \wedge q \wedge \neg r$. If each variable occurs at most once (either positively or negatively), we can think of this as a *partial truth assignment*. In this example, any truth assignment that satisfies the formula has to set p false, q true, and r false.

4.6 Exercises

1. Prove that if A is a subformula of B and B is a subformula of C then A is a subformula of C. (Hint: prove by induction on C that for every $B \in \mathit{subformulas}(C)$, every subformula of B is a subformula of C.)
2. Prove that for every A, B, and p, $\mathit{depth}(A[B/p]) \leq \mathit{depth}(A) + \mathit{depth}(B)$.
3. Prove that A and B are logically equivalent if and only if the formula $A \leftrightarrow B$ is valid.
4. Use algebraic calculations to show that all of the following are tautologies:
   - $((A \wedge \neg B) \vee B) \leftrightarrow (A \vee B)$
   - $(A \to \neg A) \to \neg A$
   - $(A \to B) \leftrightarrow (\neg B \to \neg A)$
   - $A \to (B \to A \wedge B)$
5. The *Sheffer stroke* $A \mid B$, also known as "nand," says that A and B are not both true. Show that $\{|\}$ is a complete set of connectives. Do the same for "nor," that is, the binary connective that holds if neither A nor B is true.
6. Show that $\{\wedge, \neg\}$ and $\{\to, \bot\}$ are complete sets of connectives.
7. Show that $\{\to, \vee, \wedge\}$ is not a complete set of connectives. Conclude that $\{\to, \vee, \wedge, \leftrightarrow, \top\}$ is not a complete set of connectives.
8. Show that $\{\bot, \leftrightarrow\}$ is not a complete set of connectives. Conclude that $\{\bot, \top, \neg, \leftrightarrow, \oplus\}$ is not complete. Here $A \oplus B$ is the "exclusive or," which is to say, $A \oplus B$ is true if one of A or B is true but not both.
9. Using the property $A \vee (B \wedge C) \equiv (A \vee B) \wedge (A \vee C)$ and the dual statement with $\wedge$ and $\vee$ switched, put $(p_1 \wedge p_2) \vee (q_1 \wedge q_2) \vee (r_1 \wedge r_2)$ in conjunctive normal form.

10. The boolean function *parity*$(x_0, x_1, \ldots, x_{n-1})$ holds if and only if an odd number of the x_i s are true. It is represented by the formula $p_0 \oplus p_1 \oplus \cdots \oplus p_{n-1}$. Show that any CNF formula representing the parity function has to have at least 2^n clauses.

CHAPTER

FIVE

IMPLEMENTING PROPOSITIONAL LOGIC

5.1 Syntax

We have seen that the set of propositional formulas can be defined inductively, and we have seen that Lean makes it easy to specify inductively defined types. It's a match made in heaven! Here is the definition of the type of propositional formulas that we will use in this course:

```
namespace hidden

inductive PropForm
  | tr     : PropForm
  | fls    : PropForm
  | var    : String → PropForm
  | conj   : PropForm → PropForm → PropForm
  | disj   : PropForm → PropForm → PropForm
  | impl   : PropForm → PropForm → PropForm
  | neg    : PropForm → PropForm
  | biImpl : PropForm → PropForm → PropForm
  deriving Repr, DecidableEq

end hidden

#print PropForm

open PropForm

#check (impl (conj (var "p") (var "q")) (var "r"))
```

You can find this example in the file *implementing_propositional_logic/examples.lean* in the *User* folder of the course repository.

The command *import LAMR.Util.Propositional* at the top of the file imports the part of the library with functions that we provide for you to deal with propositional logic. We will often put a copy of a definition from the library in an examples file so you can experiment with it. Here we have put it in a namespace called *hidden* so that our copy's full name is *hidden.PropForm*, which won't conflict with the one in the library. Outside the *hidden* namespace, the command *#print PropForm* refers to the real one, that is, the one in the library. The command *open PropForm* means that we can write, for example, *tr* for the first constructor instead of *PropForm.tr*. Try writing some propositional formulas of your own. There should be squiggly blue lines under the the *#print* and *#check* commands in VSCode, indicating that there is Lean output associated with these. You can see it by hovering over the commands, or by moving the caret to the command and checking the *Lean infoview* window.

The phrase *deriving Repr, DecidableEq* tells Lean to automatically define functions to be used to test equality of two expressions of type *PropForm* and to display the result of an *#eval*. We'll generally leave these out of the display from now on. You an always use *#check* and *#print* to learn more about a definition in the library. If you hold down *ctrl* and

click on an identifier, the VSCode Lean extension will take you to the definition in the library. Simply holding down *ctrl* and hovering over it will show you the definition in a pop-up window. Try taking a look at the definition of *PropForm* in the library.

Writing propositional formulas using constructors can be a pain in the neck. In the library, we have used Lean's mechanisms for defining new syntax to implement nicer syntax.

```
#check prop!{p ∧ q → (r ∨ ¬ p) → q}
#check prop!{p ∧ q ∧ r → p}

def propExample := prop!{p ∧ q → r ∧ p ∨ ¬ s1 → s2 }

#print propExample
#eval propExample

#eval toString propExample
```

You can get the symbols by typing *\and*, *\to*, *\or*, *\not*, and *\iff* in VS Code. And, in general, when you see a symbol in VSCode, hovering over it with the mouse shows you how to type it. Once again, try typing some examples of your own. The library defines the function *PropForm.toString* that produces a more readable version of a propositional formula, one that, when inserted within the *prop!{…}* brackets, should produce the same result.

Because *PropForm* is inductively defined, we can easily define functions using structural recursion.

```
namespace PropForm

def complexity : PropForm → Nat
  | var _ => 0
  | tr => 0
  | fls => 0
  | neg A => complexity A + 1
  | conj A B => complexity A + complexity B + 1
  | disj A B => complexity A + complexity B + 1
  | impl A B => complexity A + complexity B + 1
  | biImpl A B => complexity A + complexity B + 1

def depth : PropForm → Nat
  | var _ => 0
  | tr => 0
  | fls => 0
  | neg A => depth A + 1
  | conj A B => Nat.max (depth A) (depth B) + 1
  | disj A B => Nat.max (depth A) (depth B) + 1
  | impl A B => Nat.max (depth A) (depth B) + 1
  | biImpl A B => Nat.max (depth A) (depth B) + 1

def vars : PropForm → List String
  | var s => [s]
  | tr => []
  | fls => []
  | neg A => vars A
  | conj A B => (vars A).union (vars B)
  | disj A B => (vars A).union (vars B)
  | impl A B => (vars A).union (vars B)
  | biImpl A B => (vars A).union (vars B)

#eval complexity propExample
#eval depth propExample
```

(continues on next page)

(continued from previous page)

```
#eval vars propExample

end PropForm

#eval PropForm.complexity propExample
#eval propExample.complexity
```

The function *List.union* returns concatenation of the two lists with duplicates removed, assuming that the original two lists had no duplicate elements.

5.2 Semantics

The course library defines the type *PropAssignment* to be *List (String × Bool)*. If *v* has type *PropAssignment*, you should think of the expression *v.eval s* as assigning a truth value to the variable named *s*. The following function then evaluates the truth value of any propositional formula under assignment *v*:

```
def PropForm.eval (v : PropAssignment) : PropForm → Bool
  | var s => v.eval s
  | tr => true
  | fls => false
  | neg A => !(eval v A)
  | conj A B => (eval v A) && (eval v B)
  | disj A B => (eval v A) || (eval v B)
  | impl A B => !(eval v A) || (eval v B)
  | biImpl A B => (!(eval v A) || (eval v B)) && (!(eval v B) || (eval v A))

-- try it out
#eval let v := PropAssignment.mk [("p", true), ("q", true), ("r", true)]
      propExample.eval v
```

The example at the end defines *v* to be the assignment that assigns the value *true* to the strings *"p"*, *"q"*, and *"r"* and false to all the others. This is a reasonably convenient way to describe truth assignments manually, so the library provides a function *PropAssignment.mk* and notation *propassign!{…}* to support that.

```
#check propassign!{p, q, r}

#eval propExample.eval propassign!{p, q, r}
```

You should think about how the next function manages to compute a list of all the sublists of a given list. It is analogous to the power set operation in set theory.

```
def allSublists : List α → List (List α)
  | [] => [[]]
  | (a :: as) =>
      let recval := allSublists as
      recval.map (a :: .) ++ recval

#eval allSublists propExample.vars
```

With that in hand, here is a function that computes the truth table of a propositional formula. The value of *truthTable A* is a list of pairs: the first element of the pair is the list of *true*/*false* values assigned to the elements of *vars A*, and the second element is the truth value of *A* under that assignment.

```
def truthTable (A : PropForm) : List (List Bool × Bool) :=
  let vars := A.vars
  let assignments := (allSublists vars).map (fun l => PropAssignment.mk (l.map (.,␣
↪true)))
  let evalLine := fun v : PropAssignment => (vars.map v.eval, A.eval v)
  assignments.map evalLine

#eval truthTable propExample
```

We can now use the list operation *List.all* to test whether a formula is valid, and we can use *List.some* to test whether it is satisfiable.

```
def PropForm.isValid (A : PropForm) : Bool := List.all (truthTable A) Prod.snd
def PropForm.isSat(A : PropForm) : Bool := List.any (truthTable A) Prod.snd

#eval propExample.isValid
#eval propExample.isSat
```

5.3 Normal Forms

The library defines an inductive type of negation-normal form formulas:

```
inductive Lit
  | tr  : Lit
  | fls : Lit
  | pos : String → Lit
  | neg : String → Lit

inductive NnfForm :=
  | lit  (l : Lit)       : NnfForm
  | conj (p q : NnfForm) : NnfForm
  | disj (p q : NnfForm) : NnfForm
```

It is then straightforward to define the negation operation for formulas in negation normal form, and a translation from propositional formulas to formulas in negation normal form.

```
def Lit.negate : Lit → Lit
  | tr   => fls
  | fls  => tr
  | pos s => neg s
  | neg s => pos s

def NnfForm.neg : NnfForm → NnfForm
  | lit l    => lit l.negate
  | conj p q => disj (neg p) (neg q)
  | disj p q => conj (neg p) (neg q)

namespace PropForm

def toNnfForm : PropForm → NnfForm
  | tr         => NnfForm.lit Lit.tr
  | fls        => NnfForm.lit Lit.fls
  | var n      => NnfForm.lit (Lit.pos n)
  | neg p      => p.toNnfForm.neg
```

(continues on next page)

(continued from previous page)

```
  | conj p q   => NnfForm.conj p.toNnfForm q.toNnfForm
  | disj p q   => NnfForm.disj p.toNnfForm q.toNnfForm
  | impl p q   => NnfForm.disj p.toNnfForm.neg q.toNnfForm
  | biImpl p q => NnfForm.conj (NnfForm.disj p.toNnfForm.neg q.toNnfForm)
                               (NnfForm.disj q.toNnfForm.neg p.toNnfForm)

end PropForm
```

Putting the first in the namespace *NnfForm* has the effect that given *A : NnfForm*, we can write *A.neg* instead of *NnfForm.neg A*. Similarly, putting the second definition in the namespace *PropForm* means we can write *A.toNnfForm* to put a propositional formula in negation normal form.

We can try them out on the example defined above:

```
#eval propExample.toNnfForm
#eval toString propExample.toNnfForm
```

To handle conjunctive normal form, the library defines a type *Lit* of literals. A *Clause* is then a list of literals, and a *CnfForm* is a list of clauses.

```
def Clause := List Lit

def CnfForm := List Clause
```

As usual, you can use *#check* and *#print* to find information about them, and ctrl-click to see the definitions in the library. Since, as usual, defining things using constructors can be annoying, the library defines syntax for writing expressions of these types.

```
def exLit0 := lit!{ p }
def exLit1 := lit!{ -q }

#print exLit0
#print exLit1

def exClause0 := clause!{ p }
def exClause1 := clause!{ p -q r }
def exClause2 := clause!{ r -s }

#print exClause0
#print exClause1
#print exClause2

def exCnf0 := cnf!{
  p,
  -p q -r,
  -p q
}

def exCnf1 := cnf!{
  p -q,
  p q,
  -p -r,
  -p r
}

def exCnf2 := cnf!{
```

(continues on next page)

(continued from previous page)

```
  p q,
  -p,
  -q
}

#print exCnf0
#print exCnf1
#print exCnf2

#eval toString exClause1
#eval toString exCnf2
```

Let us now consider what is needed to put an arbitrary propositional formula in conjunctive normal form. In Section 4.5, we saw that the key is to show that the disjunction of two CNF formulas is again CNF. Lean's library has a function *List.insert*, which adds an element to a list; if the element already appears in the list, it does nothing. It has a function *List.union* that will form the union of two lists; if the original two lists have no duplicates, the union won't either. Finally, we have a function *List.Union* which takes the union of a list of lists. Since clauses are lists, we can use them on clauses:

```
#eval List.insert lit!{ r } exClause0

#eval exClause0.union exClause1

#eval List.Union [exClause0, exClause1, exClause2]
```

We can now take the disjunction of a single clause and a CNF formula by taking the union of the clause with each element of the CNF formula. We can implement that with the function *List.map*:

```
#eval exCnf1.map exClause0.union
```

This applied the function "take the union with *exClause0*" to each element of *exCnf1*, and returns the resulting list. We can now define the disjunction of two CNF formulas by taking all the clauses in the first, taking the disjunction of each clause with the second CNF, and then taking the union of all of those, corresponding to the conjunctions of the CNFs. Here is the library definition, and an example:

```
def CnfForm.disj (cnf1 cnf2 : CnfForm) : CnfForm :=
(cnf1.map (fun cls => cnf2.map cls.union)).Union

#eval cnf!{p, q, u -v}.disj cnf!{r1 r2, s1 s2, t1 t2 t3}
#eval toString $ cnf!{p, q, u -v}.disj cnf!{r1 r2, s1 s2, t1 t2 t3}
```

Functional programmers like this sort of definition; it's short, clever, and inscrutable. You should think about defining the disjunction of two CNF formulas by hand, using recursions over clauses and CNF formulas. Your solution will most likely reconstruct the effect of the instance *map* and *Union* in the library definition, and that will help you understand why they make sense.

In any case, with this in hand, it is easy to define the translation from negation normal form formulas and arbitrary propositional formulas to CNF.

```
def NnfForm.toCnfForm : NnfForm → CnfForm
  | NnfForm.lit (Lit.pos s) => [ [Lit.pos s] ]
  | NnfForm.lit (Lit.neg s) => [ [Lit.neg s] ]
  | NnfForm.lit Lit.tr      => []
  | NnfForm.lit Lit.fls     => [ [] ]
  | NnfForm.conj A B        => A.toCnfForm.conj B.toCnfForm
  | NnfForm.disj A B        => A.toCnfForm.disj B.toCnfForm
```

(continues on next page)

(continued from previous page)

```
def PropForm.toCnfForm (A : PropForm) : CnfForm := A.toNnfForm.toCnfForm
```

We can try them out:

```
#eval propExample.toCnfForm

#eval prop!{(p1 ∧ p2) ∨ (q1 ∧ q2)}.toCnfForm.toString

#eval prop!{(p1 ∧ p2) ∨ (q1 ∧ q2) ∨ (r1 ∧ r2) ∨ (s1 ∧ s2)}.toCnfForm.toString
```

5.4 Exercises

1. Write a Lean function that, given any element of *PropForm*, outputs a list of all the subformulas.
2. Write a Lean function that, given a list of propositional formulas and another propositional formula, determines whether the second is a logical consequence of the first.
3. Write a Lean function that, given a clause, tests whether any literal *Lit.pos p* appears together with its negation *Lit.neg p*. Write another Lean function that, given a formula in conjunctive normal form, deletes all these clauses.

CHAPTER

SIX

DECISION PROCEDURES FOR PROPOSITIONAL LOGIC

We have seen that it is possible to determine whether or not a propositional formula is valid by writing out its entire truth table. This seems pretty inefficient; if a formula A has n variables, the truth table has 2^n lines, and hence checking it requires at least that many lines. It is still an open question, however, whether one can do substantially better. If $P \neq NP$, there is no polynomial algorithm to determine satisfiability (and hence validity.)

Nonetheless, there are procedures that seem to work better in practice. In fact, we can generally do *much* better in practice. In the next chapter, we will discuss *SAT solvers*, which are pieces of software that are are remarkably good at determining whether a propositional formula has a satisfying assignment.

Before 1990, most solvers allowed arbitrary propositional formulas as input. Most contemporary SAT solvers, however, are designed to determine the satisfiability of formulas in conjunctive normal form. Section 4.5 shows that, in principle, this does not sacrifice generality, because any propositional formula A can be transformed to an equivalent CNF formula B. The problem is that, in general, however, the smallest such B may be exponentially longer than A, which makes the transformation impractical. (See the exercises at the end of Chapter 4.) In the next section, we will show you an efficient method for associating a list of clauses to A with the property that A is satisfiable if and only if the list of clauses is. With this transformation, solvers can be used to test the satisfiability of any propositional formula.

It's easy to get confused. Remember that most formulas are neither valid nor unsatisfiable. In other words, most formulas are true for some assignments and false for others. So testing for validity and testing for satisfiability are two different things, and it is important to keep the distinction clear. But there is an important relationship between the two notions: a formula A is valid if and only if $\neg A$ is unsatisfiable. This provides a recipe for determining the validity of A, namely, use a SAT solver to determine whether $\neg A$ is satisfiable, and then change a "yes" answer to a "no" and vice-versa.

6.1 The Tseitin transformation

We have seen that if A is a propositional formula, the smallest CNF equivalent may be exponentially longer. The Tseitin transformation provides an elegant workaround: instead of looking for an *equivalent* formula B, we look for one that is *equisatisfiable*, which is to say, one that is satisfiable if and only if A is. For example, instead of distributing p across the conjunction in $p \vee (q \wedge r)$, we can introduce a new definition d for $q \wedge r$. We can express the equivalence $d \leftrightarrow (q \wedge r)$ in conjunctive normal form as

$$(\neg d \vee q) \wedge (\neg d \vee r) \wedge (\neg q \vee \neg r \vee d).$$

Assuming that equivalence holds, the original formula $p \vee (q \wedge r)$ is equivalent to $p \vee d$, which we can add to the conjunction above, to yield a CNF formula.

The resulting CNF formula implies $p \vee (q \wedge r)$, but not the other way around: $p \vee (q \wedge r)$ does not imply $d \leftrightarrow (q \wedge r)$. But the resulting formulas is equisatisfiable with the original one: given any truth assignment to the original one, we can give d the truth value of $q \wedge r$, and, conversely, for any truth assignment satisfying the resulting CNF formula, d *has* to have that value. So determining whether or not the resulting CNF formula is satisfiable is tantamount to determining whether the original one is. This may seem to be a roundabout translation, but the point is that the number of definitions is bounded

by the length of the original formula and the size of the CNF representation of each definition is bounded by a constant. So the length of the resulting formula is linear in the length of the original one.

The following code, found in the *LAMR* library in the *NnfForm* namespace, runs through a formula in negation normal form and produces a list of definitions *def_0*, *def_1*, *def_2*, and so on, each of which represents a conjunction or disjunction of two variables. We assume that none these *def* variables are found in the original formula. (A more sophisticated implementation would check the original formula and start the numbering high enough to avoid a clash.)

```
def defLit (n : Nat) := Lit.pos s!"def_{n}"

def mkDefs : NnfForm → Array NnfForm → Lit × Array NnfForm
  | lit l, defs    => (l, defs)
  | conj A B, defs =>
      let ⟨fA, defs1⟩ := mkDefs A defs
      let ⟨fB, defs2⟩ := mkDefs B defs1
      add_def conj (lit fA) (lit fB) defs2
  | disj A B, defs =>
      let ⟨fA, defs1⟩ := mkDefs A defs
      let ⟨fB, defs2⟩ := mkDefs B defs1
      add_def disj (lit fA) (lit fB) defs2
where
  add_def (op : NnfForm → NnfForm → NnfForm) (fA fB : NnfForm) (defs : Array␣
↪NnfForm) :=
    match defs.findIdx? ((. == op fA fB)) with
    | some n => (defLit n, defs)
    | none   => let newdefs := defs.push (op fA fB)
                (defLit (newdefs.size - 1), newdefs)
```

The keyword *where* is used to define an auxiliary function that is not meant to be used anywhere else. The function *mkDefs* takes an NNF formula and a list of definitions, and it returns an augmented list of definitions and a literal representing the original formula. More precisely, the list *defs* is an array of disjunctions and conjunctions of variables, where the first one corresponds to *def_0*, the next one corresponds to *def_1*, and so on. In the cases where the original formula is a conjunction or a disjunction, the function first recursively creates definitions for the component formulas and then adds a new definition for the original formula. The auxiliary function *add_def* first checks to see whether the formula to be added is already found in the array. For example, when passed a conjunction $p \wedge$ *def_0*, if that formula is already in the list of definitions in position 7, *add_def* returns *def_7* as the definition of the formula and leaves the array unchanged.

To illustrate, we start by putting the formula

$$\neg(p \wedge q \leftrightarrow r) \wedge (s \rightarrow p \wedge t)$$

in negation normal form.

```
def ex1 := prop!{¬ (p ∧ q ↔ r) ∧ (s → p ∧ t)}.toNnfForm

#eval toString ex1
```

Removing extraneous parentheses, we get

$$((p \wedge q \wedge \neg r) \vee (r \wedge (\neg p \vee \neg q)) \wedge (\neg s \vee (p \wedge t)).$$

In the following, we compute the list of definitions corresponding to *ex1*, and then we use a little program to print them out in a more pleasant form.

```
#eval  ex1.mkDefs #[]

def printDefs (A : NnfForm) : IO Unit := do
```

(continues on next page)

(continued from previous page)

```
  let ⟨fm, defs⟩ := A.mkDefs #[]
  IO.println s!"{fm}, where"
  for i in [:defs.size] do
    IO.println s!"def_{i} := {defs[i]}"

#eval printDefs ex1

/-
output:

def_7, where
def_0 := (p ∧ q)
def_1 := (def_0 ∧ (¬ r))
def_2 := ((¬ p) ∨ (¬ q))
def_3 := (r ∧ def_2)
def_4 := (def_1 ∨ def_3)
def_5 := (p ∧ t)
def_6 := ((¬ s) ∨ def_5)
def_7 := (def_4 ∧ def_6)
-/
```

We can obtain an equisatisfiable version of the formula by putting all the definitions into conjunctive normal form, collecting them all together, and adding one more conjunct with the variable d corresponding to the definition of the top level formula. There is a constant bound on the size of each CNF definition, which corresponds to a single conjunction or disjunction. (This would still be true if we added other binary connectives, like a bi-implication.) Since the number of definitions is linear in the size of the original formula, the size of the equisatisfiable CNF formula is linear in the original one.

There is an important optimization of the Tseitin transformation due to Plaisted and Greenbaum, who observed that for equisatisfiability, only one direction of the implication is needed for each subformula. If we start with a formula in negation normal form, each subformula occurs *positively*, which means that switching its truth value from negative to positive can only change the truth value of the entire formula in the same direction. In that case, only the forward direction of the implication is needed. For example, the formula $(p \wedge q) \vee r$ is equisatisfiable with $(d \vee r) \wedge (d \to p \wedge q)$, which can be expressed in CNF as $(d \vee r) \wedge (\neg d \vee p) \wedge (\neg d \vee q)$. To see that they are equisatisfiable, notice that any satisfying assignment to $(p \wedge q) \vee r$ can be extended to a satisfying assignment of $(d \vee r) \wedge (d \to p \wedge q)$ by giving d the same truth assignment as $p \wedge q$, and, conversely, $(d \vee r) \wedge (d \to p \wedge q)$ entails $(p \wedge q) \vee r$.

It isn't hard to put a implication of the form $(A \to B)$ into conjunctive normal form. The LAMR library defines a function to do that, and another one that turns the entire list of definitions returned by *mkDefs* into a single CNF formula.

```
def implToCnf (A B : NnfForm) : CnfForm :=
  (disj A.neg B).toCnfForm

def defsImplToCnf (defs : Array NnfForm) : CnfForm := aux defs.toList 0
  where aux : List NnfForm → Nat → CnfForm
  | [],          n => []
  | nnf :: nnfs, n => implToCnf (lit (defLit n)) nnf ++ aux nnfs (n + 1)
```

If we take the resulting formula and add a conjunct for the variable representing the top level, we have an equisatisfiable CNF formula, as desired.

A moment's reflection shows that we can do better. For example, if the formula A is already in CNF, we don't have to introduce any definitions at all. The following functions from the library do their best to interpret an NNF formula as a CNF formula, introducing definitions only when necessary. The first function, *NnfForm.orToCnf*, interprets a formula as a clause. For example, given the formula $p \vee (q \wedge \neg r) \vee \neg s$ and a list of definitions, it adds a definition d for $q \wedge \neg r$ and returns the clause $p \vee d \vee \neg s$. The function *NnfForm.andToCnf* does the analogous thing for conjunctions, and the function *NnfForm.toCnf* puts it all together.

```
def orToCnf : NnfForm → Clause → Array NnfForm → Clause × Array NnfForm
  | lit Lit.tr,  cls, defs  => ([Lit.tr], defs)
  | lit Lit.fls, cls, defs  => (cls, defs)
  | lit l, cls, defs        => (l :: cls, defs)
  | disj A B, cls, defs =>
      let ⟨cls1, defs1⟩ := orToCnf A cls defs
      let ⟨cls2, defs2⟩ := orToCnf B cls1 defs1
      (cls1.union cls2, defs2)
  | A, cls, defs =>
      let ⟨l, defs1⟩ := A.mkDefs defs
      (l::cls, defs1)

def andToCnf : NnfForm → Array NnfForm → CnfForm × Array NnfForm
  | conj A B, defs =>
    let ⟨fA, defs1⟩ := andToCnf A defs
    let ⟨fB, defs2⟩ := andToCnf B defs1
    (fA.union fB, defs2)
  | A, defs =>
    let ⟨cls, defs1⟩ := orToCnf A [] defs
    ([cls], defs1)

def toCnf (A : NnfForm) : CnfForm :=
  let ⟨cnf, defs⟩ := andToCnf A #[]
  cnf.union (defsImplToCnf defs)
```

The following example tests it out on *ex1.toCnf*. The comment afterward shows the resulting CNF formula and then reconstructs the definitions to show that the result is equivalent to the original formula.

```
#eval toString ex1.toCnf

/-
Here is ex1:

((p ∧ q ∧ ¬ r) ∨ (r ∧ (¬ p ∨ ¬ q)) ∧ (¬ s ∨ (p ∧ t))

Here is the CNF formula:

def_3 def_1,
def_4 -s,
-def_0 p,
-def_0 q,
-def_1 def_0,
-def_1 -r,
-def_2 -p -q,
-def_3 r,
-def_3 def_2,
-r -def_2 def_3,
-def_4 p,
-def_4 t,

Here we check to make sure it works:

def_0 := p ∧ q
def_1 := p ∧ q ∧ ¬ r
def_2 := ¬ p ∨ ¬ q
def_3 := r ∧ (¬ p ∨ ¬ q)
def_4 := p ∧ t
```

(continues on next page)

(continued from previous page)

```
def_3 def_1 := (p ∧ q ∧ ¬ r) ∨ (p ∧ q ∧ ¬ r)
def_4 -s    := ¬ s ∨ (p ∧ t)

Each ':=' is really an implication.
-/
```

There is one additional optimization that we have not implemented in the library: we can be more efficient with iterated conjunctions and disjunctions. For example, you can check that $d \to \ell_1 \wedge \cdots \wedge \ell_n$ can be represented by the conjunction of the clauses $\neg d \vee \ell_1$ to $\neg d \vee \ell_n$, and $d \to \ell_1 \vee \cdots \vee \ell_n$ can be represented by the single clause $\neg d \vee \ell_1 \vee \cdots \vee \ell_n$.

6.2 Unit propagation and the pure literal rule

In earlier chapters, we considered only truth assignments that assign a truth value to all propositional variables. However, complete search methods for SAT like DPLL use *partial assignments*, which assign truth values to some subset of the variables. For a clause C and partial truth assignment τ, we denote by $[\![C]\!]_\tau$ the reduced clause constructed by removing falsified literals. If τ satisfies a literal in C, we interpret $[\![C]\!]_\tau$ as the singleton clause consisting of $\top$, while if τ falsifies all literals in C, then $[\![C]\!]_\tau$ is the empty clause, which we take to represent $\bot$. For a CNF formula Γ, we denote by $[\![\Gamma]\!]_\tau$ the conjunction of $[\![C]\!]_\tau$ with $C \in \Gamma$, where we throw away the clauses C such that $[\![C]\!]_\tau$ is $\top$. Remember that if $[\![\Gamma]\!]_\tau$ is an empty conjunction, we interpret it as being true. In other words, we identify the empty conjunction with $\top$. On the other hand, if $[\![\Gamma]\!]_\tau$ contains an empty clause, then $[\![\Gamma]\!]$ is equivalent to $\bot$.

Notice that we are reusing the notation $[\![A]\!]_\tau$ that we introduced in Section 4.2 to describe the evaluation of a formula A under a truth assignment τ. This is an abuse of notation, because in that section we were interpreting $[\![A]\!]_\tau$ as a *truth value*, whereas now we are taking $[\![C]\!]_\tau$ to be a *clause* and we are taking $[\![\Gamma]\!]_\tau$ to be *CNF formula*. But the abuse is harmless, and, indeed, quite helpful. Notice that if τ is a partial truth assignment that assigns values to all the variables in a clause C, then $[\![C]\!]_\tau$ evaluates to either the empty clause or $\top$, and if τ is a partial truth assignment that assigns values to all the variables in a CNF formula Γ, then $[\![\Gamma]\!]_\tau$ evaluates to either the empty CNF formula, $\top$, or the singleton conjunction of the empty clause, which is our CNF representation of $\bot$. In that sense, the notation we are using in this chapter is a generalization of the notation we used to describe the semantics of propositional logic.

Remember also that if τ is a truth assignment in the sense of Section 4.2, the semantic evaluation $[\![A]\!]_\tau$ only depends on the values of τ that are assigned to variables that occur in $[\![A]\!]$. In this chapter, when we say "truth assignment," we will generally mean "partial truth assignment," and the analogous fact holds: the values of $[\![C]\!]_\tau$ and $[\![\Gamma]\!]_\tau$ depend only on the variables found in C and Γ, respectively. In particular, if τ is a satisfying assignment for C, we can assume without loss of generality that τ assigns values only to the variables that occur in C, and similarly for Γ.

A key SAT-solving technique is *unit propagation*. Given a CNF formula Γ and a truth assignment τ, a clause $C \in \Gamma$ is *unit under* τ if τ falsifies all but one literal of C and the remaining literal is unassigned. In other words, C is unit under τ if $[\![C]\!]_\tau$ consists of a single literal. The only way to satisfy C is to assign that literal to true. Unit propagation iteratively extends τ by satisfying all unit clauses. This process continues until either no new unit clauses are generated by the extended τ or until the extended τ falsifies a clause in Γ.

For example, consider the partial truth assignment τ with $\tau(p_1) = \top$ and the following formula:

$$\Gamma_{\text{unit}} := (\neg p_1 \vee \neg p_3 \vee p_4) \wedge (\neg p_1 \vee \neg p_2 \vee p_3) \wedge (\neg p_1 \vee p_2) \wedge (p_1 \vee p_3 \vee p_6) \wedge (\neg p_1 \vee p_4 \vee \neg p_5) \wedge (p_1 \neg p_6) \wedge (p_4 \vee p_5 \vee p_6) \wedge (p_5 \vee \neg p_6)$$

The clause $(\neg p_1 \vee p_2)$ is unit under τ because $[\![(\neg p_1 \vee p_2)]\!]_\tau = (p_2)$. Hence unit propagation will extend τ by assigning p_2 to $\top$. Under the extended τ, $(\neg p_1 \vee \neg p_2 \vee p_3)$ is unit, which will further extend τ by assigning p_3 to $\top$. Now the clause $(\neg p_1 \vee \neg p_3 \vee p_4)$ becomes unit and thus assigns p_4 to $\top$. Ultimately, no unit clauses remain and unit propagation terminates with $\tau(p_1) = \tau(p_2) = \tau(p_3) = \tau(p_4) = \top$.

Another important simplification technique is the *pure literal rule*. A literal l is called pure in a formula Γ if no clause in Γ contains the literal $\neg l$. The pure literal rule sets pure literals to $\top$.

In contrast to unit propagation, the pure literal rule can reduce the number of satisfying assignments. Consider for example for the formula $\Gamma = (p \vee q) \wedge (\neg q \vee r) \wedge (q \vee \neg r)$. The literal p is pure, so the pure literal rule will assign it to $\tau(p) = \top$. We leave it to you to check that Γ has three satisfying assignments, while $[\![\Gamma]\!]_\tau$ has only two satisfying assignments.

6.3 DPLL

One of the first and most well-known decision procedures for SAT problems is the Davis-Putnam-Logemann-Loveland (DPLL) algorithm, which was invented by Davis, Logemann, and Loveland in 1962, based on earlier work by Davis and Putnam in 1960. DPLL is a complete, backtracking-based search algorithm that builds a binary tree of truth assignments. For nearly four decades since its invention, almost all complete SAT solvers have been based on DPLL.

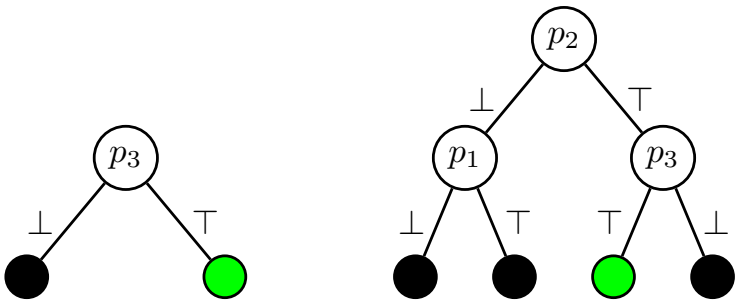
The DPLL procedure is a recursive algorithm that takes a CNF formula Γ and a truth assignment τ as input and returns a Boolean indicating whether Γ under the assignment τ can be satisfied. Each recursive call starts by extending τ using unit propagation and the pure literal rule. Afterwards, it checks whether the formula is satisfiable ($[\![\Gamma]\!]_\tau = \top$) or unsatisfiable ($[\![\Gamma]\!]_\tau = \bot$). In either of those cases it returns $\top$ or $\bot$, respectively. Otherwise, it selects an unassigned propositional variable p and recursively calls DPLL with one branch extending τ with $\tau(p) = \top$ and another branch extending τ with $\tau(p) = \bot$. If one of the calls returns $\top$ it returns $\top$, otherwise it returns $\bot$.

The effectiveness of DPLL depends strongly on the quality of the propositional variables that are selected for the recursive calls. The propositional variable selected for a recursive call is called a *decision variable*. A simple but effective heuristic is to pick the variable that occurs most frequently in the shortest clauses. This heuristics is called *MOMS* (Maximum Occurrence in clauses of Minimum Size). More expensive heuristics are based on *lookaheads*, that is, assigning a variable to a truth value and measuring how much the formula can be simplified as a result.

Consider the following CNF formula:

$$\Gamma_{\text{DPLL}} := (p_1 \vee p_2 \vee \neg p_3) \wedge (\neg p_1 \vee p_2 \vee p_3) \wedge (\neg p_1 \vee \neg p_2 \vee p_3) \wedge (p_1 \vee p_3) \wedge (\neg p_1 \vee \neg p_3)$$

Selecting p_3 as decision variable in the root node, results in a DPLL tree with two leaf nodes. However, when selecting p_2 as decision variable in the root node, the DPLL tree consists of four leaf nodes. The figure below shows the DPLL trees with green nodes denoting satisfying assignments, while black nodes denote falsifying assignments.



We have implemented a basic version of the DPLL procedure in Lean, which can be found in the *Examples* folder. We will start at the top level and work our way down. The function *dpllSat* accepts a CNF formula and returns either a satisfying assignment or *none* if there aren't any. It calls *propagateUnits* to perform unit propagation, and then calls *dpllSatAux* to iteratively pick an unassigned variable and split on that variable. We use the keyword *partial* to allow for an arbitrary recursive call.

```
partial def dpllSatAux (τ : PropAssignment) (φ : CnfForm) :
    Option (PropAssignment × CnfForm) :=
  if φ.hasEmpty then none
  else match pickSplit? φ with
  -- No variables left to split on, we found a solution.
  | none => some (τ, φ)
```

(continues on next page)

(continued from previous page)

```
  -- Split on `x`.
  -- `<|>` is the "or else" operator, which tries one action and if that fails
  -- tries the other.
  | some x => goWithNew x τ φ <|> goWithNew (-x) τ φ

where
  /-- Assigns `x` to true and continues out DPLL. -/
  goWithNew (x : Lit) (τ : PropAssignment) (φ : CnfForm) :
      Option (PropAssignment × CnfForm) :=
    let (τ', φ') := propagateWithNew x τ φ
    dpllSatAux τ' φ'

/-- Solve `φ` using DPLL. Return a satisfying assignment if found, otherwise `none`.␣
↪-/
def dpllSat (φ : CnfForm) : Option PropAssignment :=
  let ⟨τ, φ⟩ := propagateUnits [] φ
  (dpllSatAux τ φ).map fun ⟨τ, _⟩ => τ
```

The function *pickSplit?* naively chooses the first variable it finds. It follows the Lean convention of using a question mark for a function that returns an element of an option type.

```
def pickSplit? : CnfForm → Option Lit
  | []      => none
  | c :: cs => match c with
    | x :: xs => x
    | _       => pickSplit? cs
```

The function *propagateUnits* performs unit propagation and returns the resulting formula and the augmented truth assignment. As long as there is a unit clause, it simplifies the formula and adds the new literal to the assignment.

```
partial def propagateUnits (τ : PropAssignment) (φ : CnfForm) : PropAssignment ×␣
↪CnfForm :=
  -- If `φ` is unsat, we're done.
  if φ.hasEmpty then ⟨τ, φ⟩
  else match φ.findUnit with
    -- If there are no unit clauses, we're done.
    | none   => ⟨τ, φ⟩
    | some x =>
      -- If there is a unit clause `x`, simplify the formula
      -- assuming `x` is true and continue propagating.
      let φ' := simplify x φ
      if τ.mem x.name
      then panic! s!"'{x}' has already been assigned and should not appear in the␣
↪formula."
      else propagateUnits (τ.withLit x) φ'
```

The function *propagateWithNew* is used to peform a split on the literal x. It assigns the value of x and then does unit propagation.

```
def propagateWithNew (x : Lit) (τ : PropAssignment) (φ : CnfForm) :
    PropAssignment × CnfForm :=
  propagateUnits (τ.withLit x) (simplify x φ)
```

Finally, the function *simplify* simplifies a CNF formula by assigning the literal x to true, assuming x is not one of the constants $\bot$ or [?].

```
def simplify (x : Lit) (φ : CnfForm) : CnfForm :=
  assert! x != lit!{⊥} && x != lit!{T}
  match φ with
  | []      => []
  | c :: cs =>
              let cs' := simplify x cs
              -- If clause became satisfied, erase it from the CNF
              if c.elem x then cs'
              -- Otherwise erase any falsified literals
              else c.eraseAll x.negate :: cs'

#eval toString <| simplify lit!{p} cnf!{p -q p -p, p, q q, -q -p, -p}
```

6.4 Autarkies and 2-SAT

Unit propagation and the pure literal rule form the core of the DPLL algorithm. In this section, we present a generalization of the pure literal rule, and we illustrate its use by providing an efficient algorithm for a restricted version of SAT.

The notion of an *autarky* or *autarky assignment* is a generalization of the notion of a pure literal. An autarky for a set of clauses is an assignment that satisfies all the clauses that are touched by the assignment, in other words, all the clauses that have at least one literal assigned. A satisfying assignment of a set of clauses is one example of an autarky. The assignment given by the pure literal rule is another. There are often more interesting autarkies that are somewhere in between the two.

Notice that if τ is an autarky for Γ, then $[\![\Gamma]\!]_\tau \subseteq \Gamma$. To see this, suppose C is any clause in Γ. If C is touched by τ, then $[\![C]\!]_\tau = \top$, and so it is removed from $[\![\Gamma]\!]_\tau$. If C is not touched by τ, then $[\![C]\!]_\tau = C$. Since every element of $[\![\Gamma]\!]_\tau$ is of one of these two forms, we have that $[\![\Gamma]\!]_\tau \subseteq \Gamma$.

Theorem

Let τ be an autarky for formula Γ. Then Γ and $[\![\Gamma]\!]_\tau$ are satisfiability equivalent.

Proof

If Γ is satisfiable, then since $[\![\Gamma]\!]_\tau \subseteq \Gamma$, we know that $[\![\Gamma]\!]_\tau$ is satisfiable as well. Conversely, suppose $[\![\Gamma]\!]_\tau$ is satisfiable and let τ_1 be an assignment that satisfies $[\![\Gamma]\!]_\tau$. We can assume that τ_1 only assigns values to the variables of $[\![\Gamma]\!]_\tau$, which are distinct from the variables of τ. Then the assignment τ_2 which is the union of τ and τ_1 satisfies Γ.

We now turn to a restricted version of SAT. A CNF formula is called k-SAT if the length of all clauses is at most k. 2-SAT formulas can be solved in polynomial time, while k-SAT is NP-complete for $k \geq 3$. A simple, efficient decision procedure for 2-SAT uses only unit propagation and autarky reasoning. The decision procedure is based on the following observation. Given a 2-SAT formula Γ that contains a propositional variable p, unit propagation on Γ using $\tau(p) = \top$ has two possible outcomes: (1) it results in a conflict, meaning that all satisfying assignments of Γ have to assign p to $\bot$, or (2) unit propagation does not result in a conflict, in which case the extended assignment after unit propagation is an autarky. To understand the latter case, note that assigning a literal in any clause of length two either satisfies the clause (if the literal is true) or reduces it to a unit clause (if the literal is false). So if there isn't a conflict, then it is impossible that unit propagation will produce a clause that is touched but not satisfied.

The decision procedure works as follows. Given a 2-SAT formula Γ, we pick a propositional variable p occurring in Γ and compute the result of unit propagation on Γ using $\tau(p) = \top$. If unit propagation does not result in a conflict, let τ' be the extended assignment and we continue with $[\![\Gamma]\!]_{\tau'}$. Otherwise let $\tau''(p) = \bot$ and we continue with $[\![\Gamma]\!]_{\tau''}$. This process is

repeated until the formula is empty, which indicates that the original formula is satisfiable, or contains the empty clause, which indicates that the original clause is unsatisfiable.

6.5 CDCL

The Conflict-Driven Clause-Learning (CDCL) decision procedure is the most successful SAT solving paradigm. Although it evolved from DPLL, modern implementations of CDCL have hardly any resemblance with the classic decision procedure. The algorithms differ in core algorithmic choices. When designing an algorithm, one typically needs to choose between making the algorithm smart or making it fast. Having it both ways is generally not an option due to conflicting requirements for the datastructures. State-of-the-art DPLL algorithms are designed to be smart: spend serious computational effort to pick the best variable in each recursive call to make the binary tree relatively small. As we will describe below, CDCL focuses on being fast. Unit propagation is by far the most computationally expensive step and everything in a CDCL solver is designed by make that as fast as possible. That prevents more complicated heuristics. Another important design decision for SAT solving algorithms is determining whether you plan to find a satisfying assignment or prove that none exists. DPLL focuses on finding a satisfying assignment by picking the most satisfiable branch first, while CDCL, as the name suggests, prefers conflicts and wants to find a short refutation. A satisfying assignment is a counterexample to no refutation exists.

The early development of CDCL is four decades later than DPLL. While DPLL simply backtracks when a conflict (leaf node) is reached, CDCL turns this conflict into a conflict clause that is added to the formula. The first and arguably most important step towards CDCL is the invention which clause to add: the first unique implication point. The first unique implication point is an invention by Marques-Silva and Sakallah (1996) and is still used in all modern day top-tier solvers.

The best decision heuristic for CDCL selects the unassigned variable that occurs most frequently in recently learned conflict clauses. This heuristic, called Variable-State Independent Decaying Sum (VSIDS in short), was introduced in the zChaff SAT solver by Moskewicz and co-authors (2001). The heuristic was been improved over the years, in particular by E'en and S"orensson (2003) in their solver MiniSAT. In recent years some other advances have been made as well. However, selecting variables in recent conflict clauses is still the foundation.

Adding clauses to the input formula could significantly increase the cost of unit propagation. Therefore clause learning was not immediately successful. The contribution that showed the effectiveness of clause learning in practice is the 2-literal watchpointers datastructure. In short, the solver does not maintain a full occurrence list of clauses, but only an occurrence list of the first two literals in a clause. These first two literals are required to have the following invariant: either both literals are unassigned or one of them is assigned to true. That invariant is enough to ensure that the clause is not unit. In case an assignment breaks the invariant, then the entire clause is explored to check whether it is possible to fix the invariant by swapping a satisfied or unassigned literal to the first two positions. If this is not possible, the clause is unit and the unassigned literal is assigned to true or the clause is falsified which triggers the learning procedure in the solver.

Another key difference between CDCL and DPLL is that the former restarts very frequently. Restarting sounds drastic, but simply means that all variables are unassigned. The remainder of the solver state, such as the heuristics and the learned clauses, are not altered. Modern SAT solver restart often: say roughly a thousand times per second (although this depends on the size of the formula). As a consequence, no binary search tree is constructed. Instead, CDCL might be best viewed a conflict-clause generator without any systematic search.

The are various other parts in CDCL solvers that are important for fast performance but that are beyond the scope of this course. For example, CDCL solvers do not only learn many clauses, but they also aggressively delete them to reduce the impact on unit propagation. Also, CDCL solvers use techniques to rewrite the formula, so called inprocessing techniques. In top-tier solvers, about half the runtime can be devoted to such techniques.

CHAPTER

SEVEN

USING SAT SOLVERS

A satisfiability (SAT) solver determines whether a propositional formula has a satisfying assignment. The performance of SAT solvers has improved significantly in the last two decades. In the late 1990s, only formulas with thousands of variables and thousands of clauses could be solved. Today, many propositional formulas with millions of variables and millions of clauses can be solved. In this chapter, we will explain how to use SAT solvers and how to encode problems into propositional logic.

7.1 First examples

Remember that contemporary SAT solvers determine that satisfiability of propositional formulas in conjunctive normal form. Specifically, they use a format for specifying such formulas known as the *DIMACS* format. Our *LAMR* library proves a function that converts any CNF formula to that format, sends it to a SAT solver called *CaDiCaL*, and parses the answer. The following can be found in *Examples/using_sat_solver/examples.lean*:

```
def cadicalExample : IO Unit := do
  let (s, result) ← callCadical exCnf0
  IO.println "Output from CaDiCaL :\n"
  --IO.println s
  --IO.println "\n\n"
  IO.println (formatResult result)
  pure ()

#eval cadicalExample
```

It uses the same example CNF formulas defined in Section 5.3. You can change *exCnf0* to *exCnf1* or *exCnf2*, or use any other CNF formula you want. If you uncomment the two lines that begin *IO.println*, the Lean output will show you the raw output from CaDiCaL.

7.2 Encoding problems

All NP-complete problems can be transformed in polynomial time into a SAT problem (i.e., into a propositional formula). For many problems, such a transformation is quite natural. For some other problems, the transformation can be complicated. The transformation is not unique. Frequently there exist many way to encode a problem as a propositional formula. The encoding can have a big impact on the runtime of SAT solvers. Generally, the smallest encoding for a problem (in terms of the number of variables and the number of clauses) results in relatively strong performance. In this section we will describe a few encodings.

One way to encode a problem into propositional logic is to describe it first using some high-level constraints. Let's consider a couple of high-level constraints: Constrain a sequence of literals such that at least one of them is true (*atLeastOne*), or

that at most one of the is true (*atMostOne*), or that an odd number of them is true (*XOR*). Afterwards these constraints are encoded into propositional logical to obtain, so a SAT solver can be used to solve the resulting formula.

How to encode *atLeastOne*, *atMostOne*, and *XOR* as a set of clauses? The constraint *atLeastOne* is easy: simply use the disjunction of all the literals in the sequence. The second constraint is requires multiple clauses. The naive way generates a quadratic number of clauses: for each pair of literals (l_i, l_j) in the sequence, include the clause $\neg l_i \vee \neg l_j$. The naive way of encoding the *XOR* constraint results in an exponential number of clauses: all possible clauses over the literals such that an odd number of them are negated. For example, the encoding of *XOR*(l_1, l_2, l_3) produces the following clauses: $l_1 \vee l_2 \vee \neg l_3, l_1 \vee \neg l_2 \vee l_3, \neg l_1 \vee l_2 \vee l_3, \neg l_1 \vee \neg l_2 \vee \neg l_3$

Although a quadratic number of clauses produced by can be acceptable *atMostOne* for a reasonable small sequence of literals, the exponential number of clauses produced by *XOR* would result in formulas that are hard to solve solely due to the size of the formula. Fortunately, one can encode both *atMostOne* and *XOR* using a linear number of clauses using the following trick: In case the sequence consists of more than four literals, split the constraint into two such that the first uses the first three literals of the sequence appended by a new literal y, while the second uses the remainder of the sequence appended by the literal $\neg y$. For example, *atMostOne*$(l_1, \ldots, l_n)$ is split into *atMostOne*(l_1, l_2, l_3, y) and *atMostOne*$(l_4, \ldots, l_n, \neg y)$. The splitting is repeated until none of the constraints has a sequence longer than four.

Another approach to encode a problem into propositional logic is to express it first as another NP-complete problem and afterwards transform the result into propositional logic. Let's demonstrate this approach for graph coloring. The graph coloring problem asks whether a given graph can be colored with a given number of colors such that adjacent vertices have different colors. Graph coloring problems can be easily encoded into a propositional formula, and SAT solvers can frequently solve these formulas efficiently.

Given a graph $G = (V, E)$ and k colors, the encoding uses $k|V|$ Boolean variables $x_{i,j}$ with $i \in \{1, \ldots, |V|\}$ and $j \in \{1, \ldots, k\}$ and $|V| + k|E|$ clauses. If $x_{i,j}$ is assigned to true it means that vertex i is assigned color j. The clauses encode two constraints: each vertex has a color and adjacent vertices have a different color. The first constraint can be encoded using a single clause per vertex. For example, for vertex i, we have the following clause: $x_{i,1} \vee \cdots \vee x_{i,k}$. The second constraint requires k binary clauses. For example, for an edge between vertices h and i, we have the following binary clauses: $(\neg x_{h,1} \vee \neg x_{i,1}) \wedge \cdots \wedge (\neg x_{h,k} \vee \neg x_{i,k})$.

The CNF formulas for a triangle (a fully connected graph with three vertices) for two colors and three colors is shown below. The first one is unsatisfiable, while the second one is satisfiable.

```
def triangleCnf2 := !cnf{
  x11 x12,
  x21 x22,
  x31 x32,
  -x11 -x21, -x12 -x22,
  -x11 -x31, -x12 -x32,
  -x21 -x31, -x22 -x32
}

def triangleCnf3 := !cnf{
  x11 x12 x13,
  x21 x22 x23,
  x31 x32 x33,
  -x11 -x21, -x12 -x22, -x13 -x23,
  -x11 -x31, -x12 -x32, -x13 -x33,
  -x21 -x31, -x22 -x32, -x23 -x23
}
```

Many problems, such as scheduling and planning, can naturally be expressed as a graph coloring problem. We can then transform the graph coloring problem into a SAT problem using the encoding described above.

An example of a problem that can be expressed as a graph coloring problem is the popular puzzle Sudoku: Place number is a grid consisting of nine squares subdivided into a further nine smaller squares in such a way that every number appears once in each horizontal line, vertical line, and square. This puzzle can be seen as a graph coloring where each small square

is a vertex, and vertices are connected if and only if the corresponding small squares occur the the same horizontal line, vertical line, or square. Below is one of the hardest sudoko puzzles with only 17 given numbers.

	4		3					
						7	9	
			6					
			1	4		5		
9							1	
2								6
				7	2			
	5					8		
				9				

If you take a look at the file *sudoku.lean* in the *Examples* folder, you will see that it is easily solved by a SAT solver.

7.3 Exercise: grid coloring

Ramsey Theory deals with patterns that cannot be avoided indefinitely. In this exercise we focus on a pattern of coloring a $n \times m$ grid with k colors: Consider all possible rectangles within the grid whose length and width are at least 2. Try to color the grid using k colors so that no such rectangle has the same color for its four corners. When this is possible, we say that the $n \times m$ grid is k-colorable while avoiding monochromatic rectangles. When using k colors, it is relatively easy to construct a valid $k^2 \times k^2$ grid. However, only few valid grids that are larger than $k^2 \times k^2$ are known. An example of a valid 3-coloring of the 9×9 grid is shown below.

```
0 0 1 1 2 2 0 1 2
2 0 0 1 1 2 2 0 1
1 2 0 0 1 1 2 2 0
0 1 2 0 0 1 1 2 2
2 0 1 2 0 0 1 1 2
2 2 0 1 2 0 0 1 1
1 2 2 0 1 2 0 0 1
1 1 2 2 0 1 2 0 0
0 1 1 2 2 0 1 2 0
```

Step 1. Encode whether there exists a coloring of the grid using three colors so that no such rectangle has the same color for its four corners. The encoding requires two types of constraints. First, each square needs to have at least one color. Second, if four squares form the corners of a rectangle, then they cannot have the same color.

Step 2. Solve the encoding for a 10×10 grid using a SAT solver and decode the solution into a valid coloring. Show the output of the SAT solver and a valid 3-coloring similar to the one above of the 9×9 grid.

Note that any valid coloring can be turned into another valid coloring by permuting the rows, columns, or colors. However, such valid colorings are isomorphic.

Step 3. Use the tool Shatter to (partially) break the symmetries of the encoding in Step 1 and count the number of solutions of the resulting formula.

7.4 Exercise: NumberMind

The game Number Mind is a variant of the well known game Master Mind.

Instead of colored pegs, you have to guess a secret sequence of digits. After each guess you're only told in how many places you've guessed the correct digit. So, if the sequence was 1234 and you guessed 2036, you'd be told that you have one correct digit; however, you would NOT be told that you also have another digit in the wrong place.

For instance, given the following guesses for a 5-digit secret sequence,

```
90342 ;2 correct
70794 ;0 correct
39458 ;2 correct
34109 ;1 correct
51545 ;2 correct
12531 ;1 correct
```

The correct sequence 39542 is unique.

Based on the following guesses,

```
5616185650518293 ;2 correct
3847439647293047 ;1 correct
5855462940810587 ;3 correct
9742855507068353 ;3 correct
4296849643607543 ;3 correct
3174248439465858 ;1 correct
4513559094146117 ;2 correct
7890971548908067 ;3 correct
8157356344118483 ;1 correct
2615250744386899 ;2 correct
8690095851526254 ;3 correct
6375711915077050 ;1 correct
6913859173121360 ;1 correct
6442889055042768 ;2 correct
2321386104303845 ;0 correct
2326509471271448 ;2 correct
5251583379644322 ;2 correct
1748270476758276 ;3 correct
4895722652190306 ;1 correct
3041631117224635 ;3 correct
1841236454324589 ;3 correct
2659862637316867 ;2 correct
```

Find the unique 16-digit secret sequence.

Step 1. Encode finding the correct sequence as a SAT problem. Use Boolean variables $x_{i,j}$ which are true if and only if at position i there is the digit j. The encoding consists of two parts: i) at each position there is exactly one digit; and ii) the correct number of digits from each line is matched. The encoding should *only* use these $x_{i,j}$ variables.

Step 2. Show that the correct sequence is unique. Which clause do you need to add to the encoding of Step 1?

Step 3. Reduce the size of the encoding by replacing the cardinality constraints from ii) in Step 1 by using the Sinz encoding.

CHAPTER

EIGHT

DEDUCTION FOR PROPOSITIONAL LOGIC

In the study of computational complexity, a *language* is a set of strings over some alphabet. For example, we can consider the language *PROP* consisting of all propositional formulas, the language *SAT* consisting of all satisfiable formulas, and the language *TAUT* consisting of all tautologies. We have seen that *SAT* and *TAUT* are both *decidable*, which is to say, there is are algorithms to decide membership in those sets. The P=NP question is precisely the question as to whether there a polynomial time algorithm for *SAT*, or, equivalently, for *TAUT*.

Still speaking in broad terms, a *proof system* for a language is a relation $P(d, x)$ between strings with the property for any x, x is in the language if and only if there is a d such that $P(d, x)$ holds. In this case, we say that d is a *proof of membership* for x. We typically require that checking a proof is easy, say, by requiring that $P(d, x)$ runs in polynomial time. This is often easy to do by putting a lot of information into d. NP is the class of languages that have a polynomial time proof system with the additional property that for every x in the language, there is a proof of membership d whose length is polynomially bounded in the length of x. The language *SAT* is in NP because there are short proofs of satisfiability, namely, the satisfying assignments.

When it comes to propositional logic, when we talk about proof systems, we generally mean a proof system for *TAUT*. In other words, a proof system for propositional logic is supposed to show that a formula is valid. It can therefore also be used to establish unsatisfiability, since a formula A is unsatisfiable if and only if $\neg A$ is valid.

Assuming we have a particular proof system in mind, we write $\vdash A$ to mean that there is a proof of A. Remember that we use $\models A$ to mean that A is valid. The property that $\vdash A$ implies $\models A$ is known as *soundness*, and the property that $\models A$ implies $\vdash A$ is known as *completeness*. We want a proof system for propositional logic to be sound and complete.

Given that the set of tautologies in propositional logic is decidable, why do we need a proof system? The complexity of the decision procedures provides one answer: as far as we know, deciding whether or not something is a tautology takes exponential time in the worst case. From a theoretical standpoint, it is not clear whether proof systems can do substantially better; the question as to whether there is a polynomial time polynomially-bounded proof system for propositional logic is equivalent to the question as to whether NP = coNP, which is an open question. But, in practice, checking a proof is usually much more efficient than determining that something is a tautology from scratch.

Another concern is reliability. Fast decision procedures for propositional logic are highly optimized and sometimes buggy. Most modern SAT solvers can be asked to output a proof to justify the claim that the input is unsatisfiable. Checking the output with an independent checker adds confidence that the claim is correct.

Yet another reason to be interested in formal notions of proof is that they provide more faithful models of informal proof, the process by which mathematicians establish that mathematical claims are true. Finally, when we turn to first-order logic, we will see that there is no decision procedure for validity. In fact, even fairly restricted versions of the question can be undecidable. In cases like that, the best we can do is search for proofs and counterexamples, with no guarantee that either will succeed in finite time. In other words, proof systems for first-order logic are essential.

The notation $\Gamma \vdash A$ is used to express that A is provable from a set of hypotheses Γ. The notation $\vdash A$ therefore abbreviates $\emptyset \vdash A$. In this more general setting, soundness says that if $\Gamma \vdash A$, then $\Gamma \models A$, and completeness says that if $\Gamma \models A$, then $\Gamma \vdash A$. If Γ is the finite set $\{B_1, \ldots, B_n\}$, then $\Gamma \models A$ is equivalent to $\models B_1 \wedge \cdots \wedge B_n \rightarrow A$. So, for many purposes, we can focus on provability and validity without hypotheses. If the set Γ is infinite, however, we cannot express $\Gamma \models A$ in those terms. For most of this chapter, we will not worry about infinite sets of hypotheses, since

mechanized reasoning generally has to work with finite representations. But we will discuss the case where Γ is infinite in Section 8.5.

When it comes to talking about formal proofs, the words *proof*, *deduction*, and *derivation* are often used interchangeably. The last two are sometimes useful to distinguish formal derivations from ordinary (informal) mathematical proofs.

8.1 Axiomatic systems

Historically, one way of describing a proof system for propositional logic is to give a list of axioms, like this one:

1. $A \to (B \to A)$
2. $(A \to (B \to C)) \to ((A \to B) \to (A \to C))$
3. $A \to (B \to A \wedge B)$
4. $A \wedge B \to A$
5. $A \wedge B \to B$
6. $A \to A \vee B$
7. $B \to A \vee B$
8. $(A \to C) \to ((B \to C) \to (A \vee B \to C))$
9. $\neg\neg A \to A$.

These are really axiom *schemas*, which is to say, we have an axiom for every choice of A, B, and C. The only rule of inference in the system is *modus ponens*, which is the rule "from A and $A \to B$ conclude B." A *proof* of a formula A from hypotheses Γ is a sequence of formula $C_1, \ldots, C_m$ such that every C_i is either:

- an axiom,
- a hypothesis, or
- consequence of two earlier formulas C_j and C_k using modus ponens.

This proof system is sound and complete. Proving soundness is straightforward: you only need to check that each axiom is valid and that modus ponens preserves truth. This enables us to show, by induction, that each line of a proof $C_1, \ldots, C_m$ of A from Γ is true under an assignment τ, assuming every formula in Γ is. Proving completeness requires more work. But axiomatic systems are no longer of much practical interest: they do not provide convenient means of modeling informal proofs, and they are not useful for automated reasoning or search. We will therefore set them aside and focus our attention on other types of proof systems.

8.2 A sequent calculus

Let Γ be a finite set of propositional formula in negation normal form. The next calculus we will consider is designed to prove that the disjunction of the formulas in Γ is valid, which is to say, for every truth assignment τ, at least one of the formula in Γ is true. If A is a formula, we write Γ, A instead of $\Gamma \cup \{A\}$. The rules are as follows:

$$\Gamma, p, \neg p$$

$$\frac{\Gamma, A \qquad \Gamma, B}{\Gamma, A \wedge B} \qquad \frac{\Gamma, A, B}{\Gamma, A \vee B}$$

The first rule says that either something in Γ is true, or p is true, or $\neg p$ is true. The second rule says that if either something in Γ is true or A is true, and either something in Γ is true or B is true, then either something in Γ is true or $A \wedge B$ is true. The third rule says that if either something in Γ is true or A is true or B is true, then either something in Γ is true

or $A \vee B$ is true. If we take these to be statements about truth values relative to some truth assignment τ, these rules are clearly sound with respect to the semantics.

The set Γ is called a *sequent*. (More specifically, it is called as *one-sided* sequent; we'll see two-sided sequents below.) A system of rules like this is therefore called a sequent calculus. As with axiomatic systems, we can think of a proof as a sequence of lines, but it is also common to represent proofs diagrammatically, as trees whose nodes are labeled by sequents. The following example provides a proof of the NNF equivalent of $A \wedge (B \vee C) \rightarrow (A \wedge B) \vee (A \wedge C)$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\neg A, \neg B \wedge \neg C, A \wedge B, A} \quad \dfrac{\dfrac{\dfrac{}{\neg A, \neg B \wedge \neg C, A, C} \quad \dfrac{\dfrac{}{\neg A, \neg B, B, C} \quad \dfrac{}{\neg A, \neg C, B, C}}{\neg A, \neg B \wedge \neg C, B, C}}{\neg A, \neg B \wedge \neg C, A \wedge B, C}}{\neg A, \neg B \wedge \neg C, A \wedge B, A \wedge C}}{\neg A, \neg B \wedge \neg C, (A \wedge B) \vee (A \wedge C)}}{\neg A \vee (\neg B \wedge \neg C), (A \wedge B) \vee (A \wedge C)}}{(\neg A \vee (\neg B \wedge \neg C)) \vee ((A \wedge B) \vee (A \wedge C))}$$

It is probably easiest to read this from the bottom up.

Remember that saying that a sequent Γ is *valid* in our semantics means that for every truth assignment τ, we have $[\![A]\!]_\tau = \top$ for some A in Γ.

Theorem

The sequent calculus presented above is sound and complete. In other words, a sequent Γ is provable if and only if it is valid.

Proof

The soundness direction is easy. Suppose there is a proof of Γ. Let τ be any truth assignment. We have already noted that each rule is sound, which is to say, if the premise or premises are true under an assignment, then so is the conclusion. By induction, we have that every sequent in the proof is true under assignment τ.

Proving completeness is usually trickier. In this case, we can use the fact that the rules of the calculus are bidirectional: any truth assignment that refutes the conclusion of a rule has to refute one of the premises. We can also use the fact that reading each rule backward decreases the number of binary connectives.

We prove the following by induction on the number of binary connectives: for every sequent Γ, either Γ is provable, or there is a truth assignment that makes every formula in Γ false. If Γ has no binary connectives, then it is a set of literals. If Γ contains a complementary pair of literals P and $\neg P$, it is an axiom, and otherwise there is an assignment that makes it false. For the inductive step, Γ must have a $\wedge$ or a $\vee$. Applying the corresponding rule, we have either the premises are valid or there is a counterexample to one of them. In the first case, Γ is valid, and in the second case, there is a counterexample to the conclusion.

Notice that this gives us yet another decision procedure for propositional logic: start with a sequent Γ (which can consist of a single formula, if you want), and apply the rules backward. If you reach an axiom on each leaf, you have a proof. If one branch fails to terminate with an axiom, reading off a counterexample to the leaf yields a counterexample to Γ. This is an important idea in automated reasoning, namely, it is desirable to search for a proof in such a way that failure implies the existence of a counterexample.

What we have described is more precisely a *cut-free* sequent calculus. It is also sound to add the cut rule:

$$\dfrac{\Gamma, A \qquad \Gamma, \sim A}{\Gamma}$$

Here $\sim A$ is the negation operator for negation-normal form formulas, which switches $\wedge$ and $\vee$ and switches positive and negative literals. Proofs with cuts can be more efficient proofs without them, but we have seen that the calculus is complete

without them.

The cut-free sequent calculus is closely related to *tableau* proof systems that are also commonly used in automated reasoning. In the sequent proof above, there is a lot of needless repetition of formulas; tableau representations do a better job of recording only what changes as we go up the tree. Another difference is that tableau proof systems usually don't require that the formulas are in negation normal form. Rather, the rules of a tableau system correspond to the sequent rules for the negation-normal-form equivalents. Since $A \to B$ is equivalent to $\neg A \vee B$, this requires changing A to $\neg A$ as we move up the tree. To avoid introducing new connectives, tableau systems often annotate formulas with emph{polarities}, so that A^+ represents A and A^- represents $\neg A$. The most jarring difference between sequent calculi and tableau systems is that the latter are often described in terms of search for a satisfying assignment rather than searching for a proof. For example, the rule in the sequent calculus that says

> to find a proof of $A \wedge B$, split and find a proof of A and a proof of B

becomes

> to find a satisfying assignment to $A \vee B$, split and find a satisfying assignment to A or a satisfying assignment to B.

The automated reasoning community is split between people who like to think in terms of searching for proofs and people who like to think in terms of searching for models. It is therefore important to learn how to speak both languages, and to be able to translate between them on the fly.

8.3 Resolution

Given that the sequent calculus implicitly corresponds to a decision procedure for propositional logic, it is natural to ask whether there is a proof system that corresponds more closely to DPLL, the decision procedure that was the focus of Chapter 6. We now describe such a system.

A *resolution* proof is designed to *refute* a CNF formula, that is, to prove that it is unsatisfiable. Let Γ be a CNF formula, represented as a set of clauses. As in Section 6.3, we can assume that none of the clauses contain repeated literals or a complementary pair, and we can think of each clause as a set of literals. If C is a clause and ℓ is a literal, we write C, ℓ for $C \vee \ell$. The *resolution rule* derives a new clause from an old one:

$$\frac{C, p \qquad D, \neg p}{C \vee D}$$

The rule says that if either C or p is true, and either D or $\neg p$ is true, then $C \vee D$ has to be true. A *resolution proof* of a clause C from a set of clauses Γ is a sequence of steps (or a labelled tree) that obtains C from Γ using instances of the resolution rule. A *resolution refutation* of Γ is a resolution proof of the empty clause from Γ.

Theorem

A CNF formula Γ has a resolution refutation if and only if it is unsatisfiable.

Resolution can therefore be understood as a proof system for propositional logic in the following way: given any formula A, put $\neg A$ in CNF, and look for a resolution refutation. Such a refutation is a proof of A. The theorem above says that this system is sound and complete: A is valid if and only if $\neg A$ is unsatisfiable, which happens if and only if there is a refutation of $\neg A$.

Proof

Soundness follows straightforwardly from the fact that the resolution rule preserves truth under any truth assignment, while the empty clause is unsatisfiable.

To prove completeness, we use induction on the number of propositional variables to show that if Γ is unsatisfiable, there is a resolution refutation of Γ. If there are no variables, the fact that Γ is unsatisfiable means that it must be the set consisting of the empty clause, and we are done.

In the induction step, let P be any variable that occurs in Γ. If Γ is unsatisfiable, then so are $[\![\Gamma]\!]_{[P\mapsto\top]}$ and $[\![\Gamma]\!]_{[P\mapsto\bot]}$. By the inductive hypothesis, both of these are refutable.

Remember the relationship between $[\![\Gamma]\!]_{[P\mapsto\top]}$ and Γ: in the former, we remove all the clauses that include P and delete $\neg P$ from the remaining clauses. So a resolution refutation of the empty clause from $[\![\Gamma]\!]_{[P\mapsto\top]}$ uses only the clauses of $[\![\Gamma]\!]$ that don't contain P, possibly with $\neg P$ removed. Restoring $\neg P$ to all the initial clauses yields either a proof of the empty clause or a proof of $\neg P$.

In the first case, we have a proof of the empty clause from Γ, and we are done. Otherwise, applying the inductive hypotheses to $[\![\Gamma]\!]_{[\neg P\mapsto\bot]}$ and repeating the previous argument, we obtain either a proof of the empty clause or a proof of P. Once again, in the first case, we are done. Otherwise, we apply the resolution rule to the proof of P and the proof of $\neg P$, and we have a proof of the empty clause.

We can once again view the completeness proof as a decision procedure in disguise. In fact, the strategy of picking a variable and trying to refute $[\![\Gamma]\!]_{[P\mapsto\top]}$ and $[\![\Gamma]\!]_{[P\mapsto\bot]}$ simultaneously is exactly the splitting rule of DPLL, formulated in terms of demonstrating unsatisfiability rather than searching for a satisfying assignment. We can formulate the theorem above in more constructive terms as follows:

Theorem

For any CNF formula Γ, either Γ is satisfiable or there is a resolution refutation.

Remember that at any point in the DPLL search, we have a partial assignment τ that we are trying to extend to a satisfying assignment to $[\![\Gamma]\!]_\tau$. If we fail, we then have to give up on τ, backtrack, and try another path. To extract either a satisfying assignment or a resolution proof from the result, it suffices to show the following constructively:

> For any partial truth assignment τ, either $[\![\Gamma]\!]_\tau$ is satisfiable or there is a resolution proof of a clause C from Γ such that $[\![C]\!]_\tau = \bot$.

This yields the desired conclusion when τ is the empty assignment, since the only clause that evaluates to $\bot$ under the empty assignment is the empty clause.

We will sketch an explanation of how to read off the information above from the DPLL search, and we will leave it as an exercise for you to fill in the details. Remember that there are three steps that are interleaved in DPLL:

1. splitting on a variable p
2. unit propagation
3. removing pure literals

Unit propagation can be viewed as a special case of the splitting rule: If a CNF formula contains a unit clause with literal ℓ, then splitting on ℓ fails immediately on one branch, and the other branch corresponds to the result of applying unit propagation.

Reasoning about the pure literal rule is more subtle. Suppose ℓ is pure in $[\![\Gamma]\!]_\tau$, and consider the DPLL search starting from $[\![\Gamma]\!]_{\tau[\ell\mapsto\top]}$, which is a subset of $[\![\Gamma]\!]_\tau$. If DPLL finds a satisfying assignment to $[\![\Gamma]\!]_{\tau[\ell\mapsto\top]}$, it can be extended to a satisfying assignment to $[\![\Gamma]\!]_\tau$ by mapping ℓ to $\top$. On the other hand, for any clause C such that $[\![C]\!]_{\tau[\ell\mapsto\top]} = \bot$, we have $[\![C]\!]_\tau = \bot$, because ℓ is pure in all the clauses of $[\![\Gamma]\!]_\tau$.

So we only have to deal with the splitting rule. A satisfying assignment for either branch results in a satisfying assignment for Γ, we only only have to prove the following:

If there are a resolution proof of a clause C from Γ such that $[\![C]\!]_{\tau[P\mapsto\top]} = \bot$ and a resolution proof of a clause D from Γ such that $[\![D]\!]_{\tau[P\mapsto\bot]} = \bot$, then there is a resolution proof of a clause E from Γ such that $[\![E]\!]_\tau = \bot$.

We leave the proof of this fact to you.

8.4 Natural deduction

We now consider a type of deductive system that was introduced by Gerhard Gentzen in the 1930s, known as *natural deduction*. As the name suggests, the system was designed to model the way someone might carry out an informal logical argument. As a result, the system is not particularly good for automated reasoning and proof search, though it might be a good choice if the goal is to find a human-readable proof in the end. The main interest, rather, is that it provides a nice framework for representing informal arguments. Of all the systems we consider in this section, this one is the closest to the foundation system of logic that is built in to Lean.

In natural deduction, the goal is to derive sequents of the form $\Gamma \vdash A$, where Γ is a finite set of formulas and A is a formula. The interpretation of such a sequent is, of course, that A follows from the hypotheses in Γ. Notice that we have overloaded the symbol $\vdash$, which is ordinarily used to express the provability relation. The two are clearly related: Γ proves A in the ordinary sense is now interpreted as saying that there is a finite subset $\Gamma' \subseteq \Gamma$ such that the sequent $\Gamma \vdash A$ is derivable in natural deduction. Sometimes people use other notation for sequents, like $\Gamma \Rightarrow A$. But Lean also uses the $\vdash$ symbol for sequents, so we will stick with that.

We write "Γ, A" for $\Gamma \cup \{A\}$ to represent the hypotheses in Γ together with the additional hypothesis A. The first rule is trivial: we always have

$$\cdot \quad \Gamma, A \vdash A$$

This says that A follows from any list of assumptions that includes A. Most of the other connectives include *introduction rules*, which allow us to *introduce* the connective into a proof, and *elimination rules*, that tell us how to use them. For example, the rules for $\wedge$ are as follows:

$$\frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \wedge B} \qquad \frac{\Gamma \vdash A_0 \wedge A_1}{\Gamma \vdash A_i}$$

The rules for implication are as follows:

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \to B} \qquad \frac{\Gamma \vdash A \to B \qquad \Gamma \vdash A}{\Gamma \vdash B}$$

Notice that in the introduction rule, to prove $A \to B$, we temporarily assume A and show that B follows. The rules for disjunction, which codify proof by cases, are as follows:

$$\frac{\Gamma \vdash A_i}{\Gamma \vdash A_0 \vee A_1} \qquad \frac{\Gamma \vdash A \vee B \qquad \Gamma, A \vdash C \qquad \Gamma, B \vdash C}{\Gamma \vdash C}$$

For classical logic, we also add the following principle of proof by contradiction:

$$\frac{\Gamma, \neg A \vdash \bot}{\Gamma \vdash A}$$

These rules cover a complete set of connectives, since we can define $\neg A$ to be $A \to \bot$, define $\top$ to be $\neg\bot$, and define $A \leftrightarrow B$ to be $(A \to B) \wedge (B \to A)$. You should think about what the natural rules for these connectives would be if we were to include them in the calculus.

As an example, here is a proof of $A \to (B \to A \wedge B)$:

$$\dfrac{\dfrac{\dfrac{A, B \vdash A \qquad A, B \vdash B}{A, B \vdash A \wedge B}}{A \vdash B \to A \wedge B}}{\vdash A \to (B \to A \wedge B)}$$

And here is a proof of $A \wedge B \to B \wedge A$:

$$\dfrac{\dfrac{\dfrac{}{A \wedge B \vdash A \wedge B}}{A \wedge B \vdash B} \quad \dfrac{\dfrac{}{A \wedge B \vdash A \wedge B}}{A \wedge B \vdash A}}{\dfrac{A \wedge B \vdash B \wedge A}{\vdash A \wedge B \to B \wedge A}}$$

As with the sequent calculus, there are more efficient ways of representing natural deduction proofs that only show the conclusion at each node and leave the hypotheses implicit. This avoids having to repeat a long list of hypotheses at every node. There are also presentations of sequent calculi for classical logic that use two-sided sequents, as we did for natural deduction. The most effective approach is to use sequents of the form $\Gamma \vdash \Delta$, where Γ and Δ are finite sets and we interpret the sequent as saying that if all the hypotheses in Γ are true, then at least one of the formulas in Δ is true. This is aligned with the annotations of positive and negative formulas that one sometimes finds in tableau calculi.

Theorem

Natural deduction is sound and complete for classical propositional logic. In other words, a sequent $\Gamma \vdash A$ is derivable if and only if $\Gamma \models A$.

We will not take the time to prove this here. One way to prove it is to show that natural deduction can simulate another calculus, like the sequent calculus, for which a completeness proof is easier.

8.5 Compactness

In automated reasoning, when we write $\Gamma \models A$ to express that A is entailed by hypotheses in Γ, we generally have the in mind case where Γ is finite. But the definition makes sense when Γ is infinite. The same is true in the when we talk about provability of A from a set of hypotheses Γ. Now, in the case of provability, it is clear that for any set Γ, finite or infinite, $\Gamma \vdash A$ if and only if there is a finite subset $\Gamma' \subseteq \Gamma$ such that $\Gamma \vdash A$. The corresponding fact about the entailment relation is also true:

Theorem (the compactness theorem for propositional logic)

For any set of propositional formulas Γ, $\Gamma \models A$ if and only if there is a finite subset $\Gamma' \subseteq \Gamma$ such that $\Gamma' \models A$.

In this chapter, we focused on the soundness and completeness theorems in the case where Γ is finite. But soundness carries over straightforwardly to the infinite case, and it is possible to prove completeness for arbitrary Γ as well. You can check that compactness follows from these stronger versions of soundness and completeness, given the fact that any proof can use only a finite number of hypotheses. Conversely, the stronger versions of the completeness theorem follows from the weaker one using the compactness theorem.

As an example of an application of the compactness theorem, imagine a finite set of oriented square tiles S, all the same size, where "oriented" means that each has is a distinguished top edge. Suppose each edge of each tile is labeled with finitely many colors. A *tiling of the plane* with tiles from S is an ordinary arrangement of (copies of) tiles in S in an infinite square grid, like the squares on an infinite sheet of graph paper, such that adjacent edges have the same color.

Theorem

Suppose that for every natural number n, there is an $n \times n$ tiling with tiles from S. Then there is a tiling of the entire plane with tiles from S.

You should think about how this follows from the compactness theorem.

CHAPTER

NINE

PROPOSITIONAL LOGIC IN LEAN

In Chapter 3, we considered the use of Lean as a programming language, and in Chapter 5 we saw that you can use Lean to define data types for things like propositional formulas and truth assignments, and thereby implement algorithms that act on these objects.

In this chapter, we will show how to represent propositional formulas directly in Lean's underlying foundation. In this sense, we are using Lean's foundation as an *object language* rather than a *metalanguage* for logic. To clarify the distinction, imagine implementing one programming language, such as Lisp, in another programming language, like C++. In this scenario, we can characterize Lisp as being the object language, that is, the one that is being implemented, and C++ as the metalanguage, the one that is carrying out the implementation. What we did in Chapter 5 is similar: we are used one logical system, Lean, to implement another one, propositional logic. One goal of this chapter is to clarify the sense in which Lean itself is a logical system, which is to say, its language can be used to state mathematical theorems and prove them. Propositional logic is only a starting point. In chapters to come we will see that Lean's logical foundation is powerful and expressive enough to carry out almost any mathematical argument.

This means that Lean's logical foundation serves as both a programming language and a mathematical language. Combining the two brings a number of benefits:

- It allows us to specify the behavior of computer programs in the same language that we write them.
- It allows us to prove, rigorously, that our computer programs are correct, which is to say, that they meet their specifications.
- It allows us to enforce preconditions on our programs. For example, we can write functions whose input is required to be a positive integer, a requirement that is enforced statically, at compile time. Compiler optimizations can make use of this knowledge.
- It allows us to compute with objects in our mathematical libraries.
- It gives us ways of using computation in mathematical proofs.

Although we will not discuss it in this course, Lean also serves as its own *metaprogramming language*, which means that we can use Lean to develop automation that can help us construct programs and proofs. In that way, Lean becomes a self-extending system, meaning that we can improve its support for programming and theorem proving using the system itself.

9.1 Implication

In Lean, we can declare variables that range over propositions, and then use them to build more complicated propositions.

```
variable (p q r s : Prop)

#check True
#check False
#check p ∧ q
#check p ∨ q
#check p → q
#check p ↔ q
#check ¬ p
```

Hovering over the symbols will give you options for typing them. Using *\and*, *\or*, *\to*, *\iff*, and *\not* will work.

In Lean, if *p* has type *Prop*, then a term *t* of type *p* is a proof of *p*. Lean's proof language is essentially the same as its programming language, which means that we can write proofs the same way we write programs. Instead of using the identifier *def*, it is conventional to use the word *theorem* to name the proof of a proposition.

```
theorem foo : p → q → p ∧ q :=
  fun hp hq => And.intro hp hq

theorem bar : p ∧ q → q ∧ p :=
  fun ⟨hp, hq⟩ => ⟨hq, hp⟩
```

The expression after the := is sometimes called a *proof term*.

You can read more about writing proof terms in Theorem Proving in Lean 4. In this chapter, we will describe an alternative method of writing proofs, using *tactics*. A tactic proof is essentially a piece of metacode, a list of instructions that tells Lean how to construct the proof term. In Lean, we enter tactic mode with the keyword *by*.

```
example : p → p := by
  intro h
  apply h
```

Here the keyword *example* simply introduces an unnamed theorem (or definition). The *intro* tactic introduces the hypothesis and names it *h*. If you put the cursor on that line and check the *Lean infoview* in the editor, you will see the current state of the proof, that looks like this:

```
p q r s : Prop
h : p
⊢ p
```

Think of this as a *sequent*, as described in the last chapter: it expresses that the current goal is to prove *p*, using the assumption *h : p*. In contrast to the sequent calculus we considered for natural deduction, here the hypothesis that *p* holds is labeled with the identifier *h*. You can use any identifier as a label. Also, in contrast with the sequent calculus for natural deduction, the variables *p*, *q*, *r*, and *s* are included, together with their type. Later on we will see that other kinds of data, like natural numbers, can be included as variables in the context. Notice also that in Lean, hypotheses like α *: Type*, *p : Prop*, *n : Nat*, and *h : p* are all possible, so the context can include variables ranging over data types, propositions, elements of the data types, and proofs (or assumptions) that the propositions hold.

In *intro* tactic implements the implication introduction rule of the natural deduction calculus. The *apply* tactic in the proof applies the hypothesis *h* to solve the goal. In this case, since *h* solves the goal exactly, we could write *exact h* instead. We could also write *assumption*, which tells Lean to look for any assumption in the context that solves the goal. This corresponds to the assumption rule in the natural deduction calculus.

The *apply* tactic can also be used to apply an implication.

```
example (h1 : p → q) (h2 : p) : q := by
  apply h1
  exact h2
```

In this example, applying $p \to q$ to the goal reduces the task of proving q to the task of proving p. Lean also provides a means of reasoning forward from hypotheses, using the *have* tactic:

```
example (h1 : p → q) (h2 : p) : q := by
  have h3 : q := h1 h2
  exact h3
```

Here the first line of the proof states an intermediate goal of proving q, which is achieved by applying *h1* to *h2*. The result is named *h3*, which we can then use in the next line. We can leave out the information that *h3* is a proof of q, because Lean can figure that out:

```
example (h1 : p → q) (h2 : p) : q := by
  have h3 := h1 h2
  exact h3
```

Defining *h3* to be *h1 h2* and then applying it is just the same as using *exact h1 h2* directly to solve the goal, and that, in turn, amounts to constructing the proof *h1 h2*.

```
example (h1 : p → q) (h2 : p) : q := by
  exact h1 h2

example (h1 : p → q) (h2 : p) : q := h1 h2
```

9.2 Conjunction

There is a theorem in the library called *And.intro* that encodes the and-introduction rule in the natural deduction calculus, telling us that we can prove $p \wedge q$ by proving each of p and q in turn.

```
theorem and_example : p → q → p ∧ q := by
  intro hp
  intro hq
  apply And.intro
  . exact hp
  . exact hq

#print and_example

example : p → q → p ∧ q :=
  fun hp hq => And.intro hp hq
```

After applying *And.intro*, the goal is split into two goals, one which requires us to prove p, and the other which requires us to prove q. Each of these is accomplished with the *exact* tactic. As you step through the proof, you can see the state change. The periods in each case serve to *focus* on the next goal, so that only that goal is visible in that part of the proof. Strictly speaking, they are not necessary; try deleting them to confirm that the proof still goes through. Still, they provide a nice way of making the structure of the proof manifest, and this, in turn, results in more robust proof scripts. The *#print* statement shows that the tactics have the effect of constructing a proof term. The subsequent example provides the proof term explicitly. In fact, the term makes it clear that introducing *hp* and *hq* and then applying *And.intro* is an unnecessary detour. The theorem we have proved is exactly *And.intro*.

```
example : p → q → p ∧ q := And.intro
```

At this point, there are two tricks that are worth mentioning. First, at any point in a proof, you can solve the current goal with the *admit* tactic. You can prove any theorem at all using *admit*. It's cheating, and the squiggly line the editor puts underneath the word tells you as much. But it is often a useful device when writing proofs, because it means you can temporarily close a goal to work on others, and then come back to it.

The other trick is to use the *done* tactic. The *done* tactic doesn't do anything; it just declares that the proof is over. If the proof isn't over, Lean gives you an error message, and the error message tells you exactly the remaining goals. So this gives you a way to mark the end of a proof in progress, so that you can easily monitor what is left to be done.

The natural way to *use* a conjunction $h : p \wedge q$ in a hypothesis is to split it to the hypotheses $hp : p$ and $hq : q$. This rule is commonly used in sequent calculi. There are various ways to do it. One is to use a destructuring *have*:

```
example : p ∧ q → q ∧ p := by
  intro h
  have ⟨hp, hq⟩ := h
  apply And.intro
  . exact hq
  . exact hp
```

Here the angle brackets are Lean's *anonymous constructor* notation. You can type them with \< and \>. In this case, the *have* command tells Lean to match against whatever constructor matches the proposition, in this case, *And.intro*. Another option is to use the *cases* tactic, which matches the cases against the constructor:

```
example : p ∧ q → q ∧ p := by
  intro h
  cases h with
  | intro hp hq =>
    exact And.intro hq hp

example : p ∧ q → q ∧ p := by
  intro h
  cases h
  case intro hp hq =>
    exact And.intro hq hp
```

These are two slightly different formulations of the same proof. Notice that in the second variation we drop the keyword *with* and use the *case* tactic instead of the vertical bar.

Here are some examples of using the pattern-matching *have* with the ordinary one.

```
example (h1 : p → q) (h2 : p) : q := by
  have h3 : q := h1 h2
  exact h3
```

In the first example, the second *have* claims an auxiliary statement, *r*, and proves it with the term *h2 hq*. The third example shows that we can also prove the intermediate claim in tactic mode, using the keyword *by*.

The *have* statement is similar to a *let* statement that we have already seen when we used Lean as a programming language. In fact, using a *let* also works:

```
example (h1 : p ∧ q) (h2 : q → r) : p ∧ r := by
  let ⟨hp, hq⟩ := h1
  let hr : r := by
    apply h2
    exact hq
  exact And.intro hp hr
```

(continues on next page)

(continued from previous page)

```
example (h1 : p ∧ q) (h2 : q → r) : p ∧ r := by
  let ⟨hp, hq⟩ := h1
  exact And.intro hp (h2 hq)
```

With both *have* and *let*, we can take the proofs to abbreviate the result of substituting the proof of *r* where we used *hr*, as in the last example above. There is a subtle difference between *have* and *let*: while the data in the *let* statement is available in the body of the statement after the *let*, a *have* only records the existence of an expression of the corresponding type. When we use a *have*, typically we only care that the corresponding fact has been proved, and the specifics of the proof are irrelevant. For that reason, *have* is more appropriate for proofs.

9.3 Disjunction

The theorem *Or.inr* in the library corresponds to the right introduction rule, which proves $p \vee q$ from p. The left introduction rule is given by *Or.inl*. We can carry out a proof by cases using either the theorem *Or.elim* or the *cases* tactic.

```
example : p ∨ q → q ∨ p := by
  intro h
  apply Or.elim h
  . intro hp
    apply Or.inr
    exact hp
  . intro hq
    apply Or.inl
    exact hq

example : p ∨ q → q ∨ p := by
  intro h
  cases h with
  | inl hp =>
    exact Or.inr hp
  | inr hq =>
    exact Or.inl hq

example : p ∨ q → q ∨ p := by
  intro h
  cases h
  case inl hp =>
    exact Or.inr hp
  case inr hq =>
    exact Or.inl hq
```

Notice that in contrast to casing on a conjunction, which results in one new goal and two new hypotheses, casing on a disjunction results in two new goals, each with one new hypothesis.

9.4 Negation

In Lean, $\neg p$ is defined to be $p \to \mathit{False}$. For most purposes, the two expressions are interchangeable. This means that you can prove $\neg p$ by assuming p and deriving *False*, and if you have $hp : p$ and $hnp : \neg p$, then the result $hnp\ hp$ of applying the first to the second is a proof of *False*. The following illustrates these ideas in action.

```
example (h : p → q) : ¬ q → ¬ p := by
  intro hnq hp
  apply hnq
  apply h
  apply hp
```

Here, *intro hnp hp* is equivalent to writing *intro hnp* followed by *intro hp*. Lean provides other means of dealing with negations. For example, the *contradiction* tactic finishes off a goal whenever the context contains a statement and its negation.

```
example (h1 : p) (h2 : ¬ p) : q := by
  contradiction

example (h1 : p ∨ q) (h2 : ¬ p) : q := by
  apply Or.elim h1
  . intro hp
    contradiction
  . intro hq
    exact hq
```

In a similar way, the theorem *absurd* shows that anything follows from a statement and its negation. The first and second examples below are really the same: writing *by exact t* simply constructs the proof *t*.

```
example (h1 : p) (h2 : ¬ p) : q := by
  exact absurd h1 h2

example (h1 : p) (h2 : ¬ p) : q := absurd h1 h2

example (h1 : p ∨ q) (h2 : ¬ p) : q := by
  apply Or.elim h1
  . intro hp
    exact absurd hp h2
  . intro hq
    exact hq
```

The *contradiction* tactic also works if the context contains *False*. Alternatively, casing on $h : \mathit{False}$ finishes off a proof. Intuitively, there are no cases!

```
example (h : False) : p := by
  contradiction

example (h : False) : p := by
  cases h
```

The principles we have used before all fall within what is known as *intuitionistic* logic. Many mathematical arguments require *classical* logic, which is embodied in the *law of the excluded middle*. In Lean, if p is a proposition, *em p* is the principal $p \vee \neg p$.

```
example (h1 : p → q) : ¬ p ∨ q := by
  apply Or.elim (Classical.em p)
```

(continues on next page)

(continued from previous page)

```
  . intro hp
    apply Or.inr
    apply h1
    exact hp
  . intro hnp
    apply Or.inl
    exact hnp

example : ¬ ¬ p → p := by
  intro hnnp
  apply Or.elim (Classical.em p)
  . intro hp; exact hp
  . intro hnp
    contradiction
```

The following illustrate two other formulations of classical logic, allowing proof by cases and proof by contradiction.

```
example : ¬ ¬ p → p := by
  intro hnnp
  apply Classical.byCases
  . intro (hp : p)
    exact hp
  . intro (hnp : ¬ p)
    exact absurd hnp hnnp

example : ¬ ¬ p → p := by
  intro hnnp
  apply Classical.byContradiction
  intro (hnp : ¬ p)
  exact hnnp hnp
```

9.5 Miscellany

The rules for bi-implication are illustrated by the next two examples.

```
example (h1 : p ↔ q) (h2 : p) : q := by
  have ⟨h3, h4⟩ := h1
  apply h3
  exact h2

example (h1 : p ↔ q) : q ↔ p := by
  have ⟨h2, h3⟩ := h1
  apply Iff.intro
  . exact h3
  . exact h2
```

It is worth keeping in mind that tactics serve to construct proof terms, and that you can often be more concise by writing the proof terms yourself, as in the following examples.

```
example : p ∨ q → q ∨ p :=
  fun h => Or.elim h (fun hp => Or.inr hp) (fun hq => Or.inl hq)

example (h1 : p → q) : ¬ q → ¬ p :=
  fun hnq hp => hnq (h1 hp)
```

You should think about these examples and how they work. One strategy for learning how to write proof terms is to name the theorems that you prove with tactics, and then use the *#print* command to print them out.

CHAPTER

TEN

FIRST-ORDER LOGIC

Consider the statement "Bill and Tracy are married and all their children are smart." Propositional logic will let us model this statement as a conjunction, but it doesn't give us the means to model the fact that marriage is a relationship between two people, a relationship that in this case is claimed to hold between Bill and Tracy. It doesn't allow us to express the fact that if a person, X, is married to another person, Y, then Y is married to X. It also doesn't allow us to model the fact that the second conjunct quantifies over children, the fact that being a child of someone is an asymmetric relationship, or the fact that being smart is a property that someone may or may not have.

First-order logic will let us do all these things. In many ways, the syntax and semantics of first-order logic is similar to the syntax and semantics of propositional logic. One difference is that now we need two categories of expressions, *terms* and *formulas*. Terms name things in the intended interpretation (in the example above, people), whereas formulas say things about those objects. But in each case, the set of expressions is defined syntactically, and we use recursive definitions to specify how to evaluate them for a given interpretation, just as we did for propositional logic.

But there is a sense in which propositional logic and first-order logic are worlds apart. We have seen that for any given propositional formula, we can specify an interpretation by assigning truth values to its finitely many variables. In contrast, there are first-order formulas that are satisfiable, but only when interpreted with an infinite domain of objects. We also saw that the method of writing out a truth table provides an easy (though inefficient) decision procedure for propositional logic, and that a propositional formula is provable if and only if it is valid. This means that the question of provability for propositional logic is also decidable. In contrast, the question of provability for first-order logic is equivalent to the halting problem.

Even worse, the question of the *truth* of a first-order sentence in an intended interpretation is often even more undecidable than the halting problem. For example, questions as to the truth of a first-order statement about the natural numbers in a vocabulary that includes only basic arithmetic is undecidable, even giving an oracle for the halting problem, an oracle for the halting problem relative to the halting problem, or any finite iteration of that sort.

What is a poor computer scientist to do? We will see that there are at least two avenues we can pursue. The first thing we can do is develop decision procedures for fragments of first-order logic, or restricted interpretations of first-order logic. In particular, we will consider procedures for equational reasoning and procedures for reasoning about linear arithmetic on the real numbers. Such procedures are implemented by contemporary *SMT solvers*, on top of SAT-solving methods. The other thing we can do is develop means of searching for proofs from axioms, in such a way that we are guaranteed to find one if such a proof exists, even though the search may not terminate if there is none. This brings us to the domain of first-order theorem proving, which we will also explore.

10.1 Syntax

To specify a first-order language, L, we start by specifying some constant symbols, some function symbols, and some relation symbols. Each of the function and relation symbols comes with an associated *arity*, namely, a specification of the number of arguments. For example, to design a language to reason about the integers, we might choose to have constants 0 and 1, a binary function $f(x, y)$ to represent addition, a binary function $g(x, y)$ to represent multiplication, a unary function $h(x)$ to represent negation, and a binary relation $R(x, y)$ to represent the relation $x < y$.

The set of *terms* is defined to be the set of all things we can obtain using variables, constants, and function symbols. For example, $f(g(x, 1), h(y))$ is a term. The following definition makes this more precise:

Definition

The set of terms of the language L is generated inductively as follows:

- Each variable $x, y, z, \ldots$ is a term.
- Each constant symbol of L is a term.
- If f is any n-ary function symbol of L and $t_1, t_2, \ldots, t_n$ are terms of L, then $f(t_1, t_2, \ldots, t_n)$ is a term.

Keep in mind that a term is supposed to name an object, given an interpretation of the symbols and an assignment to the free variables. For example, with the interpretation above, $f(g(x, 1), h(y))$ denotes the integer 4 when x is assigned to 6 and y is assigned to 2. The semantics we present below makes this precise. In contrast, a formula is supposed to make a statement, again given an interpretation of the symbols and an assignment to the free variables.

Definition

The set of formulas of the language L is generated inductively as follows:

- If R is any n-ary relation symbol of L and $t_1, t_2, \ldots, t_n$ are terms of L, then $R(t_1, t_2, \ldots, t_n)$ is a formula.
- If s and t are terms, then $s = t$ is a formula.
- $\top$ and $\bot$ are formulas.
- If A and B are formulas, so are $\neg A$, $A \wedge B$, $A \vee B$, $A \to B$, and $A \leftrightarrow B$.
- If A is a formula and x is any variable, then $\forall x.\, A$ and $\exists x.\, A$.

Most of the clauses should be familiar. In the last one, $\forall x.\, A$ ("for all x, A") expresses that A holds of every element in the intended interpretation, and $\exists x.\, A$ ("there exists an x such that A") expresses that A holds of some element in the intended interpretation. Once again, the semantics we present below makes this precise. It is sometimes useful to think of constants as 0-ary function symbols, that is, function symbols that don't take any arguments. In a similar way, a 0-ary relation symbol is just a propositional variable.

As we did in Chapter 4, we can define notions of depth and complexity for terms and formulas. We can say what it means for a term to be a subterm of another term, and say what it means for a formula to be a subformula of another formula. We can also say what it means to substitute a term t for a variable x in another term s, which we denote $s[t/x]$, and what it means to substitute a term t for a variable x in a formula A, which we denote $A[t/x]$. These operations are of central importance in first-order logic.

We will later need a notion of *simultaneous substitution*, which replaces multiple variables at once. In this context, a *substitution* σ is a mapping from a set of variables to terms in a language. (If we want, we can think of σ as a mapping from the set of *all* variables to terms which assigns the remaining variables to themselves.) Given a substitution σ and a

term t, the substitution $\sigma\, t$ is defined recursively as follows:

$$\sigma\, x = \sigma(x)$$
$$\sigma\, f(t_1, \ldots, t_n) = f(\sigma\, t_1, \ldots, \sigma\, t_n)$$

In other words, σ tells us what to plug in for the variables, and otherwise the substitution leaves the function symbols and constants the same. If A is a formula, the definition of the substitution $\sigma\, A$ is similar, though it is slightly complicated by the fact that we might have to rename the bound variables when we carry out the substitution. Implementations of first-order logic have to deal with that appropriately, but we will not worry about the details here.

The set of variables that occur in a term can be defined by recursion on terms. Consider the formula $\exists z.\, (x < z \wedge z < y)$. Intuitively, this says "there is something between x and y." The variable z is said to be *bound* in this formula, whereas x and y are said to be *free*. The set of free variables of a formula can be defined by recursion on formulas. Intuitively, the formula above says the same thing as $\exists w.\, (x < w \wedge w < y)$. Logicians and computer scientists often "identify" formulas up to renaming of their bound variables, which is to say, they consider these formulas to be the same and rename bound variables freely.

You should take care to rename bound variables when carrying out substitution to avoid capture. For example, the formula $\forall x.\, \exists y.\, y > x$ says that for every x there is some number greater than it. This is clearly true when we interpret the statement in the integers, but the statement is patently false when we substitute y for x in $\exists y.\, y > x$. If we rename the bound variable y to z, we can substitute y for x without problems. You should *never* rename a free variable, however. Saying that there is something bigger than x is not the same as saying there is something bigger than z. A formula without free variables is called a *sentence*.

Spelling out the nuances of bound variables precisely is one of the most annoying theoretical chores in mathematical logic and computer science. It can be done, but we will gloss over the details, and rely on your intuition and common sense to get by.

10.2 Using first-order logic

Learning to use the language of first-order logic takes some practice. Consider the following statements:

- Every integer is even or odd, but not both.
- A integer is even if and only if it is divisible by two.
- If some integer, x, is even, then so is x^2.
- A integer x is even if and only if $x + 1$ is odd.
- For any three integers x, y, and z, if x divides y and y divides z, then x divides z.

Given the language of arithmetic described above, let's write $x + y$ instead of $f(x, y)$ and $x \cdot y$ instead of $g(x, y)$. We can then write 2 for $1 + 1$ and x^2 for $x \cdot x$, and we can define the following formulas:

- $\mathit{even}(x) \equiv \exists y.\, x = 2 \cdot y$
- $\mathit{odd}(x) \equiv \exists y.\, x = 2 \cdot y + 1$
- $x \mid y \equiv \exists z.\, y = x \cdot z$.

With these, the statements above can be written as follows:

- $\forall x.\, (\mathit{even}(x) \vee \mathit{odd}(x)) \wedge \neg(\mathit{even}(x) \wedge \mathit{odd}(x))$
- $\forall x.\, \mathit{even}(x) \leftrightarrow 2 \mid x$
- $\forall x.\, \mathit{even}(x) \rightarrow \mathit{even}(x^2)$
- $\forall x.\, \mathit{even}(x) \leftrightarrow \mathit{odd}(x + 1)$
- $\forall x.\, \forall y.\, \forall z.\, x \mid y \wedge y \mid z \rightarrow x \mid z$.

The statement with which we began this chapter might be written as follows:

$$\mathit{married}(\mathit{Bill}, \mathit{Tracy}) \wedge \forall x.\, \mathit{childOf}(x, \mathit{Bill}) \wedge \mathit{childOf}(x, \mathit{Tracy}) \rightarrow \mathit{smart}(x).$$

When reading such formulas, we give the quantifiers the widest scope possible, and use parentheses to limit the scope. In other words, $\forall x.\, A \wedge B$ means $\forall x.\, (A \wedge B)$, and we write $(\forall x.\, A) \wedge B$ if we want to limit the scope to A. We can shorten the last example by writing $\forall x\, y\, z.\, x \mid y \wedge y \mid z \rightarrow x \mid z$.

Notice that quantifiers always range over the entire universe of objects, integers in the first set of examples and possibly people in the example involving Bill and Tracy. We can restrict the domain of a quantifier using propositional connectives:

- To say "there is an even number between 1 and 3," we write $\exists x.\, \mathit{even}(x) \wedge 1 < x \wedge x < 3$.
- To say "every even number greater than 1 is greater than 3," we write $\forall x.\, \mathit{even}(x) \wedge x > 1 \rightarrow x > 3$.

This process is known as *relativization*.

It is natural to consider variations on first-order logic with different *sorts* of variables. For example, a formal representation of Euclidean geometry might have variables $p, q, r, \ldots$ ranging over points, other variables $L, M, N, \ldots$ ranging over lines, and maybe even variables $\alpha, \beta, \gamma, \ldots$ ranging over circles. A relation symbol $\mathit{on}(p, L)$ used to express that point p lies on the line L should come with a specification that the first argument is a point and the second argument is a line. This is known as *many-sorted first-order logic*. We will consider even more expressive generalizations of first-order logic later on. In the meanwhile, to keep the theoretical exposition simple, we will focus on first-order logic with only one variable sort.

10.3 Semantics

In order to evaluate a propositional formula A, all we need to know is the assignment of truth values to variables occurring in A. Given such an assignment, τ, we were able to define the truth value $[\![A]\!]_\tau$.

In first-order logic, there are two things we need to evaluate, namely, terms and formulas. A term like $f(x, g(y, z))$ has variables x, y, z, that we think of as ranging over objects, like numbers, people, or whatever. So in this case our interpretation of the language has to specify the domain of objects that we have in mind. It also has to specify interpretations of all the function and relation symbols in the language as functions and relations on the corresponding set. Such a structure is known as a *model*.

Definition

A *model* $\mathfrak{M}$ for a language consists of

- A set of objects, $|\mathfrak{M}|$, called the *universe* of $\mathfrak{M}$.
- For each function symbol f in the language, a function $f^{\mathfrak{M}}$ from the universe of $\mathfrak{M}$ to itself, with the corresponding arity.
- For each relation symbol R in the language, a relation $R^{\mathfrak{M}}$ on the universe of $\mathfrak{M}$, with the corresponding arity.

Let σ be an assignment of elements of $|\mathfrak{M}|$ to variables. Then every term t has a value $[\![t]\!]_{\mathfrak{M},\sigma}$ in $|\mathfrak{M}|$ defined recursively as follows:

- $[\![x]\!]_{\mathfrak{M},\sigma} = \sigma(x)$
- For every n-ary function symbol f and every tuple of terms $t_1, \ldots, t_n$, $[\![f(t_1, \ldots, t_n)]\!]_{\mathfrak{M},\sigma} = f^{\mathfrak{M}}([\![t_1]\!]_{\mathfrak{M},\sigma}, \ldots, [\![t_n]\!]_{\mathfrak{M},\sigma})$

Remember that we can think of constant symbols as 0-ary function symbols, so $[\![c]\!]_{\mathfrak{M},\sigma} = c_{\mathfrak{M}}$ is implicit in the second clause.

We can also say what it means for a formula A to be true in $\mathfrak{M}$ relative to the assignment σ, which we write as $\mathfrak{M} \models_\sigma A$:

- $\mathfrak{M} \models_\sigma t = t'$ if and only if $[\![t]\!]_{\mathfrak{M},\sigma} = [\![t']\!]_{\mathfrak{M},\sigma}$.
- $\mathfrak{M} \models_\sigma R(t_0, \ldots, t_{n-1})$ if and only if $R^{\mathfrak{M}}([\![t_0]\!]_{\mathfrak{M},\sigma}, \ldots, [\![t_{n-1}]\!]_{\mathfrak{M},\sigma})$.
- $\mathfrak{M} \models_\sigma \bot$ is always false.
- $\mathfrak{M} \models_\sigma \top$ is always true.
- $\mathfrak{M} \models_\sigma A \wedge B$ if and only if $\mathfrak{M} \models_\sigma A$ and $\mathfrak{M} \models_\sigma B$.
- $\mathfrak{M} \models_\sigma A \vee B$ if and only if $\mathfrak{M} \models_\sigma A$ or $\mathfrak{M} \models_\sigma B$.
- $\mathfrak{M} \models_\sigma A \to B$ if and only if $\mathfrak{M} \not\models_\sigma A$ or $\mathfrak{M} \models_\sigma B$.
- $\mathfrak{M} \models_\sigma A \leftrightarrow B$ if and only if $\mathfrak{M} \models_\sigma A$ and $\mathfrak{M} \models_\sigma B$ either both hold or both don't hold.
- $\mathfrak{M} \models_\sigma \forall x.\, A$ if and only if for every $a \in |\mathfrak{M}|$, $\mathfrak{M} \models_{\sigma[x \mapsto a]} A$.
- $\mathfrak{M} \models_\sigma \exists x.\, A$ if and only if for some $a \in |\mathfrak{M}|$, $\mathfrak{M} \models_{\sigma[x \mapsto a]} A$.

Most of the clauses are the same as for propositional logic. It's the first two base cases and the clauses for the quantifiers that are new.

The fact that formulas can be interpreted in different models is central to modern logic. Take, for example, the sentence $\forall x.\, \exists y.\, R(x, y)$. This is true of the real numbers with the less-than relation, but false of the natural numbers with the greater-than relation. What about the integers with the relation "x divides y?"

If $\mathfrak{M} \models_\sigma A$, we also say that A is satisfied by $\mathfrak{M}$ and σ, and that A is *satisfiable*. Remember that saying that A is a sentence means that it has no free variables. In that case, we don't have to talk about an assignment σ, so a sentence is satisfiable if it is true in some model. If $\mathfrak{M} \models A$, we also say that $\mathfrak{M}$ is a model of A.

We say that a sentence A is *valid*, written $\models A$, if it is true in every model. (More generally, a formula with free variables is valid if it is true in every model with respect to every assignment of variables to elements in the universe of that model.) In analogy with propositional logic, if Γ is a set of sentences and A is a sentence, we say Γ *entails* A and write $\Gamma \models A$ if every model of Γ is also a model of A. This means that the symbol $\models$ is overloaded, since $\Gamma \models A$ and $\mathfrak{M} \models A$ mean different things. Sometimes people also write $\models_{\mathfrak{M}} A$ instead of $\mathfrak{M} \models A$, and $\models_{\mathfrak{M},\sigma} A$ instead of $\mathfrak{M} \models_\sigma A$.

10.4 Normal forms

The notion of a formula in negation normal form carries over to first-order logic, where now we allow quantifiers as well. Every formula can be put in negation normal form using the identities $(\neg\forall x.\, A) \leftrightarrow \exists x.\, \neg A$ and $(\neg\exists x.\, A) \leftrightarrow \forall x.\, \neg A$.

Is there anything similar to CNF? It helps that it is always possible to bring quantifiers to the front of a formula, using these identities, which hold when x is not free in B:

$$(\forall x.\, A) \vee B \leftrightarrow \forall x.\, A \vee B$$
$$(\forall x.\, A) \wedge B \leftrightarrow \forall x.\, A \wedge B$$
$$(\exists x.\, A) \vee B \leftrightarrow \exists x.\, A \wedge B$$
$$(\exists x.\, A) \wedge B \leftrightarrow \exists x.\, A \wedge B$$

We can ensure that these apply by renaming the bound variable if necessary. We can then put the inside formula in CNF if we want. But it turns out to be more useful to automated reasoning to eliminate the quantifiers entirely. We can do that using *Skolem functions*, which we will tell you about later on.

CHAPTER

ELEVEN

IMPLEMENTING FIRST-ORDER LOGIC

Our implementation of first-order logic in Lean is similar to our implementation of propositional logic in Lean, covering both the syntax and the semantics. We will also show how to implement *unification*, and algorithm that is fundamental to the mechanization of first-order reasoning.

11.1 Terms

Our implementation of terms is straightforward.

```
inductive FOTerm
  | var : String → FOTerm
  | app : String → List FOTerm → FOTerm
  deriving Repr, Inhabited, BEq
```

A term is either a variable or a function symbol applied to a list of terms. We have defined syntax for *FOTerm*:

```
def ex1 := term!{ %x }
def ex2 := term!{ c }
def ex3 := term!{ f(f(a, %x), f(g(c, f(%y, d)), b)) }

#print ex1
#print ex2
#print ex3

#eval ex1
#eval ex2
#eval ex3
```

The notation *%x* is used for a variable. Notice that a constant like *c* is represented as an application of the symbol to the empty list. Notice also that the definition does nothing to check the arity of the function symbols. Ordinarily, first-order logic allows us to specify that *f* and *g* are binary functions and that another function, *h*, is unary. Our definition of *FOTerm* allows the application of any string to any number of arguments. This simplifies a number of aspects of the implementation. As an exercise, you might want to write a function *well-formed* in Lean that, given a specification of a list of symbols and their arities, checks that an *FOTerm* uses only those symbols and with the correct arity. Later in the course, we will talk about systems more expressive than first-order logic that provide other ways of specifying a function's intended arguments. Lean's type system provides a very elaborate and expressive means for doing so, and you can think of the specification of arities in first-order logic as being a minimal form of a typing judgment.

Remember that to evaluate a first-order language, we need an assignment of values to the variables, as well as interpretations of the function and relation symbols. Since our symbols are identified as strings, in general an interpretation of symbols is an assignment of values to strings:

```
def FOAssignment α := String → α
```

Any function definable in Lean can serve this purpose. Keep in mind that we have to fix a type α, corresponding to the universe of the structure in which we carry out the interpretation.

Since it is often useful to specify an assignment by giving a finite list of values, we have implemented syntax for that:

```
#eval assign!{x ↦ 3, y ↦ 5, z ↦ 2}
```

You can type the symbol ↦ as *\mapsto*. Formally, the notation produces an *association list*, essentially just a list of key / value pairs. But we have also told Lean how to coerce such an association list to an *FOAssignment* when necessary. The following examples provide a few different Lean idioms for specifying that an *assign!* expression should be interpreted as an *FOAssignment*. (It should also happen automatically when you pass such an expression as an argument to a function that expects an *FOAssignment*.)

```
def assign1 : FOAssignment Nat := assign!{x ↦ 3, y ↦ 5, z ↦ 2}
#check assign1
#eval  assign1 "x"

#check (assign!{x ↦ 3, y ↦ 5, z ↦ 2} : FOAssignment Nat)

#check @id (FOAssignment Nat) assign!{x ↦ 3, y ↦ 5, z ↦ 2}

#check show FOAssignment Nat from assign!{x ↦ 3, y ↦ 5, z ↦ 2}

#check let this : FOAssignment Nat := assign!{x ↦ 3, y ↦ 5, z ↦ 2}
      this
```

It is now easy to define substitution for terms. Such a function should take a term and an assignment of terms to variables, and replace the variables by the assigned terms.

```
partial def subst (σ : FOAssignment FOTerm) : FOTerm → FOTerm
  | var x   => σ x
  | app f l => app f (l.map (subst σ))
```

Here we try it out:

```
#eval ex3.subst assign!{x ↦ term!{h(a)}, y ↦ term!{f(a, h(a), d)}}
```

11.2 Evaluating terms

To evaluate a term, we need not only an assignment of values to the variables occurring in the term, but also an interpretation of all the function symbols. Setting aside concerns about arities, we can interpret any function taking some number of elements of α to α as an element of type *List* $\alpha \to \alpha$.

```
/-- an interpretation of function symbols -/
def FnInterp α := FOAssignment (List α → α)

-- coerces an association list to a function
instance [Inhabited α] : Coe (AssocList String (List α → α)) (FnInterp α) :=
⟨fun l => l.getA⟩
```

If a function is intended to be used as a binary function, we really care about the interpretation when it is applied to lists of length two. In our quick-and-dirty implementation, we have to define values for lists of other lengths, but any values will do. For example, we can define an interpretation of constants and functions on the natural numbers as follows:

```
def arithFnInterp : FnInterp Nat
  | "plus"  => fun l => l.getA 0 + l.getA 1
  | "times" => fun l => l.getA 0 * l.getA 1
  | "zero"  => fun l => 0
  | "one"   => fun l => 1
  | "two"   => fun l => 2
  | "three" => fun l => 3
  | _       => fun l => arbitrary
```

Or, alternatively:

```
def arithFnInterp' : FnInterp Nat :=
assign!{
  plus ↦ fun l : List Nat => l.getA 0 + l.getA 1,
  times ↦ fun l => l.getA 0 * l.getA 1,
  zero ↦ fun l => 0,
  one ↦ fun l => 1,
  two ↦ fun l => 2,
  three ↦ fun l => 3 }
```

With *FnInterp* in place, it is easy to define a function that evaluates terms:

```
/-- evaluate a term relative to a variable assignment -/
partial def eval {α} [Inhabited α] (fn : FnInterp α) (σ : FOAssignment α) : FOTerm →␣
↪α
  | var x   => σ x
  | app f l => fn f (l.map (eval fn σ))
```

Even though the function always terminates, Lean 4 is not yet able to prove termination automatically, so we use the keyword *partial*. Let's try it out.

```
def arith_ex1 := term!{ plus(times(%x, two), plus(%y, three)) }

#eval arith_ex1.eval arithFnInterp assign!{ x ↦ 3, y ↦ 5 }
#eval arith_ex1.eval arithFnInterp assign!{ x ↦ 7, y ↦ 11 }
```

When we talked about propositional logic, we proved a theorem that says that evaluating the result of a substitution is the same as evaluating the original formula relative to an assignment of the values of the substituted formula. In the context of terms, the identity is as follows:

$$[\![t[s/x]]\!]_\sigma = [\![t]\!]_{\sigma[x \mapsto [\![s]\!]_\sigma]}.$$

The proof is essentially the same. Our current implementation is more general in that it allows us to substitute multiple terms at once, but we can see the principle at play in the fact that the two evaluations below produce the same answer.

```
def arith_ex2 := term!{ plus(one, times(three, %z))}

def arith_ex3 := term!{ plus(%z, two) }

-- these two should give the same result!

#eval (arith_ex1.subst
        assign!{ x ↦ arith_ex2, y ↦ arith_ex3 }).eval
        arithFnInterp assign!{z ↦ 7}

#eval arith_ex1.eval arithFnInterp
  assign!{ x ↦ (arith_ex2.eval arithFnInterp assign!{z ↦ 7}),
           y ↦ (arith_ex3.eval arithFnInterp assign!{z ↦ 7}) }
```

And here is another crazy idea: we can view substitution as the result of evaluating a term in a model where the universe consists of terms, and where each function symbol f is interpreted as the function "build a term by applying f."

```
def TermFnInterp : FnInterp FOTerm := FOTerm.app

def FOTerm.subst' := eval TermFnInterp

-- the same!
#eval arith_ex1.subst assign!{ x ↦ arith_ex2, y ↦ arith_ex3 }
#eval arith_ex1.subst' assign!{ x ↦ arith_ex2, y ↦ arith_ex3  }
```

You should think about what is going on here. Such a model is known as a *term model*.

11.3 Formulas

Since the universe of a first-order model may be infinite, we would not expect to be able to evaluate arbitrary first-order formulas in an arbitrary model. Evaluating quantifiers in our model of arithmetic would require testing instantiations to all the natural numbers. If we could do that, we could easily settle the truth of the sentence

$$\forall x.\, \exists y.\, y > x \wedge \mathit{prime}(y) \wedge \mathit{prime}(y+2),$$

which says that there are infinitely many values y such that y and $y+2$ are both prime. This is known as the *twin primes conjecture*, and it is a major open question in number theory.

We can, however, evaluate quantifiers over finite universes. In the definition of a first-order model below, we assume that the universe is given by a finite list of values.

```
/-- an interpretation of relation symbols -/
def RelInterp α := FOAssignment (List α → Bool)

structure FOModel (α : Type) where
  (univ : List α)
  (fn : FnInterp α)
  (rel : RelInterp α)
```

In our quick-and-dirty implementation, we don't require that the universe *univ* is closed under the functions. In other words, it's possible that we can apply a function to some elements on the list *univ* and get something that isn't in the list. It wouldn't be hard to write a procedure that checks that, given a finite list of functions and their intended arities. (A more efficient way of handling this is instead to *prove* that the functions all return values in universe, using Lean's theorem proving capabilities.) In the examples in this section, we won't use function symbols at all, except for some constants that are clearly o.k.

To handle the quantifiers, we need a procedure that takes an assignment σ and produces an updated assignment $\sigma[x \mapsto v]$.

```
def FOAssignment.update (σ : FOAssignment α) (x : String) (v : α) : FOAssignment α
  | y => if y == x then v else σ y
```

With that in hand, the evaluation function is entirely straightforward.

```
def FOForm.eval {α} [Inhabited α] [BEq α]
    (M : FOModel α) (σ : FOAssignment α) : FOForm → Bool
  | eq t1 t2 => t1.eval M.fn σ == t2.eval M.fn σ
  | rel r ts =>  M.rel r (ts.map $ FOTerm.eval M.fn σ)
  | tr => true
  | fls => false
```

(continues on next page)

(continued from previous page)

```
  | neg A => !(eval M σ A)
  | conj A B => (eval M σ A) && (eval M σ B)
  | disj A B => (eval M σ A) || (eval M σ B)
  | impl A B => !(eval M σ A) || (eval M σ B)
  | biImpl A B => (!(eval M σ A) || (eval M σ B)) && (!(eval M σ B) || (eval M σ A))
  | ex x A => M.univ.any fun val => eval M (σ.update x val) A
  | all x A => M.univ.all fun val => eval M (σ.update x val) A
```

Let's try it out on a baby model of arithmetic that only has the numbers 0 to 9 in the universe. We reuse the function interpretation *arithFnInterp* from before, so that we have the constants *zero*, *one*, *two*, and *three*. The interpretation also gives us addition and multiplication, but we won't use those. As far as relations, we interpret two: the binary less-than relation *lt* and the predicate *even*. We also define the trivial assignment which assigns 0 to all the variables.

```
def babyArithMdl : FOModel Nat where
  univ := List.range 10  /- 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 -/
  fn   := arithFnInterp
  rel  := assign!{
            lt ↦ fun l : List Nat => if l.getA 0 < l.getA 1 then true else false,
            even ↦ fun l : List Nat => l.getA 0 % 2 == 0 }

def trivAssignment : FOAssignment Nat := fun x => 0
```

We can try it out:

```
#eval fo!{even(%x)}.eval babyArithMdl assign!{x ↦ 5}
#eval fo!{even(%x)}.eval babyArithMdl assign!{x ↦ 6}
#eval fo!{∃ y. lt(%x, %y)}.eval babyArithMdl assign!{x ↦ 8}
#eval fo!{∃ y. lt(%x, %y)}.eval babyArithMdl assign!{x ↦ 9}
```

It's an unpleasant feature of our syntax that we have to put a percent sign in front of variables in order to distinguish them from constants, while we do not use the percent sign with the quantifiers. To facilitate testing sentences, we write a simple function *testeval* that evaluates any formula in our model with respect to the trivial variable assignment.

```
def FOForm.testeval (A : FOForm) : Bool := A.eval babyArithMdl trivAssignment

#eval fo!{even(two)}.testeval
#eval fo!{even(three)}.testeval
#eval fo!{∃ x. even(%x)}.testeval
#eval fo!{∀ x. even(%x)}.testeval

#eval fo!{∃ x. lt(%x, two) ∧ even(%x)}.testeval
#eval fo!{∃ x. ∃ y. lt(%x, %y) ∧ lt(%y, two) ∧ even(%x) ∧ even(%y)}.testeval
#eval fo!{∀ x. even(%x) ∧ lt(%x,two) → %x = zero}.testeval
#eval fo!{∀ x. even(%x) ∧ lt(%x,three) → %x = zero}.testeval
#eval fo!{∀ x. even(%x) ∧ lt(%x,three) → %x = zero ∨ %x = two}.testeval

#eval fo!{∀ x. ∃ y. lt(%x, %y)}.testeval
#eval fo!{∀ x. even(%x) → ∃ y. lt(%x, %y)}.testeval
#eval fo!{∀ x. ¬ even(%x) → ∃ y. lt(%x, %y)}.testeval
```

A software package called *Tarski's World* by Jon Barwise and John Etchemendy offers a fun way of experimenting with first-order logic. It allows users to lay down blocks on an 8×8 grid and evaluate sentences about them. Each block is either a tetrahedron, a cube, or a dodecahedron, and each is either small, medium, or large. The language includes predicates for these, as well as relations like *FrontOf(x, y)* and *Between(x, y, z)*. The second one of these holds if and only if x, y, and z are either in the same row, in the same column, or on the same diagonal, and x is between y and z.

The file *TarskisWorld* in the *Examples* folder includes a simple implementation in Lean. You can define a world and

display it:

```
def myWorld : World := [
  ⟨tet, medium, 0, 2⟩,
  ⟨tet, small, 0, 4⟩,
  ⟨cube, small, 4, 4⟩,
  ⟨cube, medium, 5, 6⟩,
  ⟨dodec, small, 7, 0⟩,
  ⟨dodec, large, 7, 4⟩ ]

#eval myWorld.show

/-
-----------------------------------------
| D- |    |    |    | D+ |    |    |    |
-----------------------------------------
|    |    |    |    |    |    |    |    |
-----------------------------------------
|    |    |    |    |    |    | C  |    |
-----------------------------------------
|    |    |    |    | C- |    |    |    |
-----------------------------------------
|    |    |    |    |    |    |    |    |
-----------------------------------------
|    |    |    |    |    |    |    |    |
-----------------------------------------
|    |    |    |    |    |    |    |    |
-----------------------------------------
|    |    | T  |    | T- |    |    |    |
-----------------------------------------
-/
```

Here a plus symbol means that the object is large, a minus symbol means that it is small, and no symbol means that it is of medium size. You can then evaluate statements about the world:

```
#eval myWorld.eval fo!{∃ x. ∃ y. ∃ z. Between(%x, %y, %z)}
#eval myWorld.eval fo!{∃ x. ∃ y. ∃ z. Cube(%x) ∧ Between(%x, %y, %z)}
#eval myWorld.eval fo!{∃ x. ∃ y. ∃ z. Dodec(%x) ∧ Between(%x, %y, %z)}
#eval myWorld.eval fo!{∃ x. Small(%x)}
#eval myWorld.eval fo!{∃ x. Small(%x) ∧ Cube(%x) }
#eval myWorld.eval fo!{∀ x. ∀ y. Cube(%x) ∧ Tet(%y) → FrontOf(%x, %y)}
#eval myWorld.eval fo!{∀ x. ∀ y. Cube(%x) ∧ Dodec(%y) → FrontOf(%x, %y)}
#eval myWorld.eval fo!{∀ x. Tet(%x) → ∃ y. Cube(%y) ∧ RightOf(%y, %x) }
#eval myWorld.eval fo!{∀ x. Dodec(%x) → ∃ y. Tet(%y) ∧ RightOf(%y, %x) }
```

The file *TWExamples* provides two puzzles, taken from *Tarski's World*, for you to try.

11.4 Unification

Suppose we are working with a language that is meant to describe the real numbers and we have either proved or assumed the following sentence:

$$\forall x, y, z.\, x < y \to x + z < y + z.$$

Suppose also that we are trying to prove

$$ab + 7 < c + 7.$$

Even though we haven't talked about proof systems for first-order logic yet, it should be clear that we want to instantiate the universal quantifiers in the sentence by substituting ab for x, c for y, and 7 for z, so that the conclusion matches the goal.

You probably did not have to think about this much. You identified the problem as that of finding a substitution for x, y, and z that has the effect of making the expression $x + z < y + z$ the same as $ab + 7 < c + 7$. This is what is known as a *matching* problem for first-order terms. This kind of pattern matching is fundamental to reasoning, and hence also fundamental to logic. The general formulation of the problem is as follows: we are given pairs of terms $(s_1, t_1), (s_2, t_2), \ldots, (s_n, t_n)$ and we are looking for a substitution σ with the property that for every i, $\sigma\, s_i = t_i$.

The generalization of the matching problem in which we are allowed to substitute for the variables in the terms t_i as well as the terms s_i is known as *unification*. In other words, the input to a unification problem is the same as before, but now we are looking for a substitution σ with the property that for every i, $\sigma\, s_i = \sigma\, t_i$. For example, consider the following two expressions:

$$f(x, f(x, a)) < z \qquad f(b, y) < c$$

If we substitute b for x, $f(b, a)$ for y, and c for z, both expressions become $f(b, f(b, a)) < c$. When we discuss resolution proofs for first-order logic, we will see that such unification problems come up in a context where are are trying to prove a contradiction and we have derived something like $\forall x, z.\, f(x, f(x, a)) < z$ and $\forall y.\, f(b, y) \not< c$. The unification problem tells us how to instantiate the universal quantifiers in order to prove a contradiction.

To prevent confusion in the future, we would like to point out that what counts as a variable or a constant depends on the context. For example, we can ask for an assignment to x and y that unifies $f(x, z)$ and $f(z, y)$ without assigning anything to z. In that case, we are treating x and y as variables and z as a constant. In Lean, the variables x and y that can be assigned in a unification problem are sometimes called *metavariables* and written as $?x$ and $?y$. For simplicity, we will continue to use letters like x, y, and z for variables and letters like a, b, and c as constants. What is important when specifying a unification problem is simply that it is clear which symbols play the role of variables and which have to remain fixed.

There is a linear-time algorithm that solves any unification problem or determines that there is no solution, and, in fact produces a *most general unifier*, or *mgu*. A most general unifier σ for a problem is a substitution that unifies each pair in the list and has the property that if τ is *any* unifier for the problem, then τ can be expressed as the result of following σ by another substitution. Here we will describe such an algorithm, but we will not go so far as to prove its correctness or show that it produces a most general unifier. Our implementation is adapted from one by John Harrison in his *Handbook of Practical Logic and Automated Reasoning*, which is an excellent reference for most of the topics we cover in this book, and a lot more. In particular, it proves that the algorithm we describe below always terminates with a most general unifier if the pairs can be unified at all, and fails with *none* otherwise. There is also a nice presentation of unification in the book *Term Rewriting and All that* by Franz Baader and Tobias Nipkow.

Some unification problems are easy. To unify x and $f(a, z)$, we simply assign x to $f(a, z)$. To be a most general unifier, it is important that the resulting term, $f(a, z)$ has the variable z. We can also solve the problem by assigning $f(a, a)$ to x and a to z, but that is less general. It can be seen as the result of mapping x to $f(a, z)$ and then applying another substitution that maps z to a.

For another easy example, we can unify $f(x, b)$ and $f(g(c), b)$ by mapping x to $g(c)$. Since both expressions are of the form $f(\cdot, \cdot)$, to unify the expressions it is clearly necessary and sufficient to unify the arguments. For the same reason, it is clear that $f(x, b)$ and $g(c)$ can't be unified.

An interesting phenomenon arises with the unification problem $(x, f(y)), (y, g(x))$. The first pair tells us we have to map x to $f(y)$, and the second pair tells us we have to map y to $g(x)$. Applying this substitution to each pair yields $(f(y), f(g(x)))$ in the first pair and $(g(x), g(f(y)))$ in the second, which doesn't solve the problem. Repeating the substitution doesn't help. The problem is that when we start with x, we find a chain of assignments to variables in the terms on the right-hand side that ultimately leads to a term that involves x. In others words, the list of associations contains a *cycle*. If we ever reach a point where we are forced to admit a cycle of associations, the algorithm should fail. This is known as the *occurs check*.

The algorithm below maintains an *association list* of pairs (x, t) indicating that the variable x should be unified with the term t. (An association list is essentially just a list of pairs, but for efficiency it uses a constructor with three arguments to cons a pair onto the list.) Each variable x occurs only once on the left side of a pair. We allow the list to contain pairs like (x, y), (y, z), and (z, w), since we can clean that up later, say, by assigning all the variables to w.

Suppose we have an association list *env* and we are considering adding the pair (x, t). The following function returns *some true* if the assignment is trivial, which is to say, t is x or there is a chain of variable associations that amounts to assigning x to x. In that case, we can ignore the pair. The function returns *some false* for any nontrivial assignment, unless it detects a cycle, in which case it returns *none* to indicate failure.

```
partial def isTriv? (env : AssocList String FOTerm) (x : String) :
    FOTerm → Option Bool
  | var y      => if y = x then true
                  else if !env.contains y then false
                  else isTriv? env x (env.getA y)
  | app f l    => loop l
where
  loop : List FOTerm → Option Bool
    | []    => false
    | a::as => match isTriv? env x a with
                 | true  => none
                 | false => loop as
                 | none  => none
```

With that in place, the following function takes an association list, *env*, and a list of pairs to unify, and it returns an association list. The clauses are as follows:

- If the list of pairs is empty, there is nothing to do.
- If the first pair on the list is a pair of function applications, then
  - if the pair is of the form $f(t_1, \ldots, t_n)$ and $f(s_1, \ldots, s_n)$, add the pairs $(s_1, t_1) \ldots (s_n, t_n)$ to the list of pairs to unify and continue recursively, and
  - if the function symbols don't match or the number of arguments is not the same, fail.
- If the pair is of the form (x, t), then
  - if there is a pair (x, s) already in *env*, add (s, t) to the list of pairs to unify and continue recursively, and
  - otherwise add (x, t) to *env* unless it is a trivial assignment, and continue recursively with the remaining pairs.
- If the pair is of the form (t, x), then turn it around and recursively use the previous case.

The algorithm is implemented as follows.

```
partial def unify? (env : AssocList String FOTerm) : List (FOTerm × FOTerm) →
                                                      Option (AssocList String FOTerm)
  | [] => some env
```

(continues on next page)

(continued from previous page)

```
  | (app f1 l1, app f2 l2) :: eqs =>
      if f1 = f2 ∧ l1.length = l2.length then
        unify? env ((l1.zip l2) ++ eqs)
      else none
  | (var x, t) :: eqs =>
      if env.contains x then unify? env (eqs.cons (env.getA x, t))
      else match isTriv? env x t with
        | true  => unify? env eqs
        | false => unify? (env.cons x t) eqs
        | none  => none
  | (t, var x) :: eqs => unify? env ((var x, t) :: eqs)
```

The final association list might contain pairs (x, t) and (y, s) where s contains the variable x. This means that the variable x has to unify with t, which we can achieve by mapping x to t. But the resulting substitution will also replace x in s, so we had better carry out the substitution for x there too. The following function, *usolve*, cleans up the list of pairs by iteratively substituting the terms on the right for the variables on the left, until the association list no longer changes. The fact that we have avoided cycles guarantees that it terminates. The function after that, *fullUnify*, puts it all together: given a list of pairs of terms to unify, it computes the association list and uses *usolve* to turn it into a substitution.

```
partial def usolve (env : AssocList String FOTerm) : AssocList String FOTerm := do
  let env' := env.mapVal (subst env)
  if env' == env then env else usolve env'

partial def fullUnify (eqs : List (FOTerm × FOTerm)) : Option (AssocList String␣
↪FOTerm) :=
  match unify? AssocList.nil eqs with
    | some l => usolve l
    | none   => none
```

Below we try the procedure out by computing some unifiers and applying them to the original pairs to make sure that the pairs are indeed unified.

```
partial def unifyAndApply (eqs : List (FOTerm × FOTerm)) : Option (List (FOTerm ×␣
↪FOTerm)) :=
  match fullUnify eqs with
    | some l => let σ : FOAssignment FOTerm := l
                eqs.map (fun (s, t) => (subst σ s, subst σ t))
    | none   => none

def unify_ex1 := [ (term!{ f(%x, g(%y))}, term!{ f(f(%z), %w) }) ]

#eval toString $ fullUnify unify_ex1
#eval toString $ unifyAndApply unify_ex1

def unify_ex2 := [ (term!{ f(%x, %y) }, term!{ f(%y, %x) }) ]

#eval toString $ fullUnify unify_ex2
#eval toString $ unifyAndApply unify_ex2

def unify_ex3 := [ (term!{ f(%x, g(%y)) }, term!{ f(%y, %x) }) ]

#eval toString $ fullUnify unify_ex3

def unify_ex4 := [ (term!{ %x0 }, term!{ f(%x1, %x1) } ),
                   (term!{ %x1 }, term!{ f(%x2, %x2) } ),
                   (term!{ %x2 }, term!{ f(%x3, %x3) } ),
```

(continues on next page)

(continued from previous page)

```
                    (term!{ %x3 }, term!{ f(%x4, %x4) } )]

#eval toString $ fullUnify unify_ex4
#eval toString $ unifyAndApply unify_ex4
```

CHAPTER

TWELVE

DECISION PROCEDURES FOR FIRST-ORDER LOGIC

Given a propositional formula A and a truth assignment τ, we have seen that it is straightforward to test wither A is true under τ. We have also seen that a formula A is valid if and only if it is provable, and we have considered decision procedures for propositional logic that determine whether that is the case.

In a similar way, given a first-order sentence A and a model $\mathfrak{M}$, we can ask whether A is true in $\mathfrak{M}$. But for most fixed choices of $\mathfrak{M}$, there is no algorithm to determine whether or not A is true. For example, there is no algorithm to determine whether a sentence in a language with two binary function symbols is true of the model $(\mathbb{Z}, +, \times)$.

Given a first-order sentence A, we can also ask whether it is valid, that is, true in all models. Once we have suitable proof systems for first-order logic, this will be equivalent to the question as to whether A is provable. If the language has a binary relation symbol or two unary functions, this, too, is undecidable.

But all is not lost. For some interesting models, the question of truth is decidable. These include the theory of the real numbers $(\mathbb{R}, 0, 1, +, \times, <)$ with zero, one, addition, multiplication, and the less-than relation, and the theory of the integers $(\mathbb{Z}, 0, 1, +, <)$ in the same language except without multiplication.

We can also ask about validity for restricted classes of formulas. A formula is said to be *quantifier-free* if it has no quantifiers, and *universal* if it consists of any number of universal quantifiers $\forall$ followed by a quantifier-free formula. Interestingly, the question as to whether a universal first-order formula is valid is decidable.

Another thing we can do is ask whether a formula A is provable from some axioms Γ. For some fixed choices of Γ, this question is decidable. For example, there are natural axioms that characterize truth in the two structures mentioned above, $(\mathbb{R}, 0, 1, +, \times, <)$ and $(\mathbb{Z}, 0, 1, +, <)$.

From the theoretical standpoint, before looking for a computational solution to a problem in logic, the first challenge is to determine whether or not the problem is decidable. If the answer is "yes," we can look for implementations that are efficient in practice. If the answer is "no," the best we can do is look for simplifications or approximations. We will see that SMT solvers focus on the case where the answer is positive. The goal of this chapter is to establish decidability in a few interesting cases, without worrying about efficiency and implementation.

12.1 Linear arithmetic

A *linear expression* is one of the form $a_1x_1 + a_2x_2 + \cdots + a_nx_n + b$, where each a_i is a rational number, b is a rational number, and each x_i is a variable. We think of the variables x_i as ranging over the real numbers. A *linear constraint* is one of the form $s = t$ or $s < t$, where s and t are linear expressions. (In practice, we usually include constraints of the form $s \leq t$ and sometimes $s \neq t$ as well, but let's keep it simple for now.)

Notice that any linear constraint is equivalent to one of the form $t = 0$ or $t > 0$, since we can move all the terms to one side. For example, the constraint $3x + 2y < 3y + 4z$ is equivalent to $-3x + y + 4z > 0$. An important observation that we will use below is that any linear constraint that involves a variable x can be written as $x = t$, $x < t$, or $t < x$, where x does not occur in t. We do this by simply solving for x. For example, the previous constraint can be expressed as $x < (1/3)y + (4/3)z$. Remember that dividing both sides of an inequality by a negative number reverses the direction.

A set Γ of linear constraints is *satisfiable* if there is an assignment of real values to the variables that makes them all true. Our first goal is to prove the following.

Theorem

The question as to whether a finite set of linear constraints is satisfiable is decidable.

Proof

We use induction on the number of variables. If there are no variables at all, Γ contains only expressions of the form $b_0 < b_1$ or $b_0 = b_1$ where b_0 and b_1 are constants, and we only need to perform the comparisons to see whether they are true. Remember that if Γ is the empty set, we take it to be trivially satisfied.

In the inductive step, Γ contains a variable. If Γ contains any false constant equations, it is unsatisfiable, and it it contains any true constant equations, we can remove them without affecting satisfiability. If Γ contains a nontrivial equation with a variable x, we put it in the form $x = t$ and then substitute t for x everywhere. The resulting set of constraints has one fewer variable, and clearly it is equisatisfiable with the original one. Given an assignment to the new set of constraints, we just assign x the value of t.

So we can now assume that there are no equations in Γ. We can divide the inequalities in Γ intro three kinds:

- those that don't contain x at all
- those that can be expressed in the form $s_i < x$
- those that can be expressed in the form $x < t_j$

Let Γ' be the set that results from removing the inequalities in the last two categories and replacing them with inequalities of the form $s_i < t_j$. We claim Γ' is equisatisfiable with Γ. Clearly any assignment that satisfies Γ also satisfies Γ'. Conversely, suppose σ is an assignment that satisfies Γ'. Then, under that assignment, the value of each s_i is less than the value of every t_j. We obtain an assignment satisfying Γ by mapping x to any value between the largest s_i and the smallest t_j. (If one of the last two categories is empty, we remove the constraints in the other category entirely, since they can be satisfied by taking x sufficiently large or sufficiently small.)

The procedure implicit in this proof is known as the *Fourier-Motzkin* procedure, since an incipient presentation of the idea can be found in the work of Jean-Baptiste Joseph Fourier in the early nineteenth century. (This is the same Fourier who gave us Fourier analysis.) In the worst case, every elimination step divides the number of equations in half and then squares it, resulting in doubly exponential behavior. The procedure works well in practice, though, since in many applications each variable is contained in only a few equations. (There are obvious heuristics, like choosing a variable at each stage that minimizes the number of equations at the next stage.) There is an implementation of the procedure in the file *FourierMotzkin.lean* in the *Examples* folder, modulo two components that we ask you to supply. SMT solvers use much more efficient methods based on the simplex algorithm from linear programming.

What does this theorem have to do with logic? Suppose the variables are labeled $x_1, x_2, \ldots, x_n$ and the constraints are labeled $c_1, c_2, \ldots, c_m$. Then what we are really asking as to whether the formula $\exists x_1, \ldots, x_n. c_1 \wedge c_2 \wedge \cdots \wedge c_m$ is true of the real numbers when the constraints are interpreted in the expected way. To make this more precise, consider the structure $(\mathbb{R}, 0, 1, +, <)$ in a language with symbols 0, 1, $+$, and $<$. All the constraints can be expressed in this language, albeit in a clunky way. For example, we can write $3x$ as $x + x + x$, and express a constraint like $x - (1/2)y + (4/3)z < 0$ as $6x + 8z < 3y$. A slight expansion of the proof of the theorem above yields the following:

Theorem

The question as to whether a sentence A is true in $(\mathbb{R}, 0, 1, +, <)$ is decidable.

We will only sketch the details here. The algorithm uses an important method known as "elimination of quantifiers." The idea is to successively eliminate quantifiers, one by one, until we are left with a quantifier-free sentence. We can determine the truth of that by simply calculating.

We will show that any formula $\exists x.\, A$, where A is quantifier-free, is equivalent to a quantifier-free formula A' that does not include x. Repeating the process and using the fact that $\forall x.\, A$ is equivalent to $\neg\exists x.\, \neg A$, we can eliminate all the quantifiers.

Given a formula $\exists x.\, A$, put A into disjunctive normal form. (We are not worrying about efficiency now, only trying to establish decidability in principle.) We can replace $s \not< t$ by $t < s \vee s = t$, and we can replace $s \neq t$ by $s < t \vee t < s$. Putting the result into disjunctive normal form again, we can assume that all the atomic formulas are of the form $s < t$ or $s = t$.

If we write A as $A_1 \vee A_2 \vee \cdots \vee A_n$, then $\exists x.\, A$ is equivalent to $(\exists x.\, A_1) \vee (\exists x.\, A_2) \vee \cdots \vee (\exists x.\, A_n)$. So we only need to show how to eliminate an existential quantifier from a conjunction of constraints of the form $s < t$ or $s = t$. But that is exactly what the proof of the first theorem in this section does, so we are done.

It is possible to write down axioms that justify every step of the transformation. The resulting set of axioms is known as the theory of *linear arithmetic*. The argument shows that the resulting set of axioms characterizes the structure exactly, and that the question of provability from those axioms is decidable.

Interestingly, the theorem remains true if we add multiplication. The resulting theory is known as the theory of *real closed fields*. The proof is much harder, however. The theorem was proved by Alfred Tarski before World War II, but it wasn't published until 1948, after the war.

12.2 Linear integer arithmetic

What happens if we replace the real numbers by the integers? It turns out that truth in the structure $(\mathbb{Z}, 0, 1, +)$ is also decidable. This was established in 1926 by Mojżesz Presburger, a student of Tarski's, who later died in the Holocaust. (The story has it that Tarski did not think the result was enough for a dissertation, and made him do more work.) The resulting theorem is known as *Presburger arithmetic* or *linear integer arithmetic*. In contrast to the reals, the order on the integers is *discrete*, since there is nothing between a value x and $x + 1$. The decision procedure is more complicated than that for linear arithmetic, and we will not discuss it here. SMT solvers, however, use efficient implementations of the *existential fragment* of the theory, which is to say, the satisfiability problem for quantifier-free formulas.

In contrast to the case with the real numbers, however, the result is false if we add multiplication. In other words, truth in the model $(\mathbb{Z}, 0, 1, +, \times)$ is undecidable. This follows from the methods that Gödel used to prove the incompleteness theorems, and it is also a consequence of *Tarski's theorem* on the undefinability of truth.

12.3 Equality

Fix a language, L. We will consider equations $s = t$ and *disequations* $s \neq t$ between closed terms. The fact that we are considering closed terms mean that there are no variables to substitute for; computer scientists sometimes call these *ground* terms. (As with unification, in some contexts we may want to treat some variables as constant. What is important here is not whether we call them variables or constants, but, rather, the fact that we are not considering substitutions.)

The problem we are addressing here is this: given a set of equations and disequations, is it satisfiable? Note that the word "satisfiable" is used in a different sense than it was used in the previous two sections. In those sections, we are interested in whether a set of formulas is satisfied by a variable assignment in a *particular* model, namely, the reals and the integers, respectively. Here we are asking whether a set of sentences is satisfied by any model.

For example, consider the following set of sentences:

1. $f(a, a) = b$

2. $g(c, a) = c$
3. $g(c, f(a, a)) = f(g(c, a), g(c, a))$
4. $f(c, c) \neq g(c, b)$

Is it satisfiable?

Before we answer that, let's make some general observations. A set that only has equations and no disequations is easily satisfiable, namely, in a model with a single element, where every expression is equal to every other one. Similarly, a set that only has disequations is easily satisfiable, unless one of the disequations is of the form $t \neq t$. For that purpose, we can use the term model, where every term is interpreted as itself. The interesting cases fall in between these two extremes, where the equations and disequations balance one another.

Coming back to the question, the following considerations show that the answer is "no." Each of the following is a consequence of the equations above:

5. $g(c, f(a, a)) = g(c, b)$ from 1
6. $f(g(c, a), g(c, a)) = f(c, c)$ from 2
7. $f(c, c) = g(c, b)$ from 3, 5, and 6.

This contradicts the disequation 4 above. To understand what is going on, it is helpful to think of f as addition, g as multiplication, a as the number 1, and b as the number 2. But the argument is fully abstract, and shows that the disequation cannot hold in any model in which all the equations are satisfied.

These considerations encapsulate the main ideas behind the proof of the following theorem:

Theorem

The question as to whether a finite set of ground equations and disequations is satisfiable is decidable.

The idea behind the proof is to use a *saturation* argument: starting from the equations in question, we derive new equations until no more equations are derivable. If we manage to contradict one of the disequations, the original set is not satisfiable. In the case where no contradiction is found, we will argue that the original set is satisfiable.

To make all this precise, we need a set of rules for deriving equations.

$$t = t \qquad \frac{s = t}{t = s} \qquad \frac{r = s \qquad s = t}{r = s} \qquad \frac{s_1 = t_1 \qquad \ldots \qquad s_n = t_n}{f(s_1, \ldots, s_n) = f(t_1, \ldots, t_n)}$$

The first three rules express the reflexivity, symmmetry, and transitivity of equality, respectively. The last rule is called the *congruence* rule. You should convince yourself that using these rules we can derive

$$\frac{r = s}{t[r/x] = t[s/x]}$$

for any terms r, s, and t and variable x. If we add relation symbols and atomic formulas, we would add the following rule:

$$\frac{s_1 = t_1 \qquad \ldots \qquad s_n = t_n \qquad R(s_1, \ldots, s_n)}{R(t_1, \ldots, t_n)}$$

Returning to our proof plan, we want to show that if applying these rules successively does not result in a contradiction, then there is a model in which the original equations and disequations are all true. But a problem arises: what if the original set contains an equation $a = f(a)$? Then our algorithm falls into an infinite loop, deriving $a = f(a) = f(f(a)) = f(f(f(a))) = \ldots$. The solution is to restrict attention to *subterms* of terms appearing in the original equations and disequations. The theorem follows from the following lemma.

Lemma

Let Γ consist of a set of equations and disequations. Let S be the set of subterms of all the terms occurring in Γ. Let Γ' be the set of al equations between elements of S that can be derived from the equations in Γ using the rules above. Then Γ is satisfiable if and only if no disequation in Γ is the negation of an equation in Γ'.

The algorithm implicit in this lemma is called *congruence closure*. We will see that it can be implemented efficiently (and *is* implemented efficiently in SMT solvers) using *union-find* data structures.

Proof

One direction of the lemma is easy. Since the equational rules preserve truth in any model, if we can derive a contradiction from the equations and disequations in Γ, then Γ is unsatisfiable. The other direction is harder. Since there are only finitely many pairs of terms in S, the algorithm necessarily terminates. We need to show that if it terminates without deriving a contradiction, then there is a model that satisfies Γ.

Say two elements s and t are *equivalent*, written $s \equiv t$, if they are proved equal from the equations in Γ. The rules guarantee that this is an *equivalence relation*, which is to say, it is reflexive, symmetric, and transitive. It is also a *congruence*, which means that applying a function symbol to equivalent terms results in equivalent terms.

To each element t, we associate its *equivalence class* $[t]$, defined by

$$[t] = \{s \in S \mid s \equiv t\}.$$

In words, $[t]$ is the set of terms equivalent to t. Assuming the algorithm terminates without a contradiction, define a model $\mathfrak{M}$ whose universe consists of all the equivalence classes of elements of S together with a new element, $\star$. For elements $t_1, \ldots t_n$ in S, interpret each n-ary function symbol f by function

$$f^{\mathfrak{M}}([t_1], \ldots, [t_n]) = \begin{cases} [f(t_1, \ldots, t_n)] & \text{if } f(t_1, \ldots, t_n) \text{ is in } S \\ \star & \text{otherwise} \end{cases}$$

In other words, what $f^{\mathfrak{M}}$ does to each equivalence class is determined by what f does to each of the elements. The fact that $\equiv$ is a congruence ensures that this makes sense. This is just a truncated version of the term model, in which provably equal terms are all glued together.

It is not hard to show that for every term t in S, $[\![t]\!]_{\mathfrak{M}}$ is equal to $[t]$. But this is what we need. For every equation $s = t$ in Γ, s and t are in the same equivalence class, so they are equal in the model. And if s and t are not provably equal, then $[s]$ and $[t]$ are not the same, so every disequation $s \neq t$ in Γ is true in $\mathfrak{M}$ as well.

For examples of the algorithm in action, first let us show that the set

$$f^3(a) = a,\ f^5(a) = a,\ f(a) \neq a$$

is unsatisfiable, where $f^n(a)$ abbreviates n-fold application $f(f(\cdots f(a)))$. The set of all subterms is

$$a,\ f(a),\ f^2(a),\ f^3(a),\ f^4(a),\ f^5(a).$$

We start with the equivalence classes $\{a, f^3(a)\}$ and $\{a, f^5(a)\}$ as well as all the others subterms in singleton sets. From $a = f^3(a)$ we derive $f(a) = f^4(a)$ by congruence, giving rise to the set $\{f(a), f^4(a)\}$. Applying congruence again gives rise to the set $\{f^2(a), f^5(a)\}$, which is merged with $\{a, f^5(a)\}$ to yield $\{a, f^2(a), f^5(a)\}$. Applying congruence again yields $\{f(a), f^3(a)\}$. (We ignore the term $f^6(a)$.) This is merged with the set $\{a, f^3(a)\}$ to yield $\{a, f(a), f^3(a)\}$. Applying congruence again yields $\{f(a), f^2(a), f^4(a)\}$, which is merged with $\{a, f(a), f^3(a)\}$ and $\{f^2(a), f^5(a)\}$ to yield $\{a, f(a), f^2(a), f^3(a), f^5(a)\}$. At this point, we have derived $f(a) = a$, contradicting the disequality in the original set. So the set is unsatisfiable.

Suppose we start instead with the set

$$f^2(a) = a,\ f^4(a) = a,\ f(a) \neq a,\ f(a) \neq b$$

You can check that in this case, the algorithm terminates with the following three equivalence classes:

- $[a] = \{a, f^2(a), f^4(a)\}$
- $[f(a)] = \{f(a), f^3(a)\}$
- $[b] = \{b\}$.

We now construct a model $\mathfrak{M}$ with these elements and an additional element $\star$, with

$$\begin{aligned} f^{\mathfrak{M}}([a]) &= [f(a)] \\ f^{\mathfrak{M}}([f(a)]) &= [a] \\ f^{\mathfrak{M}}([b]) &= \star \\ f^{\mathfrak{M}}(\star) &= \star \end{aligned}$$

You can check that this satisfies the original set of equations and disequations.

Suppose we allow atomic formulas $R(t_1, \ldots, t_n)$ and negated atomic formulas in Γ. To test satisfiability, we do not have to change much. Using the congruence rule for relations, whenever we have derived $R(s_1, \ldots, s_n)$ and $s_i = t_i$ for every i, we can conclude $R(t_1, \ldots, t_n)$. The algorithm terminates when we contradict a disequality or another negated atomic formula. If the algorithm terminates without a contradiction, we build a model as before, where we simply declare that $R^{\mathfrak{M}}([t_1], \ldots, [t_n])$ holds if and only if we have determined that $R(t_1, \ldots, t_n)$ in a consequence of the original set.

Now suppose we are given an existential sentence $\exists x_1, \ldots, x_n. A$ where A is quantifier-free, and suppose we want to determine whether it is satisfiable. Replace $x_1, \ldots, x_n$ in A with new constants $c_1, \ldots, c_n$. The resulting quantifier-free sentence is satisfiable if and only if the existential one is. Put that sentence in DNF, and use the fact $A_1 \vee \cdots \vee A_n$ is satisfiable if and only if one of the sentences A_i is. That reduces the task to determining whether a conjunction of literals is satisfiable, and we have just explained how to do that.

Since a sentence is valid if and only if its negation is satisfiable, and since the negation of a universal sentence is an existential sentence, we have shown the following.

Theorem

The satisfiability of an existential sentence in first-order logic is decidable. Equivalently, the validity of a universal sentence is decidable.

CHAPTER

THIRTEEN

USING SMT SOLVERS

Satisfiability Modulo Theories (SMT) solvers determine whether a quantifier-free first-order formula can be satisfied with respect to some background theories. In many application areas, problem instances can be transformed into SMT formulas and the SMT solver determines whether there exists a satisfying assignment. The effectiveness of an SMT solver depends on the selection of background theories for a given problem instance. Several strong SMT solvers have been developed, including Z3, CVC4, and Boolector. SMT solvers are frequently used in industry and academia.

13.1 SMT-LIB Format

The input format for SMT solvers is called SMT-LIB. SMT-LIB is much more readable than the DIMACS format for SAT solvers. For example, in SMT-LIB, variable names are strings, while DIMACS uses numbers. Also, whereas SAT solvers require the input to be in conjunctive normal form, this is not the case in SMT-LIB.

Most SMT-LIB input files consist of five blocks:

- Selecting the theory. Examples of theories are QF_UF (uninterpreted functions) and QF_LIA (linear integer arithmetic).
- Declaring variables, functions, and types (called sorts). To declare a variable, one uses a line of the form `(declare-const name type)`, where `name` is the variable name and `type` is the variable type. Functions can be declared/defined using `(declare-fun name (inputTypes) outputType)` for uninterpreted functions and using `(define-fun name (inputTypes) outputType (body))`, otherwise. In both cases `name` is the function name, `(inputTypes)` the input types, and `outputType` the output type. The `(body)` part defines the function. In a similar way, one can define types, but we won't use this functionality in this chapter. Several predefined types are supported depending on the selected theory. For example QF_UF supports Bool (propositional variables), while QF_LIA supports Int (integers).
- A list of constraints. Constraints in SMT-LIB are of the form `(assert ...)` with `...` describing the constraint.
- The command `(check-sat)` solves the formula encoded above it. Depending on the result, one can then use `(get-model)` to obtain a model when the formula is satisfiable or `(get-unsat-core)` to extract an unsatisfiable core (a subset of the constraints that is also unsatisfiable) when the formula is unsatisfiable.
- Finally, the command `(exit)` terminates the solver.

The example formula shown below uses the theory QF_UF and asks whether the formula $p \wedge \neg p$ can be satisfied. For each blocked described above, only a single line appears in the formula.

```
(set-logic QF_UF)
(declare-const p Bool)
(assert (and p (not p)))
(check-sat)
(exit)
```

Another small example is shown below. It uses the theory QF_LIA and asks whether there exists an integer `x` that is larger than an integer `y` without further constraining these variables. This formula is satisfiable, so we can ask the SMT solver to provide us an example `x` and `y` which makes the formula true using `(get-model)`. For example, the solver can return that `x` is 5 and `y` is 2.

```
(set-logic QF_LIA)
(declare-const x Int)
(declare-const y Int)
(assert (> x y))
(check-sat)
(get-model)
(exit)
```

13.2 Example: Magic squares

A magic square of order n is a $n \times n$ grid with the numbers 1 to n^2 occurring exactly once such that the sum for each row, column, and the two main diagonals is the same. Two magic squares are shown below, one of order 3 and one of order 8.

4	9	2
3	5	7
8	1	6

61	3	2	64	57	7	6	60
12	54	55	9	16	50	51	13
20	46	47	17	24	42	43	21
37	27	26	40	33	31	30	36
29	35	34	32	25	39	38	28
44	22	23	41	48	18	19	45
52	14	15	49	56	10	11	53
5	59	58	8	1	63	62	4

It is relatively straightforward to encode the existence of a magic square as an SMT formula. After stating the theory, the encoding consists of four parts: 1) declaring a variable for each cell in the grid; 2) enforcing that each cell has a value from 1 to n^2; 3) enforcing that all each cell has a unique value; and 4) enforcing that the sum of each row, column, and main diagonals is equal to $(n^3 + n)/2$.

The formula shown below uses the quantifier-free theory of linear integer arithmetic (QF_LIA). The variable for the cell in row `i` and column `j` is called `m_i_j`. The variable is declared using `(declare-const m_i_j Int)`, while the lines with `assert` constrain the variables.

```
(set-logic QF_LIA)
(declare-const m_0_0 Int)
(declare-const m_0_1 Int)
(declare-const m_0_2 Int)
(declare-const m_1_0 Int)
(declare-const m_1_1 Int)
(declare-const m_1_2 Int)
(declare-const m_2_0 Int)
(declare-const m_2_1 Int)
(declare-const m_2_2 Int)
```

(continues on next page)

(continued from previous page)

```
(assert (and (> m_0_0 0) (<= m_0_0 9)))
(assert (and (> m_0_1 0) (<= m_0_1 9)))
(assert (and (> m_0_2 0) (<= m_0_2 9)))
(assert (and (> m_1_0 0) (<= m_1_0 9)))
(assert (and (> m_1_1 0) (<= m_1_1 9)))
(assert (and (> m_1_2 0) (<= m_1_2 9)))
(assert (and (> m_2_0 0) (<= m_2_0 9)))
(assert (and (> m_2_1 0) (<= m_2_1 9)))
(assert (and (> m_2_2 0) (<= m_2_2 9)))
(assert (distinct m_0_0 m_0_1 m_0_2 m_1_0 m_1_1 m_1_2 m_2_0 m_2_1 m_2_2))
(assert (= 15 (+ m_0_0 m_0_1 m_0_2)))
(assert (= 15 (+ m_1_0 m_1_1 m_1_2)))
(assert (= 15 (+ m_2_0 m_2_1 m_2_2)))
(assert (= 15 (+ m_0_0 m_1_0 m_2_0)))
(assert (= 15 (+ m_0_1 m_1_1 m_2_1)))
(assert (= 15 (+ m_0_2 m_1_2 m_2_2)))
(assert (= 15 (+ m_0_2 m_1_1 m_2_0)))
(assert (= 15 (+ m_2_0 m_1_1 m_0_2)))
(check-sat)
(get-model)
(exit)
```

The encoding in the quantifier-free theory of bitvectors (QF_BV) is very similar and shown below. When using bitvectors, one needs to declare the number of bits. In this example, we use 16 bits, which is large enough to compute magic squares of reasonable size. Note that QF_BV uses the bitvector variants of +, -, >, >=, <, and <=, which are `bvadd`, `bvsub`, `bvugt`, `bvuge`, `bvult`, and `bvule`, respectively. Also, constants are expressed differently in QF_BV: they are written as `#x` followed by the bitvector in hexadecimal notation.

```
(set-logic QF_BV)
(declare-const m_0_0 (_ BitVec 16))
(declare-const m_0_1 (_ BitVec 16))
(declare-const m_0_2 (_ BitVec 16))
(declare-const m_1_0 (_ BitVec 16))
(declare-const m_1_1 (_ BitVec 16))
(declare-const m_1_2 (_ BitVec 16))
(declare-const m_2_0 (_ BitVec 16))
(declare-const m_2_1 (_ BitVec 16))
(declare-const m_2_2 (_ BitVec 16))
(assert (and (bvugt m_0_0 #x0000) (bvule m_0_0 #x0009)))
(assert (and (bvugt m_0_1 #x0000) (bvule m_0_1 #x0009)))
(assert (and (bvugt m_0_2 #x0000) (bvule m_0_2 #x0009)))
(assert (and (bvugt m_1_0 #x0000) (bvule m_1_0 #x0009)))
(assert (and (bvugt m_1_1 #x0000) (bvule m_1_1 #x0009)))
(assert (and (bvugt m_1_2 #x0000) (bvule m_1_2 #x0009)))
(assert (and (bvugt m_2_0 #x0000) (bvule m_2_0 #x0009)))
(assert (and (bvugt m_2_1 #x0000) (bvule m_2_1 #x0009)))
(assert (and (bvugt m_2_2 #x0000) (bvule m_2_2 #x0009)))
(assert (distinct m_0_0 m_0_1 m_0_2 m_1_0 m_1_1 m_1_2 m_2_0 m_2_1 m_2_2))
(assert (= #x000f (bvadd m_0_0 m_0_1 m_0_2)))
(assert (= #x000f (bvadd m_1_0 m_1_1 m_1_2)))
(assert (= #x000f (bvadd m_2_0 m_2_1 m_2_2)))
(assert (= #x000f (bvadd m_0_0 m_1_0 m_2_0)))
(assert (= #x000f (bvadd m_0_1 m_1_1 m_2_1)))
(assert (= #x000f (bvadd m_0_2 m_1_2 m_2_2)))
(assert (= #x000f (bvadd m_0_2 m_1_1 m_2_0)))
(assert (= #x000f (bvadd m_2_0 m_1_1 m_0_2)))
```

(continues on next page)

(continued from previous page)

```
(check-sat)
(get-model)
(exit)
```

Although the encodings of magic squares in QF_LIA and QF_BV look very similar, the performance of SMT solvers on these encodings differs a lot. For example, computing a magic square of order 5 is difficult for SMT solvers using the QF_LIA encoding, while it is easy when using the QF_BV encoding. In fact, it is even easy to solve the QF_BV encoding expressing the existence of a magic square of order 10. The main difference between these two theories is that the solver applies linear arithmetic when using QF_LIA, while it applies what is known as *bit blasting* when using QF_BV. Bit blasting transforms the formula into propositional logic by introducing a propositional variable for each bit in the problem. For some problems, such as magic squares, bit blasting can be very effective. For other problems, bit blasting can result in formulas that are hard to solve.

13.3 Calling SMT solvers from Lean

To use an SMT solver, you can simply create an SMT-LIB file and run the solver on it. We also provide a convenient interface for calling any of the three popular solvers Z3, CVC4, and CVC5 from Lean. There is an example in the file `magicSquares.lean` in `Examples/using_smt_solvers`.

To start with, we provide syntax with brackets `sexp!{` and `}` for writing s-expressions, and you can use the notation `{ t }` inside an s-expression to fill in the value of a Lean expression `t`. For example the following declares constants `m_{i}_{j}` as `i` and `j` range over values less than `n`.

```
for i in [:n] do
  for j in [:n] do
    consts := consts.push sexp!{(declare-const {s!"m_{i}_{j}"} (_ BitVec 32))}
```

The following declares that each cell is nonzero:

```
for i in [:n] do
  for j in [:n] do
    asserts := asserts.push sexp!{(bvugt {s!"m_{i}_{j}"} {toBVConst 32 0})}
```

You can use the syntax `...{ }` to splice a list of s-expressions into an s-expression. For example. `(foo ...{List.range 3 |>.map (toString ·)})` becomes `(foo 0 1 2)`.

The following wraps all the statements into an assert, and packages them into SMT-LIB format.

```
asserts := asserts.map fun c => sexp!{(assert {c})}

-- Use the theory of quantifier-free bitvector expressions, and find a model if SAT
sexps!{
  (set-logic QF_BV)
  (set-option :produce-models true)
  ...{(consts ++ asserts).toList}
  (check-sat)
  (get-model)
}
```

The preceding snippets are all part of a function `magicSquareToBvSmt` that, for each value of n, assembles the constraints into a list of SMT-LIB commands that ask for a model of an $n \times n$ magic square. The following calls the SMT solver CVC5 on the resulting formula:

```
#eval (do
  let cmds := magicSquareToBvSmt 3
  -- Set `verbose := false` to hide SMT-LIB communications
  let out ← callCvc5 cmds (verbose := true)
  match out with
  | Sexp.atom "sat" :: m :: _ =>
    IO.println "SAT with assignment:"
    for (x, b) in decodeModelConsts m do
      IO.println s!"{x} ↦ {evalNumConst b |>.get!}"
    IO.println "\nSquare:"
    printMagicSquare 3 m
  | ss =>
    IO.println "Not SAT. Solver output:"
    IO.println ss

  : IO Unit)
```

The function `printMagicSquare` shows the result. You can change `Cvc5` to `Cvc4` or `Z3`, as long as the relevant solver is in `LAMR/bin`.

13.4 Application: Verification

SMT solvers are frequently used for verification tasks. In software verification, SMT solvers can be used to validate whether some optimized code is functionally equivalent to some straightforward code (the specification). For example, consider the C code below, which efficiently computes the number of bits that are set to true in an unsigned integer (32-bit). This code is significantly more efficient compared to looping over the bits to perform the counting.

```
int popCount32 (unsigned int x) {
    x = x - ((x >> 1) & 0x55555555);
    x = (x & 0x33333333) + ((x >> 2) & 0x33333333);
    x = ((x + (x >> 4) & 0xf0f0f0f) * 0x1010101) >> 24;
    return x;
}
```

Validating the correctness of the above procedure can be done efficiently using the small SMT-LIB file shown below. After selecting the theory QF_BV, the file starts with declaring a single 32-bit bitvector `x`. For each line in the C code, a function is defined with 32-bit bitvectors as input and as output. Additionally, the specification function is declared as well. Each line in that function extracts a single bit from the bitvector and checks whether it is set to true (`#b1`). In that case it increases the count by 1 (shown as a 32-bit bitvector).

At the end of the SMT-LIB file, there is a single constraint. To check whether the two implementations are equivalent, it asks whether there exists an `x` such that the implementations produce a different result. If that formula is satisfiable, then we found a counterexample to the equivalence. If the formula is unsatisfiable, then the implementations are equivalent.

```
(set-logic QF_BV)
(declare-const x (_ BitVec 32))

(define-fun pcLine1 ((x (_ BitVec 32))) (_ BitVec 32)
    (bvsub x (bvand (bvlshr x #x00000001) #x55555555)))

(define-fun pcLine2 ((x (_ BitVec 32))) (_ BitVec 32)
    (bvadd (bvand x #x33333333) (bvand (bvlshr x #x00000002) #x33333333)))

(define-fun pcLine3 ((x (_ BitVec 32))) (_ BitVec 32)
```

(continues on next page)

(continued from previous page)

```
    (bvlshr (bvmul (bvand (bvadd (bvlshr x #x00000004) x) #x0f0f0f0f) #x01010101)
                   #x00000018))

(define-fun popCount32 ((x (_ BitVec 32))) (_ BitVec 32)
     (bvadd (ite (= #b1 ((_ extract  0  0) x)) #x00000001 #x00000000)
            (ite (= #b1 ((_ extract  1  1) x)) #x00000001 #x00000000)
            (ite (= #b1 ((_ extract  2  2) x)) #x00000001 #x00000000)
            ...
            (ite (= #b1 ((_ extract 30 30) x)) #x00000001 #x00000000)
            (ite (= #b1 ((_ extract 31 31) x)) #x00000001 #x00000000)))

(assert (not (= (pcLine3 (pcLine2 (pcLine1 x))) (popCount32 x))))
(check-sat)
(exit)
```

This formula can be solved in about a second. You can find the example implemented in the file `popCount.lean`. Note that this approach is significantly faster than any implementation that explores the entire search space of 2^{32} possible inputs. Try changing any of the bitvector parameters, and you will see that the SMT solvers finds a counterexample to the equivalence instantaneously.

13.5 Exercise: Almost squares

The almost square of order n is a rectangle of size $n \times (n+1)$. The almost squares of orders 1 to 3 can fully cover the almost square of order 4. A solution is shown below.

1	1	3	3	3
2	2	3	3	3
2	2	3	3	3
2	2	3	3	3

In this exercise, we are going to encode whether the almost squares of order 1 to n can fully cover the almost of order m. The encoding uses the QF_LIA theory. The encoding uses $4n$ variables: for the almost square of order `i`, we use variables `xmin_i`, `xmax_i`, `ymin_i`, and `ymax_i`. The variable `xmin_i` (`xmax_i`) denotes the first (last, respectively) row in which the almost square of order `i` is placed. Similarly, the variable `ymin_i` (`ymax_i`) denotes the first (last, respectively) column in which the almost square of order `i` is placed.

The covering of the almost square of order 4 shown above can be expressed using the following assignment to these variables:

- `xmin_1 = 1, xmax_1 = 2, ymin_1 = 4, ymax_1 = 4`
- `xmin_2 = 1, xmax_2 = 2, ymin_2 = 1, ymax_2 = 3`
- `xmin_3 = 3, xmax_3 = 5, ymin_3 = 1, ymax_3 = 4`

The code fragment below shows the first part of the encoding used to compute the covering. It shows the declaration of the first variables and the first constraints on those variables.

```
(set-logic QF_LIA)
(declare-const xmin_1 Int)
(declare-const xmax_1 Int)
(declare-const ymin_1 Int)
(declare-const ymax_1 Int)
...
```

(continues on next page)

(continued from previous page)

```
(assert (and (>= xmin_1 1) (<= xmax_1 5)))
(assert (and (>= ymin_1 1) (<= ymax_1 4)))
...
```

Finish the encoding use the following steps:

Step 1) Express as the constraints that ensure that the almost square of order `i` covers exactly a subgrid of $n \times (n+1)$ or $(n+1) \times n$. The only variables that you can use are `xmin_i`, `xmax_i`, `ymin_i`, and `ymax_i`. Hint: Split the constraint into three parts with one part that enforces the relation between `xmin_i` and `xmax_i`, one part that enforces the relation between `ymin_i` and `ymax_i`, and one part that enforces the relation between all four variables.

Step 2) For each pair of almost squares, express the constraint that they cannot overlap each other, i.e., there is no cell that is covered by multiple almost squares.

Step 3) Determine a grid assignment showing that the almost squares of orders 1 to 8 can fully cover the almost square of order 15. SMT solvers should be able to quickly solve the intended encoding. The same encoding can also be used to cover the almost square of order 55 with the almost squares of order 1 to 20. Solving this formula can take minutes.

Step 4) Encode the same problem using the theory QF_BV and compare the runtimes between the two theories.

CHAPTER

FOURTEEN

DEDUCTION FOR FIRST-ORDER LOGIC

The fundamental difference between propositional logic and first-order logic is that in first-order logic there are variables and terms that stand for objects, and we can form atomic propositions that depend on those objects. Within the framework, the two key ingredients are equality and the quantifiers. Deduction systems for first-order logic have to extend those for propositional logic by providing rules for these.

As was the case for propositional logic, all the deductive systems we describe in this section are sound and complete for first-order logic, though we do not provide detailed proofs here.

14.1 Axiomatic systems

We have already discussed equational reasoning in Section 12.3. We have seen that the natural rules for equality are given by reflexivity, symmetry, transitivity, and congruence with respect to functions and relations. These can be expressed as rules, but also as first-order axioms:

- $\forall x.\, x = x$
- $\forall x, y.\, x = y \to y = x$
- $\forall x, y, z.\, x = y \wedge y = z \to x = z$
- $\forall x_1, \ldots, x_n, y_1, \ldots, y_n.\, x_1 = y_1 \wedge \cdots x_n = y_n \to f(x_1, \ldots, x_n) = f(y_1, \ldots, y_n)$
- $\forall x_1, \ldots, x_n, y_1, \ldots, y_n.\, x_1 = y_1 \wedge \cdots x_n = y_n \wedge R(x_1, \ldots, x_n) \to R(y_1, \ldots, y_n)$.

From these, we can derive substitution for terms and formulas:

- $\forall x, y.\, x = y \to t(x) = t(y)$
- $\forall x, y.\, x = y \wedge A(x) \to A(y)$

Sometimes these are taken as axioms instead of congruence. Remember that we are adopting the convention what when we use notation like $t(x)$, we have in mind a certain variable z that t might depend on and we take $t(x)$ to stand for $t[x/z]$ and $t(y)$ to stand for $t[y/z]$. Similar conventions hold for formula $A(x)$.

The axioms for the quantifiers follow naturally from their meanings:

- $(\forall x.\, A) \to A[t/x]$
- $A[t/x] \to \exists x.\, A$.

The first says that if A holds of everything, then it holds of any particular thing, and the second says that if A holds of any particular thing, then it holds of something.

These axioms are only half the story, though. The first axiom tells us how to *use* a statement that starts with a universal quantifier but not how to *prove* it, and the second one tells us how to prove a statement with an existential quantifier but not how to use it. How do we prove $\forall x.\, A$? Establishing $\forall x.\, A$ involves showing that A holds of an arbitrary value of x. To do that, we let x be arbitrary, and prove A. This suggests the following rule of generalization:

From A, conclude $\forall x.\, A$.

The actual rule we use is a generalization of this: if we have shown that A follows from B, and B doesn't say anything about x, then we have shown that B implies that A holds for any x at all.

- From $B \to A$ conclude $B \to \forall x.\, A$, assuming x is not free in B.

The dual rule for the existential quantifier says that if B follows from the assumption that A holds of some x, then B follows from the assumption that there exists an x satisfying B.

- From $A \to B$ conclude $(\exists x.\, A) \to B$, assuming x is not free in B.

14.2 A sequent calculus

We can also extend the cut-free sequent calculus described in Section 8.2 to first-order logic. Remember that the system derives sets of formulas Γ in negation-normal form. We interpret a proof as telling us that in any model, and with any assignment to the free variables, at least one of the formula in Γ is true. In the case of propositional logic, we took the axioms to be finite sets Γ that contain a complementary pair of atoms P and $\neg P$. In the presence of equality, we now take the axioms to be sets Γ such that the set of negations of those formulas can be refuted using equational reasoning. For example, $\neg P(a), a \neq b, P(b)$ is an axiom, because equational reasoning refutes the set $P(a), a = b, \neg P(b)$. We can determine whether a finite set of formulas is an axiom using congruence closure.

The other rules of the system are as follows:

$$\frac{\Gamma, A \qquad \Gamma, B}{\Gamma, A \wedge B} \qquad \frac{\Gamma, A, B}{\Gamma, A \vee B}$$

$$\frac{\Gamma, A}{\Gamma, \forall x.\, A} \qquad \frac{\Gamma, A[t/x]}{\Gamma, \exists x.\, A}$$

In the rule for the universal quantifier, we require that x is not free in any formula in Γ. This is analogous to the requirement in the previous section that x is not free in the formula B in the generalization rule. You can think of it as saying that x is really arbitrary.

As in Section 8.2, we can add the cut rule, which represents a form of modus ponents. In class, we will sketch a proof that this system is complete even without the cut rule.

14.3 Resolution

14.4 Natural deduction

CHAPTER

FIFTEEN

USING FIRST-ORDER THEOREM PROVERS

Given a set of hypotheses and a conclusion, a first-order theorem prover does its best to find a proof of the conclusion from the hypotheses, using first-order logic with equality. If it finds one, it reports success. If the conclusion does not follow from the hypotheses, in some situations, the prover can detect that fact and report it. But that is the exception rather than the rule. In many situations, the prover will simply burn CPU cycles and gobble up memory, and, if we are lucky, eventually time out gracefully.

Let's not blame the provers. They are fighting a brave battle against incompleteness and undecidability, and the fact that there are no a priori guarantees that a search will be successful makes the quest all the more exciting. In this chapter, we show you how you can call a theorem prover from within Lean.

15.1 Example: Aunt Agatha

The following story is taken from a talk by Peter Baumgartner.

> Someone who lives in Dreadbury Mansion killed Aunt Agatha. Agatha, the butler, and Charles live in Dreadbury Mansion, and are the only people who live therein. A killer always hates his victim, and is never richer than his victim. Charles hates no one that Aunt Agatha hates. Agatha hates everyone except the butler. The butler hates everyone not richer than Aunt Agatha. The butler hates everyone Aunt Agatha hates. No one hates everyone. Agatha is not the butler.
>
> Who killed Aunt Agatha?

We can represent the hypotheses as follows.

```
def aunt_agatha_hypotheses : List FOForm := [
  fo!{∃ x. lives_at_dreadbury(%x) ∧ killed(%x, agatha)},
  fo!{∀ x. lives_at_dreadbury(%x) ↔ (%x = agatha ∨ %x = butler ∨ %x = charles)},
  fo!{∀ x. ∀ y. killed(%x, %y) → hates(%x, %y)},
  fo!{∀ x. ∀ y. killed(%x, %y) → ¬ richer(%x, %y)},
  fo!{∀ x. hates(charles, %x) → ¬ hates(agatha, %x)},
  fo!{∀ x. ¬ hates(agatha, %x) ↔ %x = butler},
  fo!{∀ x. ¬ richer(%x, agatha) → hates(butler, %x)},
  fo!{∀ x. hates(agatha, %x) → hates(butler, %x)},
  fo!{∀ x. ∃ y. ¬ hates(%x, %y)},
  fo!{¬ agatha = butler}
]
```

Our first guess, of course, is that the butler did it. Here we call Vampire to test that assumption.

```
def aunt_agatha_guess :=
  fo!{killed(butler, agatha)}
```

(continues on next page)

(continued from previous page)

```
-- Termination reason: Satisfiable
#eval (do
  discard <| callVampireTptp aunt_agatha_hypotheses aunt_agatha_guess (verbose :=␣
↪true)
  : IO Unit)
```

Like resolution provers for propositional logic, first-order provers generally work by negating the conclusion, adding it to the hypotheses, and trying to prove a contradiction. In this case, Vampire runs for a while, and ultimately reports that the entailment is not valid: the hypotheses together with the negation of the guess are satisfiable. In fact, if we negate the guess, Vampire proves instantaneously that the butler is not the killer. We have no better luck with Charles, leaving us to conclude that this must be a case of suicide. Vampire confirms this:

```
def aunt_agatha_conclusion :=
  fo!{killed(agatha, agatha)}

-- Termination reason: Refutation
#eval (do
  discard <| callVampireTptp aunt_agatha_hypotheses aunt_agatha_conclusion (verbose␣
↪:= true)
  : IO Unit)
```

It is also worthwhile finding a proof on your own.

15.2 Example: The Eighth Asylum

In an article called "The Asylum of Doctor Tarr and Professor Fether," published in the *American Mathematical Monthly* and later in a collection *The Lady or the Tiger? and other Logic Puzzles*, Raymond Smullyan tells of an investigation of 11 insane asylums by Inspector Craig of Scotland Yard. In each of these asylums, every inhabitant is either a doctor or a patient, and every inhabitant is either sane or insane. The sane inhabitants are totally sane and the insane inhabitants are totally insane, in the following sense: for any proposition P, a sane inhabitant believes P if and only if P is true, and an insane inhabitant believes P if and only if P is false.

The eighth asylum is described as follows:

> The next asylum proved to be quite a puzzler, but Craig finally managed to get to the bottom of things. He found out that the following conditions prevailed:
>
> 1. Given any two inhabitants A and B, either A trusts B or he doesn't.
> 2. Some of the inhabitants are *teachers* of other inhabitants. Each inhabitant has at least one teacher.
> 3. No inhabitant A is willing to be a teacher of an inhabitant B unless A believes that B trusts himself.
> 4. For any inhabitant A there is an inhabitant B who trusts all and only those inhabitants who have at least one teacher who is trusted by A. (In other words, for any inhabitant X, B trusts X if A trusts some teacher of X, and B doesn't trust X unless A trusts some teacher of X.)
> 5. There is one inhabitant who trusts all the patients but does not trust any of the doctors.
>
> Inspector Craig thought this over for a long time and was finally able to prove that either one of the patients was sane or one of doctors was insane. Can you find the proof?

Given the instructions, we can represent the fact that x is insane with the formula $\neg \mathit{Sane}(x)$, and the statement that x is a patient with the formula $\neg \mathit{Doctor}(x)$. Moreover, any statement of the form "x believes P" is equivalent to $\mathit{Sane}(x) \leftrightarrow P$, because either x is sane and P holds or x is insane and P doesn't hold. We can therefore formalize the hypotheses as follows:

```
def asylum_eight_hypotheses : List FOForm := [
  fo!{∀ x. ∃ y. Teaches(%y, %x)},
  fo!{∀ x. ∀ y. Teaches(%x, %y) → (Sane(%x) ↔ Trusts(%y, %y))},
  fo!{∀ x. ∃ y. ∀ z. Trusts(%y, %z) ↔ ∃ w. Teaches(%w, %z) ∧ Trusts(%x, %w)},
  fo!{∃ x. ∀ y. ¬ Doctor(%y) ↔ Trusts(%x, %y)}
]
```

It then takes Lean only a few seconds to draw the relevant conclusion.

```
def asylum_eight_conclusion :=
  fo!{∃ x. Doctor(%x) ↔ ¬ Sane(%x)}

#eval (do
  discard <| callVampireTptp asylum_eight_hypotheses asylum_eight_conclusion
    (verbose := true)
  : IO Unit)
```

Once again, it is worthwhile to find a proof by hand. You can use Vampire to check some of your conclusions along the way.

15.3 Exercise: The Last Asylum

The last puzzle in the chapter reads as follows:

> The last asylum Craig visited he found to the be the most bizarre of all. This asylum was run by two doctors named Doctor Tarr and Professor Fether. There were other doctors on the staff as well. Now, an inhabitant was called *peculiar* if he believed that he was a patient. An inhabitant was called *special* if all patients believed he was peculiar and no doctor believed he was peculiar. Inspector Craig found out that at least one inhabitant was sane and that the following condition held:
>
> *Condition C:* Each inhabitant had a best friend in the asylum. Moreover, given any two inhabitants, A and B, if A believed that B was special, then A's best friend believed that B was a patient.
>
> Shortly after this discovery, Inspector Craig had private interviews with Doctor Tarr and Professor Fether. Here is the interview with Doctor Tarr:
>
> *Craig:* Tell me, Doctor Tarr, are all the doctors in this asylum sane?
>
> *Tarr:* Of course they are!
>
> *Craig:* What about the patients? Are they all insane?
>
> *Tarr:* At least one of them is.
>
> The second answer struck Craig as a surprisingly modest claim! Of course, if all the patients are insane, the it certainly is true that at least one is. But why was Doctor Tarr being so cautious? Craig then had his interview with Professor Fether, which went as follows:
>
> *Craig:* Doctor Tarr said that at least one patient here is insane. Surely that is true, isn't it?
>
> *Professor Fether:* Of course it is true! All the patients in this asylum are insane! What kind of asylum do you think we are running?
>
> *Craig:* What about Doctor Tarr? Is he sane?
>
> *Professor Fether:* Of course he is! How dare you ask me such a question?
>
> At this point, Craig realized the full horror of the situation! What was it?

In the solutions, Smullyan provides a proof that, under these hypotheses, all the patients are sane and all the doctors are insane. Formalizing the hypotheses, we were able to use Vampire to show something stronger, namely, that the hypotheses themselves are inconsistent. In other words, no such asylum can possibly exist!

We encourage you to formalize the problem, and see if you can get Vampire to find a contradiction. We are grateful to Alexander Bentkamp and Seulkee Baek for working out pen-and-paper proofs. See if you can find one as well, perhaps using Smullyan's solution as a starting point.

CHAPTER

SIXTEEN

FIRST-ORDER LOGIC IN LEAN

16.1 Equational reasoning

16.2 Structural induction

A feature of working with a system like Lean, which is based on a formal logical foundation, is that you can not only define data types and functions but also prove things about them. The goal of this section is to give you a flavor of using Lean as a proof assistant. It isn't easy: Lean syntax is finicky and its error messages are often inscrutable. In class, we'll try to give you some pointers as to how to interact with Lean to construct proofs. The examples in this section will serve as a basis for discussion.

Remember that Lean's core library defines the *List* data type and notation for it. In the example below, we import the library, open the namespace, declare some variables, and try out the notation.

```
import Init

open List

variable {α : Type}
variable (as bs cs : List α)
variable (a b c : α)

#check a :: as
#check as ++ bs

example : [] ++ as = as := nil_append as

example : (a :: as) ++ bs = a :: (as ++ bs) := cons_append a as bs
```

The *variable* command does not do anything substantive. It tells Lean that when the corresponding identifiers are used in definitions and theorems that follow, they should be interpreted as arguments to those theorems and proofs, with the indicated types. The curly brackets around the declaration α *: Type* indicate that that argument is meant to be *implicit*, which is to say, users do not have to write it explicitly. Rather, Lean is expected to infer it from the context.

The library proves the theorems *[] ++ as* and *(a :: as) ++ bs = a :: (as ++ bs)* under the names *nil_append* and *cons_append*, respectively. You can see them by writing *#check nil_append* and *#check cons_append*. Remember that we took these to be the defining equations for the *append* function in Section 2.3. Although Lean uses a different definition of the append function, for illustrative purposes we will treat them as the defining equations and base our subsequent proofs on that.

Lean's library also proves *as ++ []* under the name *append_nil*, but to illustrate how proofs like this go, we will prove it again under the name *append_nil'*.

```
theorem append_nil' : as ++ [] = as := by
  induction as with
  | nil => rw [nil_append]
  | cons a as ih => rw [cons_append, ih]
```

In class, we will help you make sense of this. The *by* command tell Lean that we are going to write a *tactic* proof. In other words, instead of writing the proof as an expression, we are going to give Lean a list of instructions that tell it how to prove the theorem. At the start of the tactic proof, the theorem in question is our *goal*. At each step, tactics act on one more more of the remaining goals; when no more goals remain, the theorem is proved.

In this case, there are only two tactics that are needed. The *induction* tactic, as the name suggests, sets up a proof by induction, and the *rw* tactic *rewrites* the goal using given equations. Moving the cursor around in the editor windows shows you the goals at the corresponding state of the proof.

```
theorem append_assoc' : as ++ bs ++ cs = as ++ (bs ++ cs) := by
  induction as with
  | nil => rw [nil_append, nil_append]
  | cons a as ih => rw [cons_append, cons_append, ih, ←cons_append]
```

Here is a similar proof of the associativity of the *append* function. Note that the left arrow in the expression ←*cons_append* tell Lean that we want to use the equation from right to left instead of from left to right.

Now let us consider Lean's definition of the *reverse* function:

```
theorem reverse_def : reverse as = reverseAux as [] := rfl

theorem reverseAux_nil : reverseAux [] as = as := rfl

theorem reverseAux_cons : reverseAux (a :: as) bs = reverseAux as (a :: bs) := rfl
```

We will use these identities in the proofs that follow. Let's think about what it would take to prove the identity *reverse (as ++ bs) = reverse bs ++ reverse as*. Since *reverse* is defined in terms of *reverseAux*, we should expect to have to prove something about *reverseAux*. And since the identity mentions the *append* function, it is natural to try to characterize the way that *reverseAux* interacts with *append*. These are the two identities we need:

```
theorem reverseAux_append : reverseAux (as ++ bs) cs = reverseAux bs (reverseAux as␣
↪cs) := by
  induction as generalizing cs with
  | nil => rw [nil_append, reverseAux_nil]
  | cons a as ih => rw [cons_append, reverseAux_cons, reverseAux_cons, ih]

theorem reverseAux_append' : reverseAux as (bs ++ cs) = reverseAux as bs ++ cs := by
  induction as generalizing bs with
  | nil => rw [reverseAux_nil, reverseAux_nil]
  | cons a as ih => rw [reverseAux_cons, reverseAux_cons, ←cons_append, ih]
```

Note the *generalizing* clause in the induction. What it means is that what we are proving by induction on *as* is that the identity holds *for every choice of bs*. This means that, when we apply the inductive hypothesis, we can apply it to any choice of the parameter *bs*. You should try deleting the *generalizing* clause to see what goes wrong when we omit it.

With those facts in hand, we have the identity we are after:

```
theorem reverse_append : reverse (as ++ bs) = reverse bs ++ reverse as := by
  rw [reverse_def, reverseAux_append, reverse_def, ←reverseAux_append', nil_append,
      reverse_def]
```

16.3 Quantifiers

CHAPTER

SEVENTEEN

SIMPLE TYPE THEORY