Introduction to Formal Reasoning (COMP2065)

Release 0.1

Thorsten Altenkirch

Oct 01, 2023

CONTENTS

CHAPTER ONE

INTRODUCTION

These are lecture notes for COMP2065: Introduction to formal reasoning. The main goal is to teach formal logic using an interactive proof system called **Lean**. You will be able to use predicate logic to make precise statements and to verify them using a proof system. The covers both statements in Mathematics and statements about computer programs, e.g. their correctness.

1.1 Interactive proof systems

Interactive proof systems have been used in the past to verify a number of interesting statements, for example the four colour theorem (that every map can be coloured with 4 colour) and the formal correctness of a optimising C compiler. The formal verification of hardware like processors is now quite standard and there is a growing number of software protocols being formally certified. In academia it is now common to accompany a paper in theoretical computer science with a formal proof to make sure that the proofs are correct. Mathematicians are just starting to verify their proofs.

Even if you are not going to use formal verification, it is an important part of a computer science degree to have some acquaintance with formal logic, either to read formal statements made my others or by being able to express facts precisely without ambiguity. I believe that the best way to learn logic these days is by using an interactive proof system because this removes any ambiguity on what constitutes a proof and also you can play with it on your own without the need of a teacher or tutor.

An interactive proof system is not an automatic theorem prover. The burden to develop a correct proof is on you, the interactive proof systems guides you and makes sure that the proof is correct. Having said this, modern proof assistants offer a lot of automatisation which enables to user not to have to get bogged down in trivial details. However, since our goal is to learn proofs, at least initially we won't use much automatisation. It is like in Harry Potter, to be allowed to use the more advanced spells you first have to show that you master the basic ones.

1.2 Lean

Many interactive proof systems are based on Type Theory, this is basically a functional programming language with a powerful type system that allows us to express propositions as types and proofs as programs. A well known example is the Coq system which was developed (and still is) in France. However, we will use a more recent system which in many respects is similar to Coq, this is Lean which was (and is) developed under the leadership of Leonardo de Moura at Microsoft Research. Leonardo is famous for his work on automatic theorem proving, he developed the Z3 theorem prover. Lean's goal is to connect automatic and interactive theorem proving. The system is called *Lean* because it only relies on a small core of primitive rules and axioms to make sure that it is itself correct.

Lean is available for free from https://leanprover.github.io/download/ It will be already installed on the lab machines (I hope). You can use it either via Microsoft's Visual Code Studio or via emacs using lean mode. Yet another way to use lean is via a browser based version we will also use in the online version of these lecture notes.

Here is an example of a simple proof in Lean: we show that the sum of the first n odd numbers is the square of n. (Actually this example already uses some advanced magic in form of the *ring* tactic).

```
def sum_ℕ : ℕ → (ℕ → ℕ) → ℕ
| zero f := 0
| (succ n) f := f n + sum_ℕ n f

def nth_odd (i : ℕ) : ℕ :=
  2*i+1

#reduce (nth_odd 3)

#reduce (sum_ℕ 5 nth_odd)

theorem odd_sum : ∀ n : ℕ , sum_ℕ n nth_odd = n^2 :=
begin
  assume n,
  induction n with n' ih,
  refl,
  calc
    sum_ℕ (succ n') nth_odd
    = 2*n'+1+ sum_ℕ n' nth_odd : refl _
    ... = 2*n'+1+n'^2 : by rw ih
    ... = (n' + 1)^2 : by ring,
end
```

If you read this online, you can just click **try it!** which should transport you to the web interface. You need to wait until the orange text *Lean is busy* is replaced by a green *Lean is ready* (which may take a while depending on your computer, especially the 1st time).

You can put the cursor in the proof to see what the proof state is and you can evaluate the expressions after `#reduce` and maybe change the parameters. You can also change the proof (maybe you have a better one) or do something completely different (but then don't forget to save your work by copy and paste). While you can work using the browser version only, for bigger exercises it may be better to install Lean on your computer.

The Lean community is very active.If you want to know more about Lean and it's underlying theory, I recommend book *Theorem Proving in Lean* [AvMoKo2015] whose online version also uses the web interface. Actually if you notice that these lecture notes and the book use a very similar format then this is because I have stolen their setup (on the suggestion of one of the authors). Another useful book is The Hitchhiker's Guide to Logical Verification [BaBeBlHo2020] (this is material for an MSc course at the University of Amsterdam) which goes beyond this course. A good place for questions and discussions is the Lean zulip chat (https://leanprover.zulipchat.com/) but please don't post your coursework questions anywhere on social networks — this is considered academic misconduct. And yes, we do have staff members who can read other languages than English.

CHAPTER

TWO

PROPOSITIONAL LOGIC

2.1 What is a proposition?

A proposition is a definitive statement which we may be able to prove. In Lean we write `P : Prop` to express that `P` is a proposition.

We will later introduce ways to construct interesting propositions i.e. mathematical statements or statements about programs, but in the moment we will use propositional variables instead. We declare in Lean:

```
variables P Q R : Prop
```

This means that `P Q R` are propositional variables which may be substituted by any concrete propositions. In the moment it is helpful to think of them as statements like "The sun is shining" or "We go to the zoo."

We introduce a number of connectives and logical constants to construct propositions:

- Implication (→), read `P → Q` as **if** `P` **then** `Q`.
- Conjunction (∧), read `P ∧ Q` as `P` **and** `Q`.
- Disjunction (∨), read `P ∨ Q` as `P` **or** `Q`.

  Note that we understand *or* here as inclusive, it is ok that both are true.
- `false`, read `false` as *Pigs can fly*.
- `true`, read `true` as *It sometimes rains in England.*
- Negation (¬), read `¬ P` as **not** `P`.

  We define `¬ P` as `P → false`.
- Equivalence, (↔), read `P ↔ Q` as `P` **is equivalent to** `Q`.

  We define `P ↔ Q` as `(P → Q) ∧ (Q → P)`.

As in algebra we use parentheses to group logical expressions. To save parentheses there are a number of conventions:

- Implication and equivalence bind weaker than conjunction and disjunction.

  E.g. we read `P ∨ Q → R` as `(P ∨ Q) → R`.
- Implication binds stronger than equivalence.

  E.g. we read `P → Q ↔ R` as `(P → Q) ↔ R`.
- Conjunction binds stronger than disjunction.

  E.g. we read `P ∧ Q ∨ R` as `(P ∧ Q) ∨ R`.

- Negation binds stronger than all the other connectives.

  E.g. we read `¬ P ∧ Q` as `(¬ P) ∧ Q`.

- Implication is right associative.

  E.g. we read `P → Q → R` as `P → (Q → R)`.

This is not a complete specification. If in doubt use parentheses.

We will now discuss how to prove propositions in Lean. If we are proving a statement containing propositional variables then this means that the statement is true for all replacements of the variables with actual propositions. We say it is a tautology.

Tautologies are sort of useless in everyday conversations because they contain no information. However, for our study of logic they are important because they exhibit the basic figures of reasoning.

2.2 Our first proof

In Lean we write `p : P` for `p` proves the proposition `P`. For our purposes a proof is a sequence of *tactics* affecting the current proof state which is the sequence of assumptions we have made and the current goal. A proof begins with `begin` and ends with `end` and every tactic is terminated with `,`.

We start with a very simple tautology `P → P`: If `P` then `P`. We can illustrate this with the statement *if the sun shines then the sun shines*. Clearly, this sentence contains no information about the weather, it is vacuously true, indeed it is a tautology.

Here is how we prove it in Lean:

```
theorem I : P → P :=
begin
  assume h,
  exact h,
end
```

We tell Lean that we want to prove a `theorem` (maybe a bit too grandios a name for this) named `I` (for identity). The actual proof is just two lines, which invoke **tactics**:

- `assume h` means that we are going to prove an implication by assuming the premise (the left hand side) and using this assumption to prove the conclusion (the right hand side). If you look at the html version of this document you can click on **Try it** to open lean in a separate window. When you move the cursor before `assume h` You see that the proof state is:

  ```
  P : Prop
  ⊢ P → P
  ```

  This means we assume that `P` is a proposition and want to prove `P → P`. The `⊢` symbol (pronounced *turnstile*) separates the assumptions and the goal. After `assume h,` the proof state is:

  ```
  P : Prop,
  h : P
  ⊢ P
  ```

  This means our goal is now `P` but we have an additional assumption `h : P`.

- `exact h,` We complete the proof by telling Lean that there is an assumption that *exactly* matches the current goal. If you move the cursor after the `,` you see `no goals`. We are done.

2.3 Using assumptions

Next we are going to prove another tautology: `(P → Q) → (Q → R) → P → R`. Here is a translation into english:

If if the sun shines then we go to the zoo then if if we go to the zoo then we are happy then if the sun shines then we are happy

Maybe this already shows why it is better to use formulas to write propositions.

Here is the proof in Lean (I call it `C` for *compose*).

```
theorem C : (P → Q) → (Q → R) → P → R :=
begin
  assume p2q,
  assume q2r,
  assume p,
  apply q2r,
  apply p2q,
  exact p,
end
```

First of all it is useful to remember that → associates to the right, putting in the invisible brackets this corresponds to:

```
(P → Q) → ((Q → R) → (P → R))
```

After the three `assume` we are in the following state:

```
P Q R : Prop,
p2q : P → Q,
q2r : Q → R,
p : P
⊢ R
```

Now we have to *use* an implication. Clearly it is `q2r` which can be of any help because it promises to show *R* given *Q*. Hence once we say `apply q2r` we are left to show `Q`:

```
P Q R : Prop,
p2q : P → Q,
q2r : Q → R,
p : P
⊢ Q
```

The next step is to use `apply p2q` to reduce the goal to `P` which can be shown using `exact P`.

Note that there are two kinds of steps in these proofs:

- `assume h` means that we are going to prove an implication `P → Q` by assuming `P` (and we call this assumption `h`) and then proving `Q` with this assumption.
- `apply h` if we have assumed an implication `h : P → Q` and our current goal matches the right hand side we can use this assumption to *reduce* the problem to showing `P` (wether this is indeed a good idea depends on wether it is actually easier to show `P`).

The `apply` tactic is a bit more general, it can also be used to use a repeated implications. Here is an example:

```
theorem swap : (P → Q → R) → (Q → P → R) :=
begin
  assume f q p,
```

(continues on next page)

(continued from previous page)

```
  apply f,
  exact p,
  exact q,
end
```

After `assume f q p` (which is a short cut for writing three times `assume`) we are in the following state:

```
P Q R : Prop,
f : P → Q → R,
q : Q,
p : P
⊢ R
```

Now we can use `f` because its conclusion matches our goal but we are left with two goals:

```
2 goals
P Q R : Prop,
f : P → Q → R,
q : Q,
p : P
⊢ P

P Q R : Prop,
f : P → Q → R,
q : Q,
p : P
⊢ Q
```

After completing the first goal with `exact p` it disappears and only one goal is left:

```
P Q R : Prop,
f : P → Q → R,
q : Q,
p : P
⊢ Q
```

which we can quickly eliminate using `exact q,`.

2.4 Proof terms

What is a proof? It looks like a proof in Lean is a sequence of tactics. But this is only the surface: the tactics are actually more like editor commands which generate the real proof which is a **program**. This also explains the syntax `p : P` which is reminiscent of the notation for typing `3 :: int` in Haskell (that Haskell uses `::` instead of `:` is a regrettable historic accident).

We can have a look at the programs generated from proofs by using the `#print` operation in Lean:

```
#print I

#print C
```

The proof term associated to `I` is:

```
theorem I : ∀ (P : Prop), P → P :=
λ (P : Prop) (h : P), h
```

and the one for `C` is:

```
theorem C : ∀ (P Q R : Prop), (P → Q) → (Q → R) → P → R :=
λ (P Q R : Prop) (p2q : P → Q) (q2r : Q → R) (p : P), q2r (p2q p)
```

If you have studied functional programming (e.g. *Haskell*) you should have a *dejavu*: indeed proofs are **functional programs**. Lean exploits the *propositions as types translation* (aka *the Curry-Howard-Equivalence*) and associates to every proposition the type of evidence for this proposition. This means that to see that a proposition holds all we need to do is to find a program in the type associated to it.

Not all Haskell programs correspond to proofs, in particular general recursion is not permitted in proofs but only some limited form of recursion that will always terminate. Also the Haskell type system isn't expressive enough to be used in a system like Lean, it is ok for propositional logic but it doesn't cover predicate logic which we will introduce soon. The functional language on which Lean relies is called *dependent type theory* or more specifically *The Calculus of Inductive Constructions*.

Type Theory is an interesting subject but I won't be able to say much in this course. If you want to learn more about this you can attend *Proofs, Programs and Types* (COMP4074) which can be also done in year 3.

2.5 Conjunction

How to prove a conjunction? Easy! To prove `P ∧ Q` we need to prove both `P` and `Q`. This is achieved via the `constructor` tactic. Here is a simple example (and since I don't want to give it a name, I am using `example` instead of `theorem`).

```
example : P → Q → P ∧ Q :=
begin
  assume p q,
  constructor,
  exact p,
  exact q,
end
```

`constructor` turns the goal:

```
P Q : Prop,
p : P,
q : Q
⊢ P ∧ Q
```

into two goals:

```
2 goals
P Q : Prop,
p : P,
q : Q
⊢ P

P Q : Prop,
p : P,
q : Q
⊢ Q
```

Now what is your next question? Exactly! How do we use a conjunction in an assumption? Here is an example, we show that ∧ is *commutative*.

```
theorem comAnd : P ∧ Q → Q ∧ P :=
begin
  assume pq,
  cases pq with p q,
    constructor,
    exact q,
    exact p
end
```

After `assume pq` we are in the following state:

```
P Q : Prop,
pq : P ∧ Q
⊢ Q ∧ P
```

Assume `P ∧ Q` is the same as assuming both `P` and `Q`. This is facilitated via the `cases` tactic which needs to know which assumption we are going to use (here `pq`) and how we want to name the assumptions which replaces it (here `p q`). Hence after `cases pq with p q`, the state is:

```
P Q : Prop,
p : P,
q : Q
⊢ Q ∧ P
```

The name `cases` seems to be a bit misleading since there is only one case to consider here. However, as we will see `cases` is applicable more generally in situations where the name is better justified.

I hope you notice that the same symmetry of tactics how to prove and how to use which we have seen in the tactics for implication aso show for conjunction. This pattern is going to continue.

It is good to know that Lean always abstracts the propositional variables we have declared. We can actually use `comAnd` with different instantiation to prove the following:

```
theorem comAndIff : P ∧ Q ↔ Q ∧ P :=
begin
  constructor,
  apply comAnd,
  apply comAnd,
end
```

In the 2nd use of `comAnd` we instantiate `Q` with `P` and `P` with `Q`. Lean will find these instances automatically but in some more complicated examples it may need some help.

2.6 The Currying Equivalence

Maybe you have already noticed that a statement like `P → Q → R` basically means that `R` can be proved from assuming both `P` and `Q`. Indeed, it is equivalent to `P ∧ Q → R`. We can show this formally by using ↔. You just need to remember that `P ↔ Q` is the same as `(P → Q) ∧ (Q → P)`. Hence a goal of the form `P ↔ Q` can be turned into two goals `P → Q` and `Q → P` using the `constructor` tactic.

All the steps we have already explained so I won't comment. It is a good idea to step through the proof using Lean (just use **Try It** if you are reading this in a browser).

```
theorem curry : P ∧ Q → R ↔ P → Q → R :=
begin
```

(continues on next page)

(continued from previous page)

```
  constructor,
  assume pqr p q,
  apply pqr,
  constructor,
  exact p,
  exact q,
  assume pqr pq,
  cases pq with p q,
  apply pqr,
  exact p,
  exact q,
end
```

I call this the currying equivalence, because this is the logical counterpart of currying in functional programming: i.e. that a function with several parameters can be reduced to a function which returns a function. E.g. in Haskell addition has the type `Int -> Int -> Int` instead of `(Int , Int) -> Int`.

2.7 Disjunction

To prove a disjunction `P ∨ Q` we can either prove `P` or we can prove `Q`. This is achieved via the appropriately named tactics `left` and `right`. Here are some simple examples:

```
example : P → P ∨ Q :=
begin
  assume p,
  left,
  exact p,
end

example : Q → P ∨ Q :=
begin
  assume q,
  right,
  exact q,
end
```

To use a disjunction `P ∨ Q` we have to show that the current goal follows both from assuming `P` and from assuming `Q`. To achieve this we are using `cases` again, but this time the name actually makes sense. Here is an example:

```
theorem case_lem : (P → R) → (Q → R) → P ∨ Q → R :=
begin
  assume p2r q2r pq,
  cases pq with p q,
  apply p2r,
  exact p,
  apply q2r,
  exact q,
end
```

After `assume` we are in the following state:

```
P Q R : Prop,
p2r : P → R,
q2r : Q → R,
```

(continues on next page)

(continued from previous page)

```
pq : P ∨ Q
⊢ R
```

We use `cases pq with p q` which means that we are going to use `P ∨ Q`. There are two cases to consider one for `P` and one for `Q` resulting in two subgoals. We indicate which names we want to use for the assumptions in each case namely `p` and `q`:

```
2 goals
case or.inl
P Q R : Prop,
p2r : P → R,
q2r : Q → R,
p : P
⊢ R

case or.inr
P Q R : Prop,
p2r : P → R,
q2r : Q → R,
q : Q
⊢ R
```

To summarise: there are two ways to prove a disjunction using `left` and `right`. To use a disjunction in an assumption we use `cases` to perform a case analysis and show that the current goal follows from either of the two components.

2.8 Logic and algebra

As an example which involves both conjunction and disjunction we prove distributivity. In algebra we know the law $x(y+z) = xy + xz$ a similar law holds in propositional logic:

```
example : P ∧ (Q ∨ R) ↔ (P ∧ Q) ∨ (P ∧ R) :=
begin
  sorry
end
```

Sorry, but I was too lazy to do the proof so I left it to you. In Lean you can say `sorry` and omit the proof. This is an easy way out if you cannot complete your Lean homework: you just type `sorry` and it is fine. However, then I will say *Sorry but you don't get any points for this.*

The correspondence with algebra goes further: the counterpart to implication is exponentiation but you have to read it backwards, that is `P → Q` becomes `Q^P`. Then the translation of the law $x^{yz} = (x^y)^z$ corresponds to the currying equivalence `P ∧ Q → R ↔ P → Q → R`.

Maybe you remember that there is another law of exponentiation $x^{y+z} = x^y x^z$. And indeed its translation is also a law of logic:

```
theorem case_thm : P ∨ Q → R ↔ (P → R) ∧ (Q → R) :=
begin
  sorry
end
```

Actually the right to left direction is a curried version of the `case_lem` we have already shown. It turns out that it is another logical equivalence. It shouldn't be too difficult to complete the proof.

Intrigued? I can only warn you not to study *category theory* to find out more because it is highly addictive.

2.9 True, false and negation

There are two logical constants `true` and `false`. I find it always difficult to translate them into english because we don't really use them in conversations to state a proposition. Hence the best approximation for `true` is something which is obviously true like *It sometimes rains in England* and to translate `false` with something obviously false, like *Pigs can fly*.

As far is logic and proof concerned, `true` is like an empty conjunction and `false` is an empty disjunction. Hence it is easy to prove `true`

```
example : true :=
begin
  constructor,
end
```

Alternatively we can use the tactic `trivial`.

While in the case of ∧ we were left with 2 subgoals now we are left with none, ie. we are already done.

Symmetrically there is no way to prove `false` because we neither have `left` nor `right` and this is good so. On the other hand doing cases on `false` as an assumption makes the current goal go away as by magic and leaves no goals to be proven.

```
theorem efq : false → P :=
begin
  assume pigs_can_fly,
  cases pigs_can_fly,
end
```

`efq` is short for the latin phrase *Ex falso quod libet* which means from false follows everything. This confuses some people because it isn't really a principle of everyday logic where there are different levels of impossibility. However, in logic the reasoning *If pigs can fly then I am the president of America* is valid.

We define `¬ P` as `P → false` which means that `P` is impossible. Dear gentlemen, if you ask a girl to marry you and she replies *If we get married then pigs can fly*, this means *no*.

As an example we can prove the law of contradiction: it cannot be that both `P` and `¬ P` hold.

```
theorem contr: ¬ (P ∧ ¬ P) :=
begin
  assume pnp,
  cases pnp with p np,
  apply np,
  exact p,
end
```

2.10 Summary of tactics

Below is a table summarising the tactics we have seen so far:

	How to prove ?	How to use?
→	`assume h`	`apply h`
∧	`constructor`	`cases h with p q`
∨	`left right`	`cases h with p q`
`true`	`constructor`	
`false`		`cases h`

They are related to introduction and elimination rules in natural deduction, a system devised by the german logician Gerhard Gentzen.

The syntax for using conjunction and disjunction is the same, `cases h with p q`, but the effect is quite different. In particular the two assumptions are added both to the context in the case of $\wedge$ but only one of them in each of the subproofs in the case of $\vee$.

You can omit the `with` clause and just write `cases h` in those situations and Lean will generate names for you. However, this is not acceptable for solutions submitted as homework.

We also have `exact h` which is a structural tactic that doesn't fit into the scheme above. Actually `h` could be any proof term but since we have not introduced proof terms we will use `exact` only to refer to assumptions. There is also an alternative - the tactic `assumption` checks wether any assumption matches the current goal. Hence we could have written the first proof as:

```
theorem I : P → P :=
begin
  assume h,
  assumption,
end
```

Important! There many more tactics available in Lean some with a higher degree of automatisation. Also some of the tactics I have introduced are applicable in ways I haven't explained. When solving exercises, please use only the tactics I have introduced and only in the way I have introduced them.

CHAPTER

THREE

CLASSICAL LOGIC

We stick to propositional logic for the moment but discuss a difference between the logic based on truth you may have seen before and the logic based on evidence which we have introduced in the previous chapter.

The truth based logic is called *classical logic* while the evidence based one is called *intuitionistic logic*.

3.1 The de Morgan laws

The de Morgan laws state that if you negate a disjunction or conjunction this is equivalent to the negation of their components with the disjunction replaced by conjunction and vice versa. More precisely:

```
¬ (P ∨ Q) ↔ ¬ P ∧ ¬ Q
¬ (P ∧ Q) ↔ ¬ P ∨ ¬ Q
```

These laws reflect the observation that the truth tables for $\wedge$ and $\vee$ can be transformed into each other if we turn them around and swap `true` and `false`.

P	Q	P ∧ Q	P ∨ Q	¬ P ∧ ¬ Q	¬ P ∨ ¬ Q	¬(P ∧ Q)	¬(P ∨ Q)
false	false	false	false	true	true	true	true
true	false	false	true	false	true	true	false
false	true	false	true	false	true	true	false
true	true	true	true	false	false	false	false

Here is the proof of the first de Morgan law in its full glory:

```
theorem dm1 : ¬ (P ∨ Q) ↔ ¬ P ∧ ¬ Q :=
begin
  constructor,
  assume npq,
  constructor,
  assume p,
  apply npq,
  left,
  exact p,
  assume q,
  apply npq,
  right,
  exact q,
  assume npnq pq,
  cases npnq with np nq,
  cases pq with p q,
```

(continues on next page)

(continued from previous page)

```
  apply np,
  exact p,
  apply nq,
  exact q,
end
```

It is rather boring because there are a lot of symmetric cases but I didn't break a sweat proving it. However, the 2nd law is a different beast. Here is my attempt:

```
theorem dm2 : ¬ (P ∧ Q) ↔ ¬ P ∨ ¬ Q :=
begin
 constructor,
 assume npq,
 left,
 assume p,
 apply npq,
 constructor,
 exact p,
 sorry,
 assume npnq pq,
 cases pq with p q,
 cases npnq with np nq,
 apply np,
 exact p,
 apply nq,
 exact q,
end
```

As you see I got stuck with the left to right direction, the right to left one went fine. What is the problem? The proof state after `assume npq` is (ignoring the propositional assumptions and the other goal):

```
npq : ¬(P ∧ Q)
⊢ ¬P ∨ ¬Q
```

Now the question is do we go `left` or `right` - there seems to be no good reason for either because everything is symmetric. Ok let's try `left` we end up with:

```
npq : ¬(P ∧ Q)
⊢ ¬P
```

Now the next steps is obvious `assume p`:

```
npq : ¬(P ∧ Q),
p : P
⊢ false
```

There is only one purveyor of false, hence we say `apply npq`:

```
npq : ¬(P ∧ Q),
p : P
⊢ P ∧ Q
```

Now we say `constructor` and the first subgoal is easily disposed with `exact p` but we end up with:

```
npq : ¬(P ∧ Q),
p : P
⊢ Q
```

And there is no good way to make progress here, indeed it could be that `P` is true but `Q` is false. As soon as we said `left` we ended up with an unprovable goal.

What has happened? The truth tables provided clear evidence that the de Morgan law should hold but we couldn't prove it. Indeed let's consider the following example: *It is not the case that I have a cat* **and** *that I have a dog* can we conclude that *I don't have a cat* **or** *I don't have a dog* ? No because we don't know which one is the case, that is we don't have evidence for either.

3.2 The law of the excluded middle

To match the truth semantics we need to assume one axiom, the *law of the excluded middle*. This expresses the idea that every proposition is either true or false or to speak with Shakespeare *To* `P` *or not to* `P` that is `P ∨ ¬ P` for any proposition `P`. In latin this law is called *Tertium non datur*, which translates to *the 3rd is not given*.

In Lean we access this axiom by:

```
open classical

#check em P
```

Here I am using the command `#check` which checks the type of a term. For any proposition `P`, `em P` proves `P ∨ ¬ P`. Using `em P` we can complete the missing direction of the 2nd de Morgan law:

```
theorem dm2_em : ¬ (P ∧ Q) → ¬ P ∨ ¬ Q :=
begin
  assume npq,
  cases em P with p np,
  right,
  assume q,
  apply npq,
  constructor,
  exact p,
  exact q,
  left,
  exact np,
end
```

The idea of the proof is that we look at both cases of `P ∨ ¬ P`. If `P` holds then we can prove `¬ Q` from `¬ (P ∧ Q)`, otherwise if we know `¬ P` then we can obviously prove `¬ P ∨ ¬ Q`.

3.3 Indirect proof

There is another law which is equivalent to the principle of excluded middle and this is the *principle of indirect proof* or in latin *reduction ad absurdo* (reduction to the absurd). This principle tells you that to prove `P` it is sufficient to show that `¬ P` is impossible. Here is how we derive this using `em`:

```
theorem raa : ¬ ¬ P → P :=
begin
  assume nnp,
  cases (em P) with p np,
  exact p,
  have f : false,
  apply nnp,
```

(continues on next page)

(continued from previous page)

```
  exact np,
  cases f,
end
```

The idea is to assume `¬¬ P` and then prove `P` by analysing `P ∨ ¬ P`: In the case `P` we are done and in the case `¬ P` we have a contradiction with `¬¬ P` and we can use that false implies everything.

We can derive `em` from `raa`. The key observation is that we can actually prove `¬ ¬ (P ∨ ¬ P)` without using classical logic.

```
theorem nn_em : ¬ ¬ (P ∨ ¬ P) :=
begin
  assume npnp,
  apply npnp,
  right,
  assume p,
  apply npnp,
  left,
  exact p,
end
```

This proof is a bit weird. After `apply npnp` we have the following state:

```
P : Prop,
npnp : ¬(P ∨ ¬P)
⊢ P ∨ ¬P
```

Now you may say again do we go left or right? But this time the cases are not symmetric. we certainly cannot prove `P` hence let's go right. After a few steps we are in the same situation again:

```
P : Prop,
npnp : ¬(P ∨ ¬P),
p : P
⊢ P ∨ ¬P
```

But something has changed! We have picked up the assumption `p : P`. And hence this time we go left and we are done.

Here is a little story which relies on the idea that double negating corresponds to time travel:

> "There was a magician who could time travel who wanted to marry the daughter of a king. There was no gold in the country but people were not sure wether diamonds exist. Hence the king set the magician the task to either find a diamond or to produce a way to turn diamonds into gold. The magician went for the 2nd option and gave the king an empty box so he could marry the daughter. However, if the king would get hold of a diamond at some point and his lie would become obvious he would just take the diamond, travel back in time and go for the first option."

Now if we assume we have a constant proving *raa* we can show *em*:

```
constant raa : ¬¬ P → P

theorem em : P ∨ ¬ P :=
begin
  apply raa,
  apply nn_em,
end
```

Note that while `em` and `raa` are equivalent as global principles this is not the case for individual propsoitions. That is if we assume `P ∨ ¬ P` we can prove `¬¬ P → P` for the same proposition `P` but if we assume `¬¬ P → P` we cannot prove `P ∨ ¬ P` for that proposition but we actually need a different instance of `raa` namely : `¬¬ (P ∨ ¬ P) → P ∨ ¬ P`.

3.4 Intuitionistic vs classical logic

Should we always assume `em` (or alternatively `raa`), hence should we always work in classical logic? There is a philosophical and a pragmatic argument in favour of avoiding it and using intuitionistic logic.

The philosophical argument goes like this: while facts about the real world are true or false even if we don't know them this isn't so obvious about mathematical constructions which take place in our head. The set of all numbers doesn't exist in the real world it is like a story we share and we don't know wether anything we make up is either true or false. However, we can talk about evidence without needing to assume that.

The idea that the world of ideas is somehow real, and that the real world is just a poor shadow of the world of ideas was introduced by the greek philosopher Plato and hence is called *Platonism*. In contrast that our ideas are just constructions in our head is called *Intuitionism*.

However, the pragmatic argument is maybe more important. Intuitionistic logic is constructive, indeed in a way that is dear to us computer scientists: whenever we show that something exists we are actually able to compute it. As a consequence intuitionistic logic introduces many distinctions which are important especially in computer science. For example we can distinguish decidable properties from properties in general. Also by a function we mean something we can compute like in a (functional) programming language.

Here is a famous example to show that the principle of excluded middle destroys constructivity. Since we haven't yet introduced predicate logic and many of the concepts needed for this example in Lean I present this just as an informal argument:

> We want to show that there are two irrational numbers p and q (that is numbers that cannot be written as fractions) such that their power p^q is rational. We know that $\sqrt{2}$ is irrational. Now what is $\sqrt{2}^{\sqrt{2}}$? Using the excluded middle it is either rational or irrational. If it is rational then we are done $p = q = \sqrt{2}$. Otherwise we use $p = \sqrt{2}^{\sqrt{2}}$ which we now assume to be irrational and $q = \sqrt{2}$. Now a simple calculation shows $p^q = (\sqrt{2}^{\sqrt{2}})^{\sqrt{2}} = \sqrt{2}^{\sqrt{2}\sqrt{2}} = \sqrt{2}^2 = 2$ which is certainly rational.

Now after this proof we still don't know two irrational numbers whose power is rational. I hasten to add that we can establish this fact without using `em` but this particular proof doesn't provide a witness because it is using excluded middle.

In the homework we will distinguish proofs that use excluded middle and once which do not. In my logic poker I ask you to prove propositions intuitionistically where possible and only use classical reasoning where necessary.

Now a common question is how to see wether a proposition is only provable classically. E.g. why can we prove all the first de Morgan law and one direction of the 2nd but not `¬ (P ∧ Q) → ¬ P ∨ ¬ Q`? The reason is that the right hand side contains some information, i.e. which of the `¬ P` or `¬ Q` is true while `¬ (P ∧ Q)` is a *negative* proposition and hence does contain no information. In contrast the both sides of the first de Morgan law `¬ (P ∨ Q)` and `¬ P ∧ ¬ Q` are negative, i.e. contain no information.

CHAPTER

FOUR

PREDICATE LOGIC

4.1 Predicates, relations and quantifiers

Predicate logic extends propositional logic, we can use it to talk about objects and their properties. The objects are organized in *types*, such as `ℕ : Type` the type of natural numbers $\{0, 1, 2, 3, \ldots\}$ or `bool : Type` the type of booleans $\{\text{tt}, \text{ff}\}$, or lists over a given `A : Type`: `list A : Type`, which we will introduce in more detail soon. To avoid talking about specific types we introduce some type variables:

```
variables A B C : Type
```

We talk about types where you may be used to *sets*. While they are subtle differences (types are static while we can reason about set membership in set theory) for our purposes types are just a replacement of sets.

A predicate is just another word for a property, e.g. we may use `Prime : ℕ → Prop` to express that a number is a prime number. We can form propositions such as `Prime 3` and `Prime 4`, the first one should be provable while the negation of the second holds. Predicates may have several inputs in which case we usually call them relations, examples are `≤ : ℕ → ℕ → Prop` or `inList : A → list A → Prop` to form propositions like `2 ≤ 3` and `InList 1 [1,2,3]` (both of them should be provable).

In the sequel we will use some generic predicates for examples, such as

```
variables PP QQ : A → Prop
```

The most important innovation of predicate logic are the quantifiers, which we can use to form new propositions:

- universal quantification (∀), read `∀ x : A , PP x` as all `x` in `A` satisfy `PP x`.
- existential quantification (∃), read `∃ x : A, PP x` as there is an `x` in `A` satsifying `PP x`.

Both quantifiers bind weaker then any other propositional operator, that is we read `∀ x : A, PP x ∧ Q` as `∀ x : A , (PP x ∧ Q)`. We need parenheses to limit the scope, e.g. `(∀ x : A, PP x) ∧ Q` which has a different meanting to the proposition before.

It is important to understand bound variables, essentially they work like scoped variables in programming. We can shadow variables as in `∀ x:A , (∃ x : A , PP x) ∧ QQ x`, here the `x` in `PP x` referes to `∃ x : A` while the `x` in `QQ x` refers to `∀ x : A`. Bound variables can be consistently renamed, hence the previous proposition is the same as `∀ y:A , (∃ z : A , PP z) ∧ QQ y`, which is actually preferable since shadowing variables should be avoided because it confuses the human reader.

Now we have introduced all these variables what can we do with them. We have new primitive proposition:

- equality (=), given `a b : A` we write `a = b` which we read as `a` is equal to `b`.

In the moment we only have variables as elements of types but this will change soon when we introduce datatypes and functions.

4.2 The universal quantifier

To prove that a proposition of the form `∀ x : A , PP x` holds we assume that there is given an arbitrary element `a` in `A` and prove it for this generic element, i.e. to prove `PP a`, we use `assumption a` to do this. If we have an assumption `h : ∀ x : A , PP x` and our current goal is `PP a` for some `a : A` then we can use `apply h` to prove our goal. Usuall we have some combination of implication and for all like `h : ∀ x : A, PP x → QQ x` and now if our current goal is `QQ a` and we invoke `apply h` Lean will instantiate `x` with `a` and it remains to show `QQ a`.

Best to do some examples. Let's say we want to prove

`(∀ x : A, PP x) → (∀ y : A, PP y → QQ y) → ∀ z : A , QQ z`

Here is a possible translation into english where we assume that `A` stands for the type of students in the class, `PP x` means `x` *is clever* and `QQ x` means `x` *is funny* then we arrive at:

> *If all students are clever then if all clever students are funny then all students are funny.*

```
example : (∀ x : A, PP x)
  → (∀ y : A, PP y → QQ y)
  → ∀ z : A , QQ z :=
begin
  assume p pq a,
  apply pq,
  apply p,
end
```

Note that after `assume` the proof state is:

```
p : ∀ (x : A), PP x,
pq : ∀ (y : A), PP y → QQ y,
a : A
⊢ QQ a
```

That is the `x` in `QQ x` has been replaced by `a`. I could have used `x` again but I though this may be misleading because you may think that you have to use the same variable as in the quantifier.

Let's prove a logical equivalence imvolving ∀ and ∧, namely that we can interchange them. That is we are going to prove

`(∀ x : A, PP x ∧ QQ x) ↔ (∀ x : A , PP x) ∧ (∀ x : A, QQ x)`

To illustrate this: to say that *all students are clever and funny* is the same as saying that *all students are clever and all students are funny*.

Here is the Lean proof:

```
example : (∀ x : A, PP x ∧ QQ x)
  ↔ (∀ x : A , PP x) ∧ (∀ x : A, QQ x) :=
  begin
    constructor,
    assume h,
    constructor,
    assume a,
    have pq : PP a ∧ QQ a,
    apply h,
    cases pq with pa qa,
    exact pa,
    assume a,
```

(continues on next page)

(continued from previous page)

```
    have pq : PP a ∧ QQ a,
    apply h,
    cases pq with pa qa,
    exact qa,
    assume h,
    cases h with hp hq,
    assume a,
    constructor,
    apply hp,
    apply hq,
  end
```

I am using a new structural rule here. After `assume a` I am in the following state (ignoring the parts not relevant now):

```
h : ∀ (x : A), PP x ∧ QQ x,
a : A
⊢ PP a
```

Now I cannot say `apply h` because `PP a` is not the conclusion of the assumption. My idea is that I can prove `PP a ∧ QQ a` from `h` and from this I can prove `PP a`. Hence I am using

```
have pq : PP a ∧ QQ a,
```

which creats a new goal:

```
h : ∀ (x : A), PP x ∧ QQ x,
a : A
⊢ PP a ∧ QQ a

h : ∀ (x : A), PP x ∧ QQ x,
a : A,
pq : PP a ∧ QQ a
⊢ PP a
```

but also inserts an assumption `pq` in my original goal. Now I can prove `PP a ∧ QQ a` using `apply h` and then I am left with:

```
h : ∀ (x : A), PP x ∧ QQ x,
a : A,
pq : PP a ∧ QQ a
⊢ PP a
```

which I can prove using `cases` on `pq`.

4.3 The existential quantifier

To prove a proposition of the form `∃ x : A , PP x` it is enough to prove `PP a` for any `a : A`. We use `existsi a` for this and we are left having to prove `PP a`. Note that `a` can be any expression of type `A` not necessarily a variable. However so far we haven't seen any way to construct elements, but this will change soon.

On the other hand to use an assumption of the form `h : ∃ x : A , P x` we are using `cases h with x px` which replaces `h` with two assumptions `x : A` and `px : P x`.

Again it is best to look at an example. We are going to prove a proposition very similar to the one for ∀:

```
(∃ x : A, PP x) → (∀ y : A, PP y → QQ y) → ∃ z : A , QQ z
```

Here is the english version using the same translation as before:

If there is a clever student and all clever students are funny then there is a funny student.

Here is the Lean proof:

```
example : (∃ x : A, PP x)
  → (∀ y : A, PP y → QQ y)
  → ∃ z : A , QQ z :=
begin
  assume p pq,
  cases p with a pa,
  existsi a,
  apply pq,
  exact pa,
end
```

After the `assume` we are in the following state:

```
p : ∃ (x : A), PP x,
pq : ∀ (y : A), PP y → QQ y
⊢ ∃ (z : A), QQ z
```

We first take `p` apart using `cases p with a pa`:

```
pq : ∀ (y : A), PP y → QQ y,
a : A,
pa : PP a
⊢ ∃ (z : A), QQ z
```

and now we can use `existsi a`:

```
pq : ∀ (y : A), PP y → QQ y,
a : A,
pa : PP a
⊢ QQ a
```

which we now should know how to complete.

As ∀ can be exchanged with ∧, ∃ can be exchanged with ∨. That is we are going to prove the following equivalence:

`(∃ x : A, PP x ∨ QQ x) ↔ (∃ x : A , PP x) ∨ (∃ x : A, QQ x)`

Here is the english version

There is a student who is clever or funny is the same as saying there is a student who is funny or there is a student who is clever.

Here is the complete lean proof (for you to step through online):

```
example : (∃ x : A, PP x ∨ QQ x)
               ↔ (∃ x : A , PP x) ∨ (∃ x : A, QQ x) :=
begin
  constructor,
  assume h,
  cases h with a ha,
  cases ha with pa qa,
  left,
  existsi a,
  exact pa,
```

(continues on next page)

(continued from previous page)

```
  right,
  existsi a,
  exact qa,
  assume h,
  cases h with hp hq,
  cases hp with a pa,
  existsi a,
  left,
  exact pa,
  cases hq with a qa,
  existsi a,
  right,
  exact qa,
end
```

4.4 Another Currying equivalence

You may have noticed that the way we prove propositions involving → and ∀ is very similar. In both cases wem use `assume` to prove them by introducing an assumption in the first case a proposition and in the secnde an element in a type and in both cases we use them using `apply` to prove the current goal. Similarily ∧ and ∃ behave similar: in both cases we prove them using constructor where we have to construct two components in the first case the two sides of the conjunction and in the second the element and the proof that it satisfies the property. And in both case we are using `cases` with two components which basically replaces the assumption by its two components.

The similarity can be seen by establishing another currying-style equivalence. While currying in propositional logic had the form

```
P ∧ Q → R ↔ P → Q → R
```

where we turn a conjunction into an implication, currying for predicate logic has the form

```
(∃ x : A, QQ x) → R ↔ (∀ x : A , QQ x → R)
```

ths time we turn an existential into a universal quantifier. For the intuition, we use `QQ x` to mean `x` *is clever* and `R` means *the professor is happy*. Hence the equivalence is:

> *If there is a student who is clever then the professor is happy is equivalent to saying if any student is clever then the professor is happy.*

Here is the proof in Lean:

```
theorem curry_pred : (∃ x : A, PP x) → R  ↔ (∀ x : A , PP x → R)   :=
begin
  constructor,
  assume ppr a p,
  apply ppr,
  existsi a,
  exact p,
  assume ppr pp,
  cases pp with a p,
  apply ppr,
  exact p,
end
```

4.5 Equality

There is a generic relation which can be applied to any type: *equality*. Given `a b : A` we can construct `a = b : Prop` expressing that `a` and `b` are equal. we can prove that everything is equal to itself using the tactic `reflexivity`.

```
example : ∀ x : A, x=x :=
begin
  assume a,
  reflexivity,
end
```

If we have assumed an equality `h : a=b` we can use it to *rewrite* `a` into `b` in the goal. That is if our goal is `PP a` we say `rewrite h` and this changes the goal into `PP b`. Here is a simple example (with a little twist):

```
example:  ∀ x y : A, x=y → PP y → PP x :=
begin
  assume x y eq p,
  rewrite eq,
  exact p,
end
```

Sometimes we want to use the equality in the other direction, that is we want to replace `b` by `a`. In this case we use `rewrite← h`. Here is another example which is actually what I wanted to do first:

```
example:  ∀ x y : A, x=y → PP x → PP y :=
begin
  assume x y eq p,
  rewrite← eq,
  exact p,
end
```

Equality is an *equivalence relation*, it mens that it is

- reflexive (`∀ x : A, x=x`),
- symmetric (`∀ x y : A, x=y → y=x`)
- transitive (`∀ x y z : A, x=y → y=z → x=z`)

We have already shown reflexivity using the appropriately named tactic. We can show symmetry and transitivity using `rewrite`:

```
theorem sym_eq : ∀ x y : A, x=y → y=x :=
begin
  assume x y p,
  rewrite p,
end
```

After the assume the goal is:

```
x y : A,
p : x = y
⊢ y = x
```

Now I was expecting that after `rewrite p` the goal would be:

```
x y : A,
p : x = y
⊢ y = y
```

but actually `rewrite` automatically applies reflexivity if possible, hence we are already done.

Moving on to transitivity:

```
theorem trans_eq : ∀ x y z : A, x=y → y=z → x=z :=
begin
  assume x y z xy yz,
  rewrite xy,
  exact yz,
end
```

Sometimes we want to use an equality not to rewrite the goal but to reqrite another assumption. We can do this giving rise to another proof of trans.

```
theorem trans_eq : ∀ x y z : A, x=y → y=z → x=z :=
begin
  assume x y z xy yz,
  rewrite<- xy at yz,
  exact yz,
end
```

That is by saying `rewrite xy at yz` we are using `xy` to rewrite `yz`. The same works for `rewrite←`.

Actually Lean already has built-in tactics to deal with symmetry and transitivity which are often easier to use:

```
example : ∀ x y : A, x=y → y=x :=
begin
  assume x y p,
  symmetry,
  exact p
end

example : ∀ x y z : A, x=y → y=z → x=z :=
begin
  assume x y z xy yz,
  transitivity,
  exact xy,
  exact yz,
end
```

After we say``transitivity`` the goals looks a bit strange:

```
2 goals
A : Type,
x y z : A,
xy : x = y,
yz : y = z
⊢ x = ?m_1

A : Type,
x y z : A,
xy : x = y,
yz : y = z
⊢ ?m_1 = z
```

The `?m_1` is a placeholder since Lean cannot figure out at this point what we are going to use as the intermidiate term. We can just proceed, because as soon as we say `exact xy`, Lean can figure out that `?m_1` is `y` and we are left with:

```
A : Type,
x y z : A,
xy : x = y,
yz : y = z
⊢ y = z
```

which is the 2nd goal with `?m_1` instantiated. Actually we are not going to use transitivity in equational proofs but we use a special format for equational proofs indicated by `calc`:

```
example : ∀ x y z : A, x=y → y=z → x=z :=
begin
  assume x y z xy yz,
  calc
    x = y   : by exact xy
    ... = z : by exact yz,
end
```

After `calc` we prove a sequence of equalities where each step is using `by` followed by some tactics. Any subsequent line starts with `...` which stends for the last expression of the previous line, in this case `y`.

There is one more property we expect from equality. Assume we have a function `f : A → B` and we know that `x = y` for some `x y : A` then we want to be able to conclude that `f x = f y`. We say that equality is a congruence. We can prove this easily using `rewrite`:

```
theorem congr_arg : ∀ f : A → B , ∀ x y : A, x = y → f x = f y :=
begin
  assume f x y h,
  rewrite h,
end
```

We will use `congr_arg f : ∀ x y : A, x = y → f x = f y` when we need that `f` preserves equality.

There is a special proof style in Lean for equational proofs `calc` which enables to have more readable equational derivations, which we will introduce later.

4.6 Classical Predicate Logic

We can use classical logic in predicate logic even though the explanation using truth tables doesn't work anymore, or let's say we have to be prepared to use infinite truth tables which need a lot of paper to write out.

There are predicate logic counterparts of the de Morgan laws which now say that you can move negation through a quantifier by negating the component and switching the quantifier. And again one of them is provable intuitionistically:

```
theorem dm1_pred : ¬ (∃ x : A, PP x) ↔ ∀ x : A, ¬ PP x :=
begin
  constructor,
  assume h x p,
  apply h,
  existsi x,
  exact p,
  assume h p,
  cases p with a pa,
  apply h,
  exact pa,
end
```

While the other direction again needs classical logic:

```
theorem dm2_pred : ¬ (∀ x : A, PP x) ↔ ∃ x : A, ¬ PP x :=
begin
  constructor,
  assume h,
  apply raa,
  assume ne,
  apply h,
  assume a,
  apply raa,
  assume np,
  apply ne,
  existsi a,
  exact np,
  assume h na,
  cases h with a np,
  apply np,
  apply na,
end
```

Actually I had to use indirect proof *raa* twice to derive the left to right direction. Still our explanation we had given previously still applies: the existential statement `∃ x : A, ¬ PP x` conatins information but the negated universal one `¬ (∀ x : A, PP x)` deosn't. Intuitively: *knowing that not all students are stupid doesn't give you a way to come up with a student who is not stupid*.

4.7 Summary of tactics

Here is the summary of basic tactics for predicate logic:

	How to prove ?	How to use?
∀	`assume h`	`apply h`
∃	`existsi a`	`cases h with x p`
=	`reflexivity`	`rewrite h` `rewrite← h`

Note that the `a` after `exsistsi` can be any expression of type `A`, while `h x p` are variables.

CHAPTER

FIVE

THE BOOLEANS

The logic we have introduced so far was very generic. We fix this in this chapter by looking at a very simple type, the booleans `bool` which has just two elements `tt` (for *true*) and `ff` (for *false*) and functions on this type. Then we are going to use predicate logic to prove some simple theorems about booleans.

In the lean prelude `bool` is defined as an inductive type:

```
inductive bool : Type
| ff : bool
| tt : bool
```

This declaration means:

- There is a new type `bool : Type`,
- There are two elements `tt ff : bool`,
- These are the only elements of `bool`,
- `tt` and `ff` are different elements of `bool`.

`Inductive` is quite versatile we can use it to define other finite types, infinite types like ℕ and type constructors like `maybe` or `list`. It is similar to the `data` type constructor in Haskell but not exactly since there are `data` definitions in Haskell which are not permitted in Lean.

5.1 Functions on bool

Let's define negation on booleans this is a function `bnot : bool → bool`. By a function here we mean something which we can feed an element of the input type (here `bool`) and it will return an element of the output type (here `bool` again. We can do this by *matching* all possible inputs:

```
def bnot : bool → bool
| tt := ff
| ff := tt
```

To define a function with two inputs, like *and* for booleans `band` we use *currying*, that is `band` applied to a boolean returns a function which applied to the 2nd boolean returns a boolean, hence `band : bool → bool → bool`. As we have already seen → is right associative hence putting the extra brackets in the type is `band :bool → (bool → bool).`

We could list all the four cases, reproducing the truth table, but we get away with just two matching only on the first argument:

```
def band : bool → bool → bool
| tt b := b
| ff b := ff
```

If the first argument is `tt`, that is we look at `band tt : bool → bool` then this is just the identity on the snd argument, because `band tt tt = tt` and `band tt ff = ff`. If the first argument is `ff` then the outcome wil be `ff` whatever the 2nd argument its. With other words `band ff : bool → bool` is the constant function which will always return `ff`.

Symmetrically we can define `bor : bool → bool → bool` with just two cases:

```
def bor : bool → bool → bool
| tt b := tt
| ff b := b
```

In this case if the first argument is `tt`, `bor tt : bool → bool` is always `tt` while if the first argument is `ff`, `bor ff : bool → bool` is just the identity.

The lean prelude also introduces the standard infix notation for operations on `bool`:

`x && y := band x y`

and

`x || y := bor x y`.

We can evaluate boolean expressions using `#reduce`:

```
#reduce ff && (tt || ff)
#reduce tt && (tt || ff)
```

When I defined the binary boolean functions my choice to match on the first argument was quite arbitrary, I could have used the 2nd argument just as well:

```
def band2 : bool → bool → bool
| b tt := b
| b ff := ff

def bor2 : bool → bool → bool
| b tt := tt
| b ff := b
```

These functions produce the same truth table as the ones we have defined before but their computational behaviour is different which is important when we prove something about them. This can be seen by applying them to a variable:

```
variable x : bool

#reduce band tt x
#reduce band2 tt x
#reduce band x tt
#reduce band2 x tt
```

We see that `band tt x` reduces to `x` because it matches on the first argument, while `band2 x tt` is stuck, it needs to see what `x` turns out to be. If we fix the 2nd argument it is just the other way around, this time `band2` is *better* behaved.

When we are doing proofs it is important to remember how the function is defined because when we are doing case analysis we should instantiate the arguments which allow us to reduce the function, if possible.

5.2 Proving some basic properties

To reason about bool we can use `cases x` to analyze a variable `x : bool` which means that there are two possibilities `tt` and `ff`. For example we can state and prove in predicate logic that every element of *bool* is either `tt` or `ff`:

```
example : ∀ x : bool, x=tt ∨ x=ff :=
begin
  assume x,
  cases x,
  right,
  refl,
  left,
  refl,
end
```

After *assume x* we are in the following state:

```
x : bool
⊢ x = tt ∨ x = ff
```

And after *cases x* we get two subgoals:

```
2 goals
case bool.ff
⊢ ff = tt ∨ ff = ff

case bool.tt
⊢ tt = tt ∨ tt = ff
```

Both of them are straightforward to prove. Now lets prove that both are different. The idea is to define a predicate `is_tt : bool → Prop` by case analysis:

```
def is_tt : bool → Prop
| ff := false
| tt := true
```

Now we can use `is_tt` to show that `tt` and `ff` cannot be equal:

```
theorem cons : tt ≠ ff :=
begin
  assume h,
  change is_tt ff,
  rewrite ← h,
  trivial,
end
```

Here I am using `tt ≠ ff` which just a shorthand for `¬ (tt = ff)` (and which again is just short for `tt = ff → false`).

I am also using a new tactic `change` which replaces the current goal with another one that is definitionally the same. In our case we have::

```
h : tt = ff
⊢ false
```

and we know that `is_tt ff = false` hance we can say `change is_tt ff` to replace the goal:

h : tt = ff ⊢ is_tt ff

But now we can use `rewrite ← h` to change the goal to `is_tt tt` which is equal to `true` and hence provable by `trivial`.

However, since this is a common situation Lean provides the tactic `contradiction` which we can use to prove goals like this:

```
example :   tt ≠ ff  :=
begin
  assume h,
  contradiction,
end
```

`contradiction` will solve the current goal if there is an inconsistent assumption like assuming that two different constructors of an inductive type are equal.

5.3 Proving equations about bool

Ok next let's prove some interesting equalities. We are going to revisit our old friend, *distributivity* but this time for booleans:

```
theorem distr_b : ∀ x y z : bool,
  x && (y || z) = x && y || x && z :=
begin
  assume x y z,
  cases x,
  dsimp [band],
  dsimp [bor],
  refl,
  dsimp [band],
  refl,
end
```

After `assume x y z,` we are in the following state:

```
x y z : bool
⊢ x && (y || z) = x && y || x && z
```

We do case analysis on `x` which can be either `tt` or `ff`. Hence after `cases x` we are left with two subgoals:

```
2 goals
case bool.ff
y z : bool
⊢ ff && (y || z) = ff && y || ff && z

case bool.tt
y z : bool
⊢ tt && (y || z) = tt && y || tt && z
```

Let's just discuss the first one. We can instruct Lean to use the definition of `&&` (i.e. `band`) by saying `dsimp [band]`, we are left with

```
y z : bool
⊢ ff = ff || ff
```

Now we just need to apply the definition of `||` (i..e bor) using `dsimp [bor]`:

```
y z : bool
⊢ ff = ff
```

and now we only need to use `refl` to dispose of this goal.

Can you see why in the 2nd case it is enough to use `dsimp [band]`.

We can apply several definitions in one tactic, i.e. in the first case `dsimp [band,bor]` would have done the same. But actually in this proof there is no need for `dsimp` at all because `refl` will automatically reduce its arguments, hence we could have just written:

```
theorem distr_b : ∀ x y z : bool,
  x && (y || z) = x && y || x && z :=
begin
  assume x y z,
  cases x,
  refl,
  refl,
end
```

However, when doing proofs interactively it may be helpful to see the reductions. Also, later we will encounter cases where using `dsimp` is actually necessary.

Could we have used another variable than `x`. Let's see what about `z`?

```
example : ∀ x y z : bool,
  x && (y || z) = x && y || x && z :=
begin
  assume x y z,
  cases z,
  sorry
end
```

After `cases zz` we are in stuck in the following state:

```
x y : bool
⊢ x && (y || ff) = x && y || x && ff
```

No reduction is possible because we have defined the functions matching on the first argument. Using `cases y` would have done a bit reduction but not enough. `cases x` was the right choice.

Now have a go at the de Morgan laws for `bool` yourselves, both are provable just using case analysis:

```
theorem dm1_b : ∀ x y : bool, bnot (x || y) = bnot x && bnot y :=
begin
  sorry,
end

theorem dm2_b : ∀ x y : bool, bnot (x && y) = bnot x || bnot y :=
begin
  sorry,
end
```

5.4 Relating bool and Prop

I am sure it has not escaped you that we seem to define logical operations twice: once for `Prop` and once for `bool`. How are the two related? Indeed we can use `is_tt` for example to relate $\wedge$ and `&&`:

```
theorem and_thm : ∀ x y : bool, is_tt x ∧ is_tt y ↔ is_tt (x && y) :=
begin
  assume x y,
  constructor,
  assume h,
  cases h with xtt ytt,
  cases x,
  cases xtt,
  cases y,
  cases ytt,
  constructor,
  assume h,
  cases x,
  cases h,
  cases y,
  cases h,
  constructor,
  constructor,
  constructor,
end
```

I recommend to step through the proof, we are only using tactics I have already explained. Note that `cases x` is used for two things: for doing case analysis on booleans and to eliminate inconsistent assumptions.

I leave it is an exercise to prove the corresponding facts about negation and disjunction:

```
theorem not_thm : ∀ x : bool, ¬ (is_tt x) ↔ is_tt (bnot x) :=
begin
  sorry,
end

theorem or_thm : ∀ x y : bool, is_tt x ∨ is_tt y ↔ is_tt (x || y) :=
begin
  sorry,
end
```

Another challenge is to define

`implb : bool → bool → bool`

and

`eqb : bool → bool → bool`

and show that they implement $\to$ and $\leftrightarrow$ for `bool`.

CHAPTER

SIX

THE NATURAL NUMBERS

We have already used the natural numbers (ℕ) in examples but now we will formally define them. We will in spirit follow Guiseppe Peano who codified the laws of natural numbers using predicate logic in the late 19th century. This is referred to as *Peano Arithmetic*.

Peano viewed the natural numbers as created from `zero` (`0 = zero`) and `succ` successor, i.e. `1 = succ 0`, `2 = succ 1` and so on. In Lean this corresponds to the following inductive definition:

```
inductive nat : Type
| zero : nat
| succ : nat → nat
```

This declaration means:

- There is a new type `nat : Type`,
- There are two elements `zero : nat`, and given `n : nat` we have `succ n : nat`.
- All the elements of `nat` can be generated by `zero` and then applying `succ` a finite number of times,
- `zero` and `succ n` are different elements,
- `succ` is injective, i.e. given `succ m = succ n` then we know that `m = n`.

We adopt the notation that `ℕ = nat` and Lean also automatically translates the usual decimal notation into elements of ℕ. This is convenient because otherwise it would be quite cumbersome to write out a number like `1234 : ℕ`.

6.1 Basic properties of ℕ

Let's verify some basic properties of ℕ which actually correspond to some of Peano's axioms. First of all we want to verify that `0 ≠ succ n` which corresponds to `true ≠ false` for `bool`. We could apply the same technique as before but actually `contradiction` does the job straight away:

```
example : ∀ n : ℕ, 0 ≠ succ n :=
begin
  assume n h,
  contradiction,
end
```

Next let's show that `succ` is injective. To do this we define a predecessor function `pred` using pattern matching which works the same way as for `bool`:

```
def pred : ℕ → ℕ
| zero := zero
| (succ n) := n

#reduce (pred 7)
```

We can test the function using `#reduce (pred 7)`.

We defined `pred 0 = 0` which is a bit arbitrary. However, `pred` does the job to show that `succ` is injective:

```
theorem inj_succ : ∀ m n : nat, succ m = succ n → m = n :=
begin
  assume m n h,
  change pred (succ m) = pred (succ n),
  rewrite h,
end
```

Here I am using `change` again to replace the goal `m = n` with `pred (succ m) = pred (succ n)` exploiting that `pred (succ x) = x`. On the new goal I can apply `rewrite`.

However, there is also a tactic called `injection` which does this automatically for all inductive types which avoids the need to define `pred`:

```
example :  ∀ m n : nat, succ m = succ n → m = n :=
begin
  assume m n h,
  injection h,
end
```

`injection h` can be applied to a hypothesis of the form `h : succ m = succ n`.

6.2 Structural recursion

You may have already seen recursive programs. When defining functions on ℕ we will need recursion but unlike the general recursion available in programming languages we will only use *structural recursion*. That is when we define a function on the natural numbers we can use the function on `n` to compute it for `succ n`. A simple example for this is the `double` function which doubles a number:

```
def double : ℕ → ℕ
| zero := 0
| (succ n) := succ (succ (double n))

#reduce (double 7)
```

Here `double (succ n)` is `suc (suc (double n))`. That is basically every `succ` is replaced by two `succ` s. For example:

```
double 3
= double (succ 2)
= succ (succ (double 2))
= succ (succ (double (succ 1)))
= succ (succ (succ (succ (double 1))))
= succ (succ (succ (succ (double (succ 0)))))
```

(continues on next page)

(continued from previous page)

```
= succ (succ (succ (succ (succ (succ (double 0))))))
= succ (succ (succ (succ (succ (succ (double zero))))))
= succ (succ (succ (succ (succ (succ zero)))))
= 6
```

That we allowed to use recursion really is a consequence of the idea that every natural number can be obtained by applying `succ` a finite number of times. Hence when we run the recursive program it will terminate in a finite number of steps. Other uses of recursion are not allowed, for example we are not allowed to write:

```
def bad : ℕ → ℕ
| zero := zero
| (succ n) := succ (bad (succ n))
```

Indeed this function would not terminate and Lean complains about it. However, actually *structural recursion* is a bit more general then what I just said, for example we can define the inverse to `double` (which is division by 2 without remainder):

```
def half : ℕ → ℕ
| zero := zero
| (succ zero) := zero
| (succ (succ n)) := succ (half n)

#reduce (half 7)
#reduce (half (double 7))
```

We define `half` which replaces every two `succ` s by one. In the recursion we use `half n` to compute `half (succ (succ n))` because it is clear that `n` is structurally smaller that `succ (succ n)`.

For more sophisticated uses of recursion we may have to prove that the recursion is actually terminating but we leave this for the advanced wizard level.

6.3 Induction

Proof by induction is very closely related to structural recursion which we have just seen, it is basically the same idea but for proofs. As an example let's actually prove that `half` is the inverse of `double`:

```
theorem half_double : ∀ n : ℕ , half (double n) = n :=
begin
  assume n,
  induction n with n' ih,
  refl,
  dsimp [double,half],
  apply congr_arg succ,
  exact ih,
end
```

After `assume n` we are in the following state:

```
n : ℕ
⊢ half (double n) = n
```

Now `induction n` works very similar to `cases` (which we can also use on natural numbers) but it gives us an extra assumption when proving the successor case, which I have labelled `ih` for *induction hypothesis*. So after `induction n` we have two subgoals:

```
2 goals
case nat.zero
⊢ half (double 0) = 0

case nat.succ
n' : ℕ,
ih : half (double n') = n'
⊢ half (double (succ n')) = succ n'
```

The first one is easily disposed off using `refl`, because `half (double 0) = half 0 = 0`. The successor case is more interesting:

```
n' : ℕ,
ih : half (double n') = n'
⊢ half (double (succ n')) = succ n'
```

Here we see the extra assumption that what we want to prove for `succ n'` already holds for `n'`. Now we can apply the definitions of `double` and half using `dsimp [double,half]`:

```
n' : ℕ,
ih : half (double n') = n'
⊢ succ (half (double n')) = succ n'
```

And now we can use that `succ` preserves equality by appealing to `congr_arg succ`:

```
n' : ℕ,
ih : half (double n') = n'
⊢ half (double n') = n'
```

And we are left with a goal that exactly matches the induction hypothesis, hence we are done using `exact ih`.

This is a very easy inductive proof but it serves to show the general idea. Because every number is finitely generated from `zero` and `succ` we can *run* an inductive proof for any number by repeating the inductive step as many times as there are `succ` s and obtain a concrete proof without using induction.

There is nothing mysterious or difficult about induction itself but it is the main work horse to prove properties on inductive types like ℕ. Not all proofs are as easy as this one, and in many cases we have to think a bit to generalize the goal we want to prove so that induction goes through or we may have to prove an auxiliary theorem (a *lemma*) first to make progress. So it is not induction itself that is difficult but it is used in many situation which require some thinking.

The best way to learn how to do proofs using induction is to look at examples, so let's just do this.

6.4 Addition and its properties

While addition is an operation which you may have learned already in kindergarden it still needs to be defined. And, horror, its definition already uses recursion. They don't tell the kids this in kindergarten!

Here is the definition of `add`:

```
def add : ℕ → ℕ → ℕ
| m  zero     := m
| m  (succ n) := succ (add m n)
```

So `add m n` applies n `succ` s to `m`. We define `m + n` as `add m n`. So for example:

```
3 + 2
= add 3 2
= add 3 (succ 1)
= succ (add 3 1)
= succ (add 3 (succ 0))
= succ (succ (add 3 0))
= succ (succ (add 3 zero))
= succ (succ 3)
= 5
```

Lean defines addition by recursion over the 2nd argument, while I actually think it is better to recur over the first one, even though the output is the same because `add` is commutative (`m + n = n + m`) as we will show soon. In any case the reasons are a bit involved but I am happy to answer to explain why if you ask me, However, let's stick with the Lean definition.

Now what are the basic algebraic properties of `+`? First of all `0` is a *neutral element*, that is `n + 0 = n` and `0 + n = n`. We may think that we only need to prove one of them since addition is commutative but actually we will need exactly this property when proving commutativity. It turns out that one of the two is trivial while one of them needs induction. First the easy one:

```
theorem add_rneutr : ∀ n : ℕ, n + 0 = n :=
begin
  assume n,
  refl,
end
```

This theorem is easy because `n + 0 = n` holds by definition of `add`. However, this is not the case for the other one which does need induction:

```
theorem add_lneutr : ∀ n : ℕ, 0 + n  = n :=
begin
  assume n,
  induction n with n' ih,
  refl,
  apply congr_arg succ,
  exact ih,
end
```

Another important property of addition is that brackets don't matter, that is `(l + m) + n = l + (m + n)` this is called *associativity*. We need induction to prove this but there is a lot of choice: do we do induction over `l m` or `n`? And yes, it does matter!

If you remember the two definitions of `band` you should know that we need to analyse the argument which is used in the pattern matching. That is we need to do induction on the 2nd argument of addition which is `n` because addition matches on this (had we used my preferred definition of addition then it would be `n`).

```
theorem add_assoc : ∀ l m n : ℕ , (l + m) + n = l + (m + n) :=
begin
  assume l m n,
  induction n with n' ih,
  refl,
  dsimp [(+), nat.add],
  apply congr_arg succ,
  exact ih,
end
```

(continues on next page)

(continued from previous page)

```
```

Already the 0 case only works if we choose n:

```
l m : ℕ
⊢ l + m + 0 = l + (m + 0)
```

Both sides reduce to l+m, notice that l + m + 0 really means (l + m) + 0 which is equal to l + m by definition. But also l + (m + 0) = l + m because m + 0 = m. Hence we just say refl.

Now in the successor case:

```
l m n' : ℕ,
ih : l + m + n' = l + (m + n')
⊢ l + m + succ n' = l + (m + succ n')
```

we observe that:

```
(l + m) + succ n'
= succ ((l + m) + n')
```

and:

```
l + (m + succ n')
= l + (succ (m + n'))
= succ (l + (m + n'))
```

Hence after using congr_arg succ we are back to the induction hypothesis.

We have shown the following facts about + and 0:

- 0 is right neutral: n + 0 = n (add_rneutr),
- 0 is left neutral: 0 + n = n (add_lneutr),
- + is associative : (l + m) + n = l + (m + n) (add_assoc).

Such a structure is called a **monoid**. If you look this up on wikipedia don't get distracted by the philosophical notion with the same name. Do you know some other monoids?

However, we have not yet talked about commutativity which isn't required for a monoid. Ok how do we prove m + n = n + m? The choice of variable isn't clear here because they swap places. Hence we may as well just go for m. Here is an attempt:

The 0 case looks like this:

```
n : ℕ
⊢ 0 + n = n + 0
```

Clearly the right hand side reduces to n and hence we just need to apply add_lneutr. But the succ case is less clear:

```
n m' : ℕ,
ih : m' + n = n + m'
⊢ succ m' + n = n + succ m'
```

Again the right hand side reduces to succ (n + m') but there isn't much we can do with the left hand side. This is the case for a lemma which states that the alternative definition of + is true as a theorem, provable by induction:

```
lemma add_succ_lem : ∀ m n : ℕ, succ m + n = succ (m + n) :=
begin
  assume m n,
  induction n with n' ih,
  refl,
  apply congr_arg succ,
  exact ih,
end
```

Now we can complete the proof of `add_comm`:

```
theorem add_comm : ∀ m n : ℕ , m + n = n + m :=
begin
  assume m n,
  induction m with m' ih,
  apply add_lneutr,
  transitivity,
  apply add_succ_lem,
  apply congr_arg succ,
  exact ih,
end
```

We need to apply two steps here, which is why we use `transitivity`

```
n m' : ℕ,
ih : m' + n = n + m'
⊢ succ m' + n = n + succ m'
```

And after `transitivity` we see:

```
2 goals
n m' : ℕ,
ih : m' + n = n + m'
⊢ succ m' + n = ?m_1

n m' : ℕ,
ih : m' + n = n + m'
⊢ ?m_1 = n + succ m'
```

The `?m_1` is a placeholder which can be filled in later. We are using our lemma which forces `?m_1` to be `succ (m' + n)` and we are left with:

```
n m' : ℕ,
ih : m' + n = n + m'
⊢ succ (m' + n) = n + succ m'
```

which now can be reduced to `ih` by using `congr_arg succ`.

Using `transitivity` and tactics to do equational proofs isn't very readable, hence there is the `calc` syntax which looks like a usual equational derivation:

```
theorem add_comm : ∀ m n : ℕ , m + n = n + m :=
begin
  assume m n,
  induction m with m' ih,
  apply add_lneutr,
  calc
```

(continues on next page)

(continued from previous page)

```
    succ m' + n = succ (m' + n) : by apply add_succ_lem
    ... = succ (n + m') : by apply congr_arg succ ih
    ... = n + succ m' : by refl
end
```

The disadvantage is that we have to construct each step by hand but it is certainly more readable. Also we can make the unfolding of definitions explicit by putting in trivial steps using `refl`.

Together with the previous facts we have now shown that ℕ with + and `0` form a *commutative monoid*.

Mathematicians prefer it if you also have inverse as for the integers where for every integer `i` there is `-i` such that `i+(-i) = 0` and `(- i) + i = 0`. Such a structure is called a **group**.

6.5 Multiplication and its properties

Ok, we are doing things slowly but I won't go in as much detail this time but leave the fun for you. First of all lets define multiplication:

```
def mul : ℕ → ℕ → ℕ
| m 0      := 0
| m (succ n) := (mul m n) + m
```

And as usual we define `x * y` to stand for `mul x y`. As + was repeated `succ`, * is repeated +. That is `m * n` is `m` added `n` times, for example:

```
3 * 2
= mul 3 2
= mul 3 (succ 1)
= mul 3 1 + 3
= mul 3 (succ 0) + 3
= mul 3 0 + 3 + 3
= 0 + 3 + 3
= 6
```

What are the properties of multiplication. First we note that it also forms a commutative monoid, with `1` now playing the role of `0` for + – it is the neutral elements. We can show:

```
theorem mult_rneutr : ∀ n : ℕ, n * 1 = n :=
begin
  sorry,
end

theorem mult_lneutr : ∀ n : ℕ, 1 * n  = n :=
begin
  sorry,
end

theorem mult_assoc : ∀ l m n : ℕ , (l * m) * n = l * (m * n) :=
begin
  sorry,
end

theorem mult_comm :  ∀ m n : ℕ , m * n = n * m :=
begin
```

(continues on next page)

(continued from previous page)

```
  sorry,
end
```

This time I leave it as an exercise to prove the properties. You will certainly need to use the properties of addition we have already shown and the order in which you prove these propositions may not be the way in which I write them. Indeed, you may want to take the next properties into account as well.

Apart from addition and multiplication both being commutative monoids they also interact in an interesting way. You have will have seen this in school when you are asked to simplify an expression of the form `x*(y + z)` by *multiplying out* to `x*y + x*z`. The property is called **distributivity**. There is also a case for an empty addition, ie. when the one argument is `0` we want that `x*0 = 0`. And if we don't want to assume commutativity (indeed we may use these properties to prove it) we also need the symmetric cases, That is we have the following properties:

```
theorem mult_zero_l : ∀ n : ℕ , 0 * n = 0 :=
begin
  sorry,
end

theorem mult_zero_r : ∀ n : ℕ , n * 0 = 0 :=
begin
  sorry,
end

theorem mult_distr_r :  ∀ l m n : ℕ , l * (m + n) = l * m + l * n :=
begin
  sorry,
end

theorem mult_distr_l :  ∀ l m n : ℕ , (m + n) * l = m * l + n * l :=
begin
  sorry,
end
```

Now this structure, we have two monoids and the distributivity laws just stated and we do need to require that addition is commutative, is called a **semiring** actually in this case where multiplication is commutative too it is a **commutative semiring**. Rings and semirings are a very central structure in algebra and they are closely related to polynomials (that is expressions like `7*x^2 + x + 5` but possibly with higher exponents and several variables).

6.6 Some Algebra

Once we have established that multiplication and addition form a commutative ring we can establish many well known equations. For example the well know binomial equality, $(x + y)^2 = x^2 + 2xy + y^2$. Ok to state this lets define exponentiation, which is just repeated multiplication in the same way as multiplication is repeated addition (you may notice a pattern):

```
def exp : ℕ → ℕ → ℕ
| n zero := 1
| n (succ m) := exp n m * n
```

In this case we have to do recursion over the 2nd argument because exponentiation is not commutative like addition and multiplication.

The Lean prelude also introduces the standard notation for exponentiation, i.e. we define `x^y = exp x y`. Now we can state the binomial theorem:

```
theorem binom : ∀ x y : ℕ, (x + y)^2 = x^2 + 2*x*y + y^2 :=
begin
  sorry,
end
```

Now you can prove this using the properties we have established above but it is very laborious. You should do it at least once but luckily some clever people have implemented a tactic which does this automatically:

```
theorem binom : ∀ x y : ℕ, (x + y)^2 = x^2 + 2*x*y + y^2 :=
begin
  assume x y,
  ring,
end
```

That is `ring` automatically solves any equations which can be shown using the fact that multiplication and addition form a (semi-)ring. How does it work? You may not want to know. The basic idea is to reduce the expression to a polynomial and then it is easy to check wether two polynomials are equal The nice thing about a system like Lean is that we also generate the proof from basic principles, there is no danger of cheating.

We have just opened the door to the exciting realm of *algebra*. There is not enough time in this course to cover this in any depth but here is a table with a quick overview over different algebraic structures:

Example	Algebraic structure
Natural numbers ($\mathbb{N}$)	Semiring
Integers ($\mathbb{Z}$)	Ring
Rational numbers ($\mathbb{Q}$)	Field
Real numbers ($\mathbb{R}$)	Complete Field
Complex numbers ($\mathbb{C}$)	Algebraically complete Field

A *ring* is a semiring with *additive inverses* that is for every number $x : \mathbb{Z}$ there is $-x : \mathbb{Z}$ such that $x + (-x) = 0$ and $-x+x = 0$. Here subtraction is not a primitive operation but it is defined by adding the additive inverse $x-y := x+(-y)$.

A *field* also has the *multiplicative inverses* for all numbers different from 0. The simplest example are the rational numbers = fractions. For every $p : \mathbb{Q}$ different from 0 there is $p^{-1} : \mathbb{Q}$ such that $p * p^{-1} = 1$ and $p^{-1} * p = 1$. Note that for a fraction $\frac{a}{b} : \mathbb{Q}$ the multiplicative inverse is $(\frac{a}{b})^{-1} = \frac{b}{a}$.

The real numbers include numbers like $\sqrt{2} : \mathbb{R}$ and $\pi : \mathbb{R}$, they have the additional property that any converging infinite sequence of real numbers has a unique limit. The complex numbers include $i = \sqrt{-1} : \mathbb{C}$ have the additional feature that every polynomial equation has a solution (e.g. $x^2 + 1 = 0$) , this is called *algebraically complete*.

6.7 Ordering the numbers

Next we will look at the $\leq$ relation, which defines a *partial order* on the numbers. We say that `m ≤ n` if there is a number `k:ℕ` such that `n = k + m`. In full lean glory that is:

```
def le(m n : ℕ) : Prop :=
 ∃ k : ℕ , k + m = n

 local notation (name := le) x ≤ y := le x y
```

I hasten to add that this is not the *official* definition of $\leq$ in Lean, because they are using an inductive relation and I don't want to introduce this concept just now.

There is a precise meaning of the word `partial order`: a partial order is a relation which is reflexive, transitive and antisymmetric. Maybe you remember that we have already met the words *reflexive* and *transitive* when we looked at equality which is an *equivalence relation*. Here is a reminder what this meant but this time applied to ≤

- reflexive (`∀ x : A, x≤x`),
- transitive (`∀ x y z : A, x≤y → y≤z → x≤z`)

For an equivalence relation we also demanded that it should be symmetric (`∀ x y : A, x=y → y=x`) but this doesn't apply to ≤. Almost the opposite is true, the only situation in which we have `x≤y` and `y≤ x` is if they are actually equal. That is we have

- antisymmetry (`∀ x y : ℕ , x ≤ y → y ≤ x → x = y`)

A relation which has these three properties (reflexive, transitive and antisymmetric) is called a partial order. It is not hard to prove that ≤ is reflexive and transitive:

```
theorem le_refl : ∀ x : ℕ , x ≤ x :=
begin
  assume x,
  existsi 0,
  ring,
end

theorem le_trans : ∀ x y z : ℕ , x ≤ y → y ≤ z → x ≤ z :=
begin
  assume x y z xy yz,
  cases xy with k p,
  cases yz with l q,
  existsi (k+l),
  rewrite← q,
  rewrite← p,
  ring,
end
```

For reflexivity we choose `k=0` and exploit that `0 + x =x` and for transitivity we add the differences exploiting associativity. I am leaving all the equational reasoning to the ring tactic avoiding unnecessary detail in the proof.

The 3rd one, antisymmetry, I found a bit harder to prove and I needed to show some lemmas. But I don't want to spoil the fun and leave this to you:

```
theorem anti_sym : ∀ x y : ℕ , x ≤ y → y ≤ x → x = y :=
begin
  sorry,
end
```

We can also define < by saying that `m < n` means that `m+1 ≤ n`:

```
def lt(m n : ℕ) : Prop :=
  m+1 ≤ n

local notation (name := lt) x < y := lt x y
```

I am not going to discuss < in much detail now: it is antireflexive (`∀ n : ℕ, ¬ (n < n)`), transitive and strictly antisymmetric (`∀ m n : ℕ, m < n → n < m → false`). Indeed antireflexivity and strict antisymmetry follow from another important property of <, that it is *well-founded*. This means that any sequence starting with a number and always choosing smaller numbers, like `10 > 5 > 3 > 0` will eventually terminate. This property is important to prove the termination of algorithms which are not primitive recursive. We are not going to define *well-founded* now because it goes a bit beyond what we have learned so far.

Another important property of < is *trichotomy*, that is that any two numbers one is greater than the other or they are equal or one is smaller than the other:

```
theorem trich : ∀ m n : ℕ, m < n ∨ m = n ∨ n < m :=
begin
  sorry
end
```

A relation that is transitive, trichotomous and well-founded is called a *well-order*. Well-orderings are very useful and there is a famous theorem by Cantor that every set can be well-ordered. However, this is not accepted by intuitionists and indeed no well-ordering of the real numbers is known.

6.8 Decidability

Equality for natural numbers is decidable, that is we can actually implement it as a function into bool:

```
def eq_nat : ℕ → ℕ → bool
| zero     zero     := tt
| zero     (succ n)  := false
| (succ m) zero     := false
| (succ m) (succ n)  := eq_nat m n
```

We need to show that `eq_nat` indeed decides equality. First of all we show that `eq_nat` returns `tt` for equal numbers:

```
lemma eq_nat_ok_1 : ∀ n : ℕ , eq_nat n n = tt :=
begin
  assume n,
  induction n with n' ih,
  reflexivity,
  exact ih,
end
```

On the other hand we can also show that if `eq_nat` returns `tt` then we its arguments must be equal:

```
lemma eq_nat_ok_2 : ∀ m n : ℕ, eq_nat m n = tt → m = n :=
begin
  assume m,
  induction m with m' ih_m,
  assume n,
  induction n with n' ih_n,
  assume _,
  reflexivity,
  assume h,
  contradiction,
  assume n,
  induction n with n' ih_n,
  assume h,
  contradiction,
  assume h,
  apply congr_arg succ,
  apply ih_m,
  exact h,
end
```

Here we need a double induction. Putting everything together we can show that `eq_nat` decides equality:

```
theorem eq_nat_ok : ∀ m n : ℕ, m = n ↔ eq_nat m n = tt :=
begin
  assume m n,
  constructor,
  assume h,
  rewrite h,
  apply eq_nat_ok_1,
  apply eq_nat_ok_2,
end
```

We can do the same for $\leq$, that is we implement a function `leb : ℕ → ℕ → bool` and then show `∀ m n : ℕ, m ≤ n ↔ leb m n = tt`.

Certainly not all relations or predicates are decidable. An example for an undecidable relation is equality of functions that is given `f g : ℕ → ℕ` is `f = g`? Two functions are equal iff they agree on all arguments `f = g ↔ ∀ n :ℕ, f n = g n`. It is clear that we cannot decide this, i.e. define a function into bool, because we would have to check infinitely many inputs.

CHAPTER

SEVEN

LISTS

If you have already used a functional programming language like Haskell you know that lists are a very ubiquitous datastructure. Given `A : Type` there is `list A : Type` which is the type of finite sequences of elements of `A`. So for example `[1,2,3] : list ℕ` or `[tt] : list bool`, we can also have lists of lists so for example `[[],[0],[0,1],[0,1,2]] : list (list ℕ)` is a list of lists of numbers. However, all elements of a list must have the same type, hence something like `[1,tt]` is not permitted.

For any type `A` we always can construct the empty list `[] : list A`, which is also called `nil` (this comes from the phrase *not in list* and it was used in the first functional language LISP which was first implemented in 1959). On the other hand if we have an element `a : A` and a list `l : List A` we can form the list `a :: l : list A` with `a` in front of `l`, e.g. `1 :: [2,3] = [1,2,3]`. We call this `cons` - another heritage from LISP. Indeed the notation using `[ .. ]` is just a short-hand, e.g.:

```
[1,2,3] = 1 :: 2 :: 3 :: []
```

or using `nil` and `cons`:

```
= cons 1 (cons 2 (cons 3 nil))
```

All lists can be created from `nil` and `cons` hence lists can be defined inductively in a manner very similar to ℕ

```
 inductive list (A : Type)
| nil  : list
| cons : A → list → list
```

This declaration means:

- For any type `A : Type` There is a new type `list A : Type`,
- There are elements `nil : list A`, and given `a : A` and `l : list A` we have `cons a l`.
- All the elements of `list A` can be generated by `nil` and then applying `cons` a finite number of times,
- `nil` and `cons a l` are different elements,
- `cons` is injective, i.e. given `cons a as = cons b bs` then we know that `a = b` and `as = bs`.

7.1 Basic properties of `list`

Most of the basic properties and their proofs are very similar to the natural numbers. First of all we want to show that `nil ≠ a :: l` which corresponds to `0 ≠ succ n` and `true ≠ false` we had seen before. These properties are called *no confusion properties*. We can use `contradiction` as before for natural numbers:

```
theorem no_conf : ∀ (a:A)(l : list A), nil ≠ a :: l :=
begin
  assume a l,
  contradiction
end
```

Next we want to show injectivity of `::`. Unlike in the case for natural numbers there are two results we want to prove: given `a :: l = a' :: l'` we need to show that `a = a'` *and* `l = l'`. The second part we can prove the same way as for ℕ, instead of `pred` we define `tl` (for tail) and use this in the proof:

```
def tl : list A → list A
| [] := []
| (a :: l) := l

theorem inj_tl : ∀ (a a':A)(l l' : list A), a :: l = a' :: l' → l = l' :=
begin
  assume a a' l l' h,
  change tl (a :: l) = tl (a' :: l'),
  rewrite h,
end
```

However the first part turns out more difficult. It seems that we need to define a function `hd : list A → list A` which extracts the head of a list. But what should we do for the [] case

```
def hd : list A → list A
| [] := ?
| (a :: l) := a
```

There is really nothing we can plug in here because `A` may be empty but there is an empty list over the empty type. We could define a function that returns an error, i.e.

```
def hd : list A → option (list A)
| [] := none
| (a :: l) := some a
```

But this doesn't help us to prove injectivity. I leave this as a challenge because the only solution I know involves dependent types which I am not covering in this course.

However, the `injection` tactic which we have seen before works in this case:

```
theorem inj_hd : ∀ (a a':A)(l l' : list A), a :: l = a' :: l' → a = a' :=
begin
  assume a a' l l' h,
  injection h,
end
```

`injection` also works in the proof of `inj_tl` alleviating the need to define `tl`.

7.2 The free monoid

In the section on natural numbers we encountered a *monoid* using + and `0` on ℕ. There is a very similar monoid for lists using ++ and `[]`. Here ++ (append) is the operation that appends two lists e.g. `[1,2]++[3,4] = [1,2,3,4]`. We define it recursively:

```
definition append : list A → list A → list A
| []        l := l
| (h :: s) t := h :: (append s t)

local notation l₁ ++ l₂ := append l₁ l₂
```

I am cheating a bit here because like +, ++ is already defined in the Lean prelude.

In the definition of `append` we are using pattern matching and structural recursion on lists. This works in the same way as for natural numbers: we can recursively use `append` for the smaller list `s`. To prove that this forms a monoid we need induction. However, different from +, now left neutrality is easy because it follows from the definition:

```
theorem app_lneutr : forall l : list A, [] ++ l = l :=
begin
  assume l,
  reflexivity,
end
```

But now right neutrality requires induction:

```
theorem app_rneutr : forall l : list A, l ++ [] = l :=
begin
  assume l,
  induction l with a l' ih,
  reflexivity,
  apply congr_arg (cons a),
  exact ih,
end
```

Induction for lists means to prove something for all lists you prove it for the empty list `[]` and assuming that it holds for a list `l` you show that it holds for `a :: l` for any `a`. Comparing the proof with the one for left neutrality for + we see that it is almost the same proof replacing `congr_arg succ` with `congr_arg (cons a)`.

The switch between left and right neutrality is a consequence of that we have defined + by recursion over the 2nd argument while we have defined ++ by recursion over the first. We could have defined + by recursion over the first argument and then this difference would have disappeared (I prefer this but the designers of the lean standard library had different ideas). We couldn't have defined ++ by recursion over the 2nd argument - do you see why?

The proof of associativity of ++ again is very similar to the one for +. The main difference is that now we do induction over the first argument instead of the last - again this is caused by the fact that we now use recursion over the first instead of the 2nd argument.

```
theorem app_assoc :
      forall l₁ l₂ l₃ : list A , (l₁ ++ l₂) ++ l₃ = l₁ ++ (l₂ ++ l₃) :=
begin
  assume l₁ l₂ l₃,
  induction l₁ with a l₁' ih,
  reflexivity,
  dsimp [(++),list.append],
  apply congr_arg (cons a),
  exact ih,
end
```

For + we also proved commutativity that is `m + n = n + m`, but clearly this doesn't hold for ++, e.g. `[1,2] ++ [3,4] = [1,2,3,4]` while `[3,4] ++ [1,2] = [3,4,1,2]`.

Indeed `list A` with ++ and `[]` is the *free monoid* over `A` which intuitively means that only the monoid equations hold but now additional laws like commutativity. In this sense this monoid is free not to follow any laws apart from the monoid ones.

7.3 Reverse

Since the list monoid is not commutative, order matters. In particular we can *reverse* a list. That is we are going to define a function which given a list like `[1,2,3]` produces the list with the elements in reverse order, in this case `[3,2,1]`.

How do we reverse a list? Recursively! The reverse of the empty list is the empty list and the reverse of a list of the form `a :: l` is the reverse of `l` with `a` put in the end. So for example the reverse of `[1,2,3] = 1 :: [2,3]` is the reverse of `[2,3]`, i.e. `[3,2]` with `1` put at the end giving `[3,2,1]`.

To define reverse we need an auxiliary operation which puts an element at the end. We call this operation `snoc` because it is the reverse of `cons`. We could define `snoc` using ++, but for our purposes it is slightly more convenient to define it directly using recursion:

```
def snoc : list A → A → list A
| [] a := [a]
| (a :: as) b := a :: (snoc as b)

#reduce (snoc [1,2] 3)
```

Using `snoc` we are ready to implement the recursive recipe for reversing a list:

```
def rev : list A → list A
| [] := []
| (a :: as) := snoc (rev as) a

#reduce rev [1,2,3]
```

A central property of *rev* is that it is self-inverse that is if we reverse a list twice we obtain the original list, e.g. `rev (rev [1,2,3])) = rev [3,2,1] = [1,2,3]`

Ok, let's try and prove this using list induction:

```
theorem revrev : ∀ as : list A , rev (rev as) = as :=
begin
  assume as,
  induction as with a as' ih,
  reflexivity,
  dsimp [rev],
  sorry,
```

I got stuck in this proof, I am in the following situation:

```
ih : rev (rev as') = as'
⊢ rev (snoc (rev as') a) = a :: as'
```

I cannot apply my induction hypothesis because my current goal doesn't contain an expression of the form `rev (rev as')`. However, I should be able to exploit the fact that to reverse a snoc is the same as a cons of the reverse. Then we can reason:

```
rev (snoc (rev as') a)
= a :: rev (rev as')
= a :: as'
```

This leads us to proving the following lemma:

```
theorem revsnoc :
              ∀ a : A, ∀ as : list A, rev (snoc as a) = a :: rev as :=
begin
  assume a as,
  induction as with b as' ih,
  reflexivity,
  dsimp [snoc,rev],
  rewrite ih,
  dsimp [snoc],
  reflexivity,
end
```

This is exactly what was missing to complete the proof of `revrev`:

```
theorem revrev : ∀ as : list A , rev (rev as) = as :=
begin
  assume as,
  induction as with a as' ih,
  reflexivity,
  dsimp [rev],
  rewrite revsnoc,
  rewrite ih,
end
```

This is a nice example about the art of proving which is a bit like putting stepping stones into a stream to cross it without getting wet feet. When getting stuck with our induction, then looking at the point where we are stuck often leads us to identifying another property which we can prove and which helps us to complete the original proof. There is no fixed method to identify a good auxiliary property (a lemma) it is a skill which improves by practice.

Here is another problem to do with reverse which you can use to practice this skill: if you have attended Prof Hutton's Haskell course you will know that the above definition of reverse is very inefficient, indeed it has a quadratic complexity. A better definition is the following:

```
def revaux : list A → list A → list A
| [] bs := bs
| (a :: as) bs := revaux as (a :: bs)

def fastrev (l : list A) : list A
  := revaux l []

#reduce fastrev [1,2,3]
```

`fastrev` only has linear complexity. However, we should convince ourselves that it behaves in the same way as `rev`, that is we should prove the following theorem:

```
theorem fastrev_thm : ∀ as : list A , fastrev as = rev as :=
begin
  sorry,
end
```

I leave it as an exercise to figure out what lemma(s) we need. Also it may be useful to employ some properties we have already proven.

7.4 Insertion sort

Finally, we are going to look at a slightly more interesting algorithm: insertion-sort. Insertion-sort is quite an inefficient algorithm because it has quadratic complexity while the best sorting algorithms have a $n \log n$ complexity. However, insertion-sort is especially easy to implement and verify.

To keep things simple I will only sort lists of natural numbers wrt `≤ : ℕ → ℕ → Prop` using the fact that we have an implementation in form of `ble : ℕ → ℕ → bool`. That means for the following we assume in particular that `ble` is decidable, i.e.

```
lemma Le2ble : ∀ m n : ℕ, m ≤ n → ble m n = true := sorry
lemma ble2Le : ∀ m n : ℕ, ble m n = true → m ≤ n := sorry
```

7.4.1 Implementing sort

So our goal is to implement a function `sort : list ℕ → list ℕ` that sorts a given list, e.g. `sort [6,3,8,2,3] = [2, 3, 3, 6, 8]`.

Insertion sort can be easily done with a deck of cards: you start with an unsorted deck and you take a card form the top each time and insert it at the appropriate position in the already sorted pile. Inserting a card means you have to go through the sorted pile until you find a card that is greater and then you insert it just before that. Ok, there is also the special case that the sorted pile is empty in which case you just make a pile out of one card. Here is the recursive function implementing the algorithm:

```
def ins : ℕ → list ℕ → list ℕ
| a [] := [a]
| a (b :: bs) := if ble a b then a :: b :: bs else b::(ins a bs)

#reduce ins 6 [2, 3, 3, 8]
```

Using `ins` it is easy to implement `sort` by inserting one element after the other, recursively:

```
def sort : list ℕ → list ℕ
| [] := []
| (a :: as) := ins a (sort as)

#reduce (sort [6,3,8,2,3])
```

7.4.2 Specifying sort

To verify that the algorithm indeed sorts we need to specify what it means for a list to be sorted. That is we need to define a predicate `Sorted : list ℕ → Prop`. To define this we also need an auxiliary predicate which tells us that an element we want to add to a sorted list is smaller then the first element of the list `Le_list : ℕ → list ℕ → Prop`.

To define these predicates we are going to use inductive definitions of predicates which are similar to the inductive datatypes we have already seen. The basic idea is that we state some rules how to prove the predicate like we used constructors for inductive types like `zero`, `succ` or `nil` and `cons`. We also presume that using those rules is the only way to prove the predicate. So for example we define `Le_list`:

```
inductive Le_list : ℕ → list ℕ → Prop
| le_nil : ∀ n:ℕ, Le_list n []
| le_cons : ∀ m n : ℕ, ∀ ns : list ℕ, m ≤ n → Le_list m (n :: ns)
```

That is the we can prove that any element fits into the empty list and that if we have an element that is smaller then the head of a list this is ok too and these are the only ways to establish this fact.

So for example we can prove:

```
example : Le_list 3 [6,8] :=
begin
 apply le_cons,
 apply ble2Le,
 trivial,
end
```

Note that I am using one of the direction of the proof that `ble` decides ≤. This was actually left as an exercise in the chapter on natural numbers.

Using the principle that the two rules `le_nil` and `le_cons` are the only way to prove `Le_list` we can also *invert* `le_cons`:

```
lemma le_cons_inv : ∀ m n : ℕ, ∀ ns : list ℕ,
              Le_list m (n :: ns) → m ≤ n :=
begin
  assume m n ns h,
  cases h with _ _ _ _ mn,
  exact mn
end
```

I am using `cases` here again. The idea is that the only way to prove `Le_list m (n :: ns)` are `le_nil` and `le_cons` but `le_nil` can be ruled out since it only proves it for the empty list. Hence we must have used `le_cons` but then we know that we have assumed `mn : m ≤ n` and we can use this.

The definition of `Le_list` is not recursive, hence `cases` is sufficient. However, the definition of `Sorted` is actually recursive:

```
inductive Sorted : list ℕ → Prop
| sorted_nil : Sorted []
| sorted_cons : ∀ n : ℕ, ∀ ns : list ℕ, Le_list n ns
              → Sorted ns → Sorted (n :: ns)
```

We say that the empty list is sorted, and given a sorted list and an element that is smaller than the first then the list obtained by consing this element in front of the list is sorted. And these are the only ways to prove sortedness.

7.4.3 Verifying sort

We now want to prove that `sort` only produces sorted outputs. What principle are we going to use? Exactly: induction over lists. The base case is easy: the empty list is sorted by definition.

```
theorem ins_sort_sorts : ∀ ns : list ℕ, Sorted (sort ns) :=
begin
  assume ns,
  induction ns with a as' ih,
  apply sorted_nil,
  dsimp [sort],
  sorry
end
```

In the cons case we are left in the following state:

```
ih : Sorted (sort as')
⊢ Sorted (ins a (sort as'))
```

We know that `sort` sorts for `as'` and from this we have to conclude that also `ins a (sort as')` is sorted. This suggests the following lemma:

```
lemma ins_lem : ∀ n : ℕ, ∀ ns : list ℕ, Sorted ns → Sorted (ins n ns) :=
begin
  assume n ns sns,
  sorry,
end
```

To prove this lemma we actually have a choice we can either do induction over the list `ns` or the derivation of `Sorted ns`. I found it is easier to the former.

```
lemma ins_lem : ∀ n : ℕ, ∀ ns : list ℕ, Sorted ns → Sorted (ins n ns) :=
begin
  assume n ns sns,
  induction ns,
  sorry,
end
```

In the base case we need to show `Sorted (ins n nil)` which reduces to `Sorted [n]`. It is easy to show that singletons (lists with just one element) are always sorted:

```
lemma sorted_sgl : ∀ n : ℕ, Sorted [n] :=
begin
  assume n,
  apply sorted_cons,
  apply le_nil,
  apply sorted_nil,
end
```

Ok this kills the base case, the cons case is certainly more difficult:

```
lemma ins_lem : ∀ n : ℕ, ∀ ns : list ℕ, Sorted ns → Sorted (ins n ns) :=
begin
  assume n ns sns,
  induction ns,
  apply sorted_sgl,
  dsimp [ins],
  sorry,
end
```

We end up with the following goal:

```
ns_ih : Sorted ns_tl → Sorted (ins n ns_tl),
sns : Sorted (ns_hd :: ns_tl)
⊢ Sorted (ite ↥(ble n ns_hd) (n :: ns_hd :: ns_tl) (ns_hd :: ins n ns_tl))
```

`ite` is just the internal code for if-then-else. Clearly the proof is stuck at the condition. We need to do a case analysis on the outcome of `ble n ns_hd`. The `tt` case is pretty straightforward, so let's do it first

```
lemma ins_lem : ∀ n : ℕ, ∀ ns : list ℕ, Sorted ns → Sorted (ins n ns) :=
begin
  assume n ns sns,
  induction ns,
```

(continues on next page)

(continued from previous page)

```
  apply sorted_sgl,
  dsimp [ins],
  cases eq : ble n ns_hd,
  sorry,
  apply sorted_cons,
  apply le_cons,
  apply ble2Le,
  exact eq,
  exact sns
end
```

The `ff` case is more subtle, we are in the following situation:

```
ns_ih : Sorted ns_tl → Sorted (ins n ns_tl),
eq : ble n ns_hd = ff,
sns_a_1 : Sorted ns_tl,
sns_a : Le_list ns_hd ns_tl
⊢ Sorted (ns_hd :: ins n ns_tl)
```

We know that `ble n ns_hd = ff` and hence `¬ (n ≤ ns_hd)` but we actually need `ns_hd ≤ n`. This actually follows from trichotomy, which we discussed in the section on natural numbers, so let's just assume it in the moment:

```
lemma ble_lem : ∀ m n : ℕ , ble m n = ff → n ≤ m := sorry
```

However, we still need another lemma because we need to know that if `ns_hd ≤ n` then it will still fit in front of `ins n ns_tl`, that is `Le_list ns_hd is n ns_tl` holds. This leads to another lemma:

```
lemma ins_lem_le_list : ∀ m n : ℕ, ∀ ns : list ℕ , n ≤ m
              → Le_list n ns → Le_list n (ins m ns) :=
begin
  sorry,
end
```

You see what I mean with the stepping stones? Anyway, now we have all the stones together we can pass to the other side - here is the complete proof:

```
lemma sorted_sgl : ∀ n : ℕ, Sorted [n] :=
begin
  assume n,
  apply sorted_cons,
  apply le_nil,
  apply sorted_nil,
end

lemma ins_lem_le_list : ∀ m n : ℕ, ∀ ns : list ℕ , n ≤ m → Le_list n ns → Le_list n
↪(ins m ns) :=
begin
  assume m n ns nm nns,
  cases ns with hd tl,
  apply le_cons,
  exact nm,
  dsimp [ins],
  cases eq : ble m hd,
  cases nns with _ _ _ _ nhd,
  apply le_cons,
  exact nhd,
```

(continues on next page)

(continued from previous page)

```
  apply le_cons,
  exact nm,
end

lemma ins_lem : ∀ n : ℕ, ∀ ns : list ℕ, Sorted ns → Sorted (ins n ns) :=
begin
  assume n ns sns,
  induction ns,
  apply sorted_sgl,
  dsimp [ins],
  cases eq : ble n ns_hd,
  cases sns with nx nsx lex sx,
  apply sorted_cons,
  apply ins_lem_le_list,
  apply ble_lem,
  exact eq,
  exact lex,
  apply ns_ih,
  exact sx,
  apply sorted_cons,
  apply le_cons,
  apply ble2Le,
  exact eq,
  exact sns
end

theorem ins_sort_sorts : ∀ ns : list ℕ, Sorted (sort ns) :=
begin
  assume ns,
  induction ns with a as' ih,
  apply sorted_nil,
  dsimp [sort],
  apply ins_lem,
  exact ih,
end
```

7.4.4 Permutations

Are we done now? Have we verified `sort`? That is if you buy a tin and it says we have proven the theorem `ins_sort_sorts` are you then satisfied that the program `sort` does the job of sorting?

Not so! We have missed one important aspect of `sort`, namely that the output should be a rearrangement, i.e. a *permutation* of the input. Otherwise an implementation of `sort` that always returns the empty list would be ok.

We can specify the relation `Perm` that one list is a permutation of another inductively, using an auxiliary relation `Insert` which means that a list is obtained form another by inserting an element somewhere.

```
inductive Insert : A → list A → list A → Prop
| ins_hd : ∀ a:A, ∀ as : list A,Insert a as (a :: as)
| ins_tl : ∀ a b:A, ∀ as as': list A, Insert a as as' → Insert a (b :: as) (b :: as')

inductive Perm : list A → list A → Prop
| perm_nil : Perm [] []
| perm_cons : ∀ a : A, ∀ as bs bs' : list A,Perm as bs → Insert a bs bs' → Perm (a␣
↪:: as) bs'
```

Using these definitions it is relatively straightforward to establish that sort permutes its input:

```
theorem sort_perm : ∀ ns : list ℕ , Perm ns (sort ns) :=
begin
  sorry
end
```

I leave it as an exercise to complete the proof which also involves identifying a rather obvious lemma.

We would expect `Perm` to be an equivalence relation, i.e. reflexive, transitive and symmetric. While the proof of reflexivity is straightforward, the other two are rather hard. I leave this as a challenge.

7.4.5 Parametrisation

I have presented sort for a rather special case, i.e. for $\leq$ on the natural numbers. However, all my constructions and proofs should work in a general case. Can you parametrize both the function and the proof so that it is applicable in much more generality?

CHAPTER

EIGHT

TREES

Trees are everywhere, there isn't just one type of trees but many datatypes that can be subsumed under the general concept of trees. Indeed the types we have seen in the previous chapters (natural numbers and lists) are special instances of trees.

Trees are also the most dangerously underused concept in programming. Bad programmers tend to try to do everything with strings, unaware how much easier it would be with trees. You don't have to be a functional programmer to use trees, trees can be just as easily defined in languages like Python - see my computerphile video.

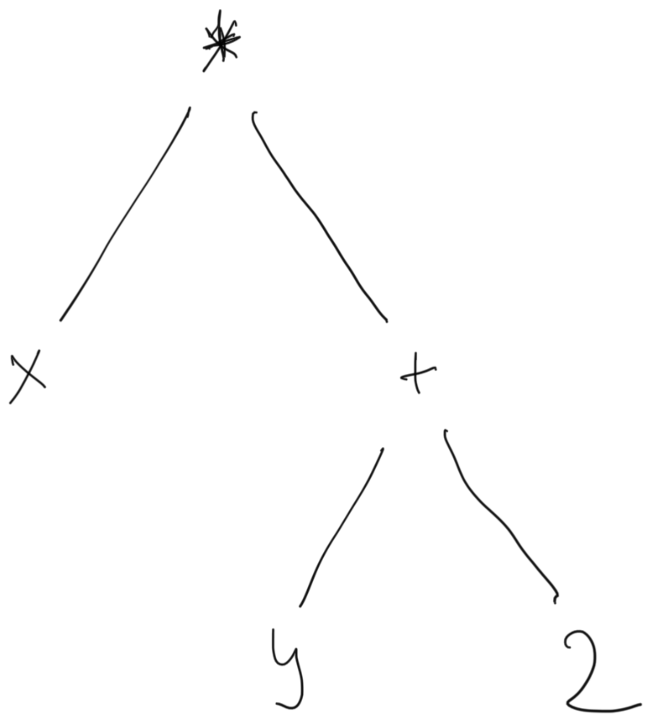
In this chapter we are going to look at two instances of trees: expression trees and sorted trees.

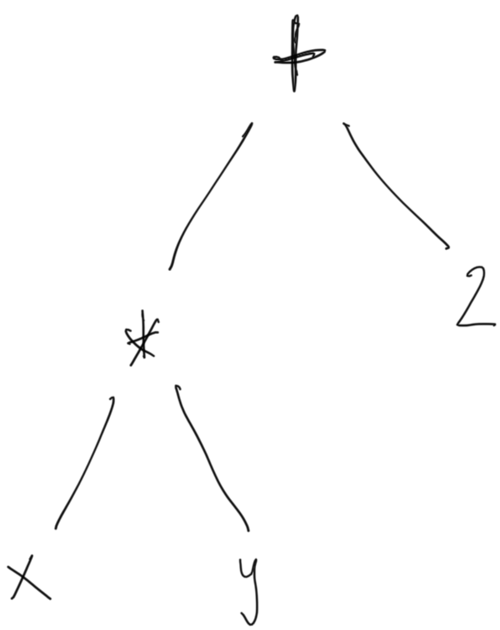
Expression trees are used to encode the syntax of expressions, we are going to define an interpreter and a compiler which compiles expression into the code for a simple stack machine and then show that the compiler is correct wrt the interpreter. This is an extended version of *Hutton's razor* an example invented by Prof Hutton.

The 2nd example is tree-sort: an efficient alternatve to insertion-sort. From a list we produce a sorted tree and then we turn this into a sorted list. Actually, tree-sort is just quicksort in disguise.

8.1 Expression trees

I am trying to hammer this in: when you see an expression like `x * (y + 2)` or `(x * y) + 2` then try to see the tree. Really expressions are a 2-dimensional structure but we use a 1-dimensional notation with brackets to write them.





But it isn't only about seeing the tree we can turn this into a datatype, indeed an inductive datatype.

```
inductive Expr : Type
```

(continues on next page)

(continued from previous page)

```
| const : ℕ → Expr
| var : string → Expr
| plus : Expr → Expr → Expr
| times : Expr → Expr → Expr

def e1 : Expr
  := times (var "x") (plus (var "y") (const 2))

def e2 : Expr
  := plus (times (var "x") (var "y")) (const 2)
```

An expression is either a constant, a variable, a plus-expression or a times-expression. To construct a constant we need a number, for variables we need a string and for both plus and times we need two expressions which serve as the first and second argument. The last two show that expression tress are recursive.

I am not going to waste any time to prove no-confusion and injectivity, e.g.

```
theorem no_conf : ∀ n : ℕ, ∀ l r : Expr, const n ≠ plus l r :=
begin
  sorry,
end

theorem inj_plus_l : ∀ l r l' r' : Expr , plus l r = plus l' r' → l=l' :=
begin
  sorry,
end
```

Btw the name of the theorem `inj_plus` is a bit misleading: it is the tree constructor `plus` that is injective not the operation +. Actually is + injective?

8.1.1 Evaluating expressions

Instead let's *evaluate* expressions! To do this we need an assignment from variable names (i.e. strings) to numbers. For this purpose I introduce a type of *environments* - I am using functions to represent environments.

```
def Env : Type
  := string → ℕ

def my_env : Env
| "x" := 3
| "y" := 5
| _ := 0

#reduce my_env "y"
```

The environment `my_env` assigns to `"x"` the number `3`, to `y` the number `5` and `0` to all other variables. Really I should have introduced some error handling for undefined variables but for the sake of brevity I am going to ignore this. To look up a variable name we just have to apply the function, e.g. `my_env "y"`

Ok, we are ready to write the evaluator for expressions which gets an expression and an environment and returns a number. And it uses - oh horror - recursion on trees.

```
def eval : Expr → Env → ℕ
| (const n) env := n
```

(continues on next page)

(continued from previous page)

```
| (var s) env := env s
| (plus l r) env := (eval l env) + (eval r env)
| (times l r) env := (eval l env) * (eval r env)

#reduce eval e1 my_env

#reduce eval e2 my_env
```

`eval` looks at the expression: if it is a constant it just returns the numerical values of the constant, it looks up variable in the environment and to evaluate a plus or a times we first recursively evaluate the subexpressions and then add them or multiply them together.

I hope you are able to evaluate the two examples `e1` and `e2` in your head before checking wether you got it right.

8.1.2 A simple Compiler

To prove something interesting let's implement a simple compiler. Our machine code is a little stack machine. We first define the instructions:

```
inductive Instr : Type
| pushC : ℕ → Instr
| pushV : string → Instr
| add : Instr
| mult : Instr

open Instr

def Code : Type
  := list Instr
```

We can push a constant or a variable from the environment onto the stack and we can add or multiply the top two items of the stack which has the effect of removing them and replacing them with their sum or product. The machine code is just a sequence of instructions.

We define a `run` function that executes a piece of code, returning what is on the top of the stack which is represented as a list of numbers.

```
def Stack : Type
  := list ℕ

def run : Code → Stack → Env → ℕ
| [] [n] env := n
| (pushC n ::c) s env := run c (n :: s) env
| (pushV x ::c) s env := run c (env x :: s) env
| (add :: c) (m :: n :: s) env := run c ((n + m) :: s) env
| (mult :: c) (m :: n :: s) env := run c ((n * m) :: s) env
| _ _ _ := 0
```

The function `run` analyzes the first instruction (if there is one) and modifies the stack accordongly and then rns the remaining instructions. Again no error handling, if something is wrong, I return 0. This calls for some serious refactoring! But not now.

As an example let's run some code that computes the values of `e1`:

```
def c1 : Code
:= [pushV "x",pushV "y",pushC 2,add,mult]

#eval run c1 [] my_env
```

Now I have compiled `e1` by hand to `c1` but certainly we can do this automatically. Here is the first version of a compiler:

```
def compile_naive : Expr → Code
| (const n) := [pushC n]
| (var x) := [pushV x]
| (plus l r) := (compile_naive l) ++ (compile_naive r) ++ [add]
| (times l r) := (compile_naive l) ++ (compile_naive r) ++ [mult]
```

The compiler translates `const` and `var` into the corresponding push instructions, and for `plus` and `times` it creates code for the subexpression and inserts an `add` or a `mult` instruction afterwards.

The naive compiler is inefficient due to the use of ++ which has to traverse the already created code each time. And it is actually a bit harder to verify because we need to exploit the fact that lists are a monoid. However, there is a nice trick to make the compiler more efficient *and* easier to verify. We add an extra argument to the compiler which is the code that should be inserted after the code for the expression we are just compiling (the *continuation*). We end up with:

```
def compile_aux : Expr → Code → Code
| (const n) c := pushC n :: c
| (var x) c := pushV x :: c
| (plus l r) c := compile_aux l (compile_aux r (add :: c))
| (times l r) c := compile_aux l (compile_aux r (mult :: c))

def compile (e :Expr) : Code
  := compile_aux e []

#reduce run (compile e1) [] my_env

#reduce run (compile e2) [] my_env
```

This version of the compiler is more efficient because it doesn't need to traverse the code it already produced. Indeed, this is basically the same issue with `rev` vs `fastrev`. The other advantage is that we don't need to use any properties of ++ in the proof because we aren't using it!

8.1.3 Compiler correctness

We can see looking at the examples that `compile` and `run` produces the same results as `eval` but we woud like to prove this, i.e. the correctness of the compiler.

```
theorem compile_ok : ∀ e : Expr, ∀ env : Env,
              run (compile e) [] env = eval e env :=
begin
  sorry,
end
```

However, we won't be able to prove this directly because here we state a property which only holds for the empty stack but once we compile a complex expression the stack won't be empty anymore.

This means we have to find a stronger proposition for which the induction goes through and which implies the proposition we actually want to prove. This is known as *induction loading*.

Clearly we need to prove a statement for `compile_aux`, namely that running `compile_aux` for some expression is the same as evaluating the expression and putting the result on the stack. This implies the statement we want to prove for the special case that both the remaining code and the stack are empty.

```
lemma compile_aux_ok :
      ∀ e : Expr, ∀ c : Code, ∀ s : Stack, ∀ env : Env,
      run (compile_aux e c) s env = run c ((eval e env) :: s) env  :=
begin
  assume e,
  induction e,
  sorry,
```

I have already started the proof. We are going to do induction over the expression `e`. After this we are in the following state:

```
e : ℕ
⊢ ∀ (c : Code) (s : Stack) (env : Env), run (compile_aux (const e) c) s env = run c␣
↪(eval (const e) env :: s) env

case Expr.var
e : string
⊢ ∀ (c : Code) (s : Stack) (env : Env), run (compile_aux (var e) c) s env = run c␣
↪(eval (var e) env :: s) env

case Expr.plus
e_a e_a_1 : Expr,
e_ih_a : ∀ (c : Code) (s : Stack) (env : Env), run (compile_aux e_a c) s env = run c␣
↪(eval e_a env :: s) env,
e_ih_a_1 : ∀ (c : Code) (s : Stack) (env : Env), run (compile_aux e_a_1 c) s env = run␣
↪c (eval e_a_1 env :: s) env
⊢ ∀ (c : Code) (s : Stack) (env : Env),
    run (compile_aux (plus e_a e_a_1) c) s env = run c (eval (plus e_a e_a_1) env ::␣
↪s) env

case Expr.times
e_a e_a_1 : Expr,
e_ih_a : ∀ (c : Code) (s : Stack) (env : Env), run (compile_aux e_a c) s env = run c␣
↪(eval e_a env :: s) env,
e_ih_a_1 : ∀ (c : Code) (s : Stack) (env : Env), run (compile_aux e_a_1 c) s env = run␣
↪c (eval e_a_1 env :: s) env
⊢ ∀ (c : Code) (s : Stack) (env : Env),
    run (compile_aux (times e_a e_a_1) c) s env = run c (eval (times e_a e_a_1) env␣
↪:: s) env
```

We see that we have four cases, one for each of the constructors and in the recursive cases for `plus` ad `mult` I have induction hypothesis which say that my theorem holds for the left and right sub expressions.

I don't want to use the names generated by lean (like `e_ih_a_1`) but using `with` here would be also getting a bit complicated given all the names we are using. Hence I am going to use `case` for each of the cases which allows my to introduce the variables separately. I also have to use `{..}` to turn the proof for each case into a block.

Hence our proof is going to look like this:

```
lemma compile_aux_ok : ∀ e : Expr, ∀ c : Code, ∀ s : Stack, ∀ env : Env,
      run (compile_aux e c) s env = run c ((eval e env) :: s) env
      :=
begin
  assume e,
  induction e,
```

(continues on next page)

(continued from previous page)

```
  sorry,
  case const : n {
  sorry, },
  case var : name {
  sorry, },
  case plus : l r ih_l ih_r {
  sorry,  },
  case times : l r ih_l ih_r {
  sorry,}
end
```

The cases for `const` and `var` are easy, we just need to appeal to `reflexivity,`. The cases for `plus` and `mult` are virtually identical (another case for refactoring, in this case for a proof). Let's have a look at `plus`:

```
l r : Expr,
ih_l : ∀ (c : Code) (s : Stack) (env : Env), run (compile_aux l c) s env = run c (eval␣
↪l env :: s) env,
ih_r : ∀ (c : Code) (s : Stack) (env : Env), run (compile_aux r c) s env = run c (eval␣
↪r env :: s) env,
c : Code,
s : Stack,
env : Env
⊢ run (compile_aux (plus l r) c) s env = run c (eval (plus l r) env :: s) env
```

By unfolding the definition of `compile_aux (plus l r) c` we obtain:

```
run (compile_aux (plus l r) c) s env
= run (compile_aux l (compile_aux r (add :: c))) s env
```

And now we can use `ih_l` to push the value of `l` on the stack:

```
... = run (compile_aux r (add :: c)) ((eval l env) :: s) env
```

So indeed the value of `l` has landed on the stack, and now we can use `ih_r`

```
... = run (add :: c) ((eval r env) :: (eval l env) :: s) env
```

We are almost done, we can run `run` one step

```
... = run c (((eval l env) + (eval r env)) :: s) env
```

And now working backward from the definition of `eval` we arrive on the other side of the equation:

```
...  = run c (eval (plus l r) env :: s) env
```

Ok here is the proof so far using `calc` for the reasoning above:

```
lemma compile_aux_ok : ∀ e : Expr, ∀ c : Code, ∀ s : Stack, ∀ env : Env,
      run (compile_aux e c) s env = run c ((eval e env) :: s) env
      :=
begin
  assume e,
  induction e,
  sorry,
case const : n {
```

(continues on next page)

(continued from previous page)

```
  assume c s env,
  reflexivity,},
  case var : name {
  assume c s env,
  reflexivity,},
  case plus : l r ih_l ih_r {
  assume c s env,
  dsimp [compile_aux],
  calc
    run (compile_aux (plus l r) c) s env
    = run (compile_aux l (compile_aux r (add :: c))) s env : by refl
    ... = run (compile_aux r (add :: c)) ((eval l env) :: s) env
              : by rewrite ih_l
    ... = run (add :: c) ((eval r env) :: (eval l env) :: s) env
              : by rewrite ih_r
    ... = run c (((eval l env) + (eval r env)) :: s) env : by refl
    ...  = run c (eval (plus l r) env :: s) env : by refl,
  },
  case times : l r ih_l ih_r {
  sorry,}
end
```

I leave it to you to fill in the case for `times`.

You may have noticed that I didn't introduce all the assumptions in the beginning. Could I have done the proof starting with:

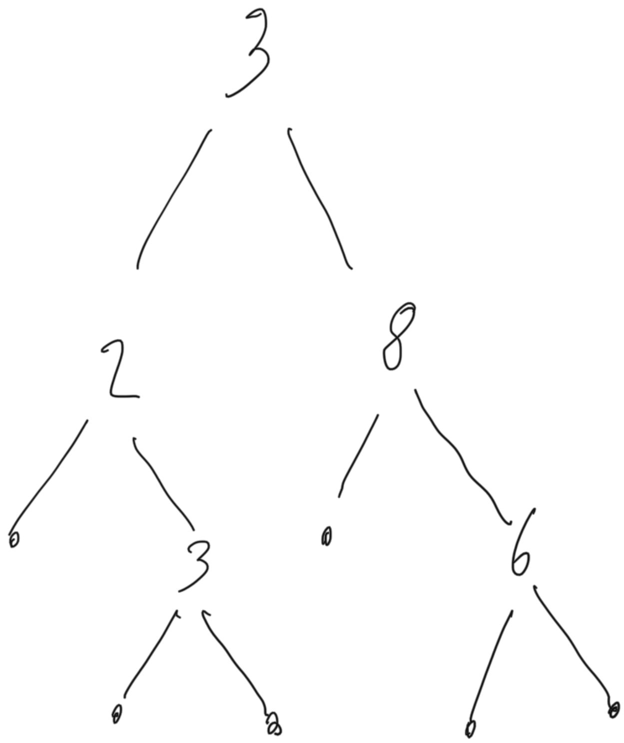
```
lemma compile_aux_ok : ∀ e : Expr, ∀ c : Code, ∀ s : Stack, ∀ env : Env,
      run (compile_aux e c) s env = run c ((eval e env) :: s) env
      :=
begin
  assume e c s env,
  induction e,
  sorry,
```

It seems I would have avoided the repeated `assume c s env,` or is there a problem? Try it out.

8.2 Tree sort

Finally we will look at another application of trees: sorting. The algorithm I am describing is tree sort and as I already said it is a variation of quicksort.

The idea is that we turn a list like our favorite example `[6,3,8,2,3]` into a tree like this one:



The nodes of the tree are labelled with numbers and the tree is sorted in the sense that at each node all the labels in the left subtree are less or equal to the number at the current node and all the ones in the right subtree are greater or equal. And once we flatten this tree into a list we get the sorted list `[2,3,3,6,8]`.

8.2.1 Implementing tree sort

First of all we need to define the type of binary trees with nodes labelled with natural numbers:

```
inductive Tree : Type
| leaf : Tree
| node : Tree → ℕ → Tree → Tree
```

To build a sorted tree from a list we need to write a function that inserts an element into a sorted tree, preserving sortedness. We define this function by recursion over trees:

```
def ins : ℕ → Tree → Tree
| n leaf := node leaf n leaf
| n (node l m r) :=
              if ble n m
              then node (ins n l) m r
              else node l m (ins n r)
```

Here we query the function `ble` which we have already seen earlier to decide wether to recursively insert the number into the right or left subtree.

To turn a list into a sorted tree we need to *fold* `ins` over the list, mapping the empty list to a leaf.

```
def list2tree : list ℕ → Tree
| [] := leaf
| (n :: ns) := ins n (list2tree ns)

#reduce list2tree [6,3,8,2,3]
```

Now all what is left to do to do is to implement a function that *flattens* a tree into a list:

```
def tree2list : Tree → list ℕ
| leaf := []
| (node l m r) := tree2list l ++ m :: tree2list r
```

Putting both together we have constructed a sorting function on lists - treesort:

```
def sort (ns : list ℕ) : list ℕ
  := tree2list (list2tree ns)

#reduce (sort [6,3,8,2,3])
```

8.2.2 Verifying tree sort

To verify that tree sort returns a sorted list we have to specify what a sorted tree is. To do this we need to be able to say things like all the nodes in a tree are smaller or greater that a number. We can do this even more generally by defining a higher order predicate that applies a given predicate to all nodes of a tree:

```
inductive AllTree (P : ℕ → Prop) : Tree → Prop
| allLeaf : AllTree leaf
| allNode : ∀ l r : Tree, ∀ m : ℕ,
              AllTree l → P m → AllTree r → AllTree (node l m r)
```

That is `AllTree P t` holds if the predicate `P` holds for all the numbers in the nodes of `t`.

Using `AllTree` we can define `SortedTree`:

```
inductive SortedTree : Tree → Prop
| sortedLeaf : SortedTree leaf
| sortedNode : ∀ l r : Tree, ∀ m : ℕ,
               SortedTree l → AllTree (λ x:ℕ, x ≤ m) l
             → SortedTree r → AllTree (λ x:ℕ, m ≤ x) r
             → SortedTree (node l m r)
```

We are now ready to state the correctness of `sort` which is the same as for insertion sort using the predicate `Sorted` on lists that we have define in the previous chapter:

```
theorem tree_sort_sorts : ∀ ns : list ℕ, Sorted (sort ns) :=
begin
sorry,
end
```

It is not difficult to identify the two lemmas we need to show:

- list2tree produces a sorted tree (list2tree_lem)
- tree2list maps a sorted tree into a sorted list (tree2list_lem)

Hence the top-level structure of our proof looks like this:

```
lemma list2tree_lem : forall l : list ℕ, SortedTree (list2tree l) :=
begin
  sorry,
end

lemma tree2list_lem : ∀ t : Tree, SortedTree t → Sorted (tree2list t) :=
begin
  sorry
end

theorem tree_sort_sorts : ∀ ns : list ℕ, Sorted (sort ns) :=
begin
  assume ns,
  dsimp [sort],
  apply tree2list_lem,
  apply list2tree_lem,
end
```

Since you have now seen enough proofs I will omit the gory details but only tell you the lemmas (stepping stones). First of all we want to prove list2tree_lem by induction over lists. Hence another lemma pops up:

```
lemma ins_lem : ∀ t : Tree,∀ n:ℕ,SortedTree t → SortedTree (ins n t) :=
begin
  sorry,
end
```

This we need to prove by induction over trees. At some point we need a lemma about the interaction of ins with AllTree, I used the following:

```
lemma insAllLem : ∀ P : ℕ → Prop, ∀ t : Tree, ∀ n : ℕ,
              AllTree P t → P n → AllTree P (ins n t) :=
begin
  sorry,
end
```

Again this just requirea tree induction. To prove the other direction it is helpfull to also introduce a higher order predicate for lists:

```
inductive AllList (P : ℕ → Prop) : list ℕ → Prop
| allListNil : AllList []
| allListCons : ∀ n : ℕ, ∀ ns : list ℕ, P n → AllList ns → AllList (n :: ns)
```

And then I prove a lemma:

```
lemma AllTree2list : ∀ P : ℕ → Prop, ∀ t : Tree,
              AllTree P t → AllList P (tree2list t) :=
begin
  sorry,
```

(continues on next page)

(continued from previous page)

```
end
```

To complete the proof of `tree2list_lem` I needed some additional lemmas about `Sorted` and `Le_list`, but you may find a different path.

8.2.3 Tree sort and permutation

As before for insertion sort we also need to show that tree sort permutes its input. The proof is actually very similar to the one for insertion sort, we just need to adopt the lemma for `ins`

```
lemma ins_inserts : ∀ n : ℕ,∀ t : Tree,
              Insert n (tree2list t) (tree2list (ins n t)) :=
begin
  sorry
end

theorem sort_perm : ∀ ns : list ℕ , Perm ns (sort ns) :=
begin
  assume ns,
  induction ns,
  apply perm_nil,
  apply perm_cons,
  apply ns_ih,
  apply ins_inserts,
end
```

To show `ins_inserts` I needed two lemmas about `Insert` and `++`:

```
lemma insert_appl : ∀ n:ℕ, ∀ ms nms is : list ℕ,
      Insert n ms nms → Insert n (is ++ ms) (is ++ nms) :=
begin
  sorry,
end

lemma insert_appr : ∀ n:ℕ, ∀ ms is nms : list ℕ,
      Insert n ms nms → Insert n (ms ++ is) (nms ++ is) :=
begin
   sorry,
end
```

Both can be shown by induction over lists but the choice of which list to do induction over is crucial.

8.2.4 Relation to quicksort

I have already mentioned that tree sort is basically quick sort. How can this be you ask because quicksort doesn't actually uses any trees. Here is quick sort in Lean:

```
def split : ℕ → list ℕ → list ℕ × list ℕ
| n [] := ([] , [])
| n (m :: ns) := match split n ns with (l,r) :=
                      if ble n m
                      then (m :: l, r)
```

(continues on next page)

(continued from previous page)

```
                            else (l, m::r) end

def qsort : list ℕ → list ℕ
| [] := []
| (n :: ms) := match split n ms with (l,r) :=
                          (qsort l)++(n::(qsort r)) end


#eval (qsort [6,3,8,2,3])
```

The program uses × and `match` which I haven't explained but I hope it is obvious. Lean isn't happy about the recursive definition of `qsort` because the recursive call isn't on a sublist of the input. This can be fixed by using *well founded recursion* but this is beyond the scope of this course. However, Lean is happy running the program using `#eval`.

We can get form tree sort to quicksort by a process called *program fusion*. In a nutshell: the function `list2tree` produces a tree which is consumed by `tree2list` but we can actually avoid the creation of the intermediate tree by fusing the two together, hence arriving at quick sort.

Can you see how to tree-ify merge sort? Hint: in this case you need to use trees where the leaves contain the data not the nodes.

BIBLIOGRAPHY

[AvMoKo2015] Jeremy Avigad, Leonardo de Moura and Soonho Kong *Theorem proving in Lean* Microsoft Research, 2015, https://leanprover.github.io/theorem_proving_in_lean/

[BaBeBlHo2020] Anne Baanen, Alexander Bentkamp, Jasmin Blanchette and Johannes Hölzl, *The Hitchhiker's Guide to Logical Verification*, 2020, https://leanprover-community.github.io/learn.html