Introduction to Formal Reasoning (G52IFR)

Thorsten Altenkirch

October 3, 2014

Chapter 1

Introduction

1.1 What is this course about?

- The precise art of *formal reasoning.*
- Use a proof assistant (*COQ*) to formalize proofs.
- *Propositional logic* as the scaffolding of reasoning
- Foundational issues: *classical* vs *intuitionistic* logic
- Express yourself precisely using the language of *predicate logic*
- Finite sets and operations on sets, reasoning by cases.
- Reasoning about natural numbers, proof by *induction*
- Equational reasoning (Algebra).
- Reasoning about *programs and data structures.*

1.2 What is Coq ?

- COQ: a Proof Assistant based on the Calculus of Inductive Constructions}
- Developed in France since 1989.
- Growing user community.
- Big proof developments:
  - Correctness of a C-compiler
  - 4 colour theorem

1.3 Why using a proof assistant?

- Avoid holes in paper proofs.
- Do Maths on a computer and properly
- Aid understanding. What is a proof?
- Formal certification of software and hardware.

1.4 Using COQ

- Download COQ from http://coq.inria.fr/
- Runs under MacOS, Windows, Linux
- coqtop : command line interface
- coqide : graphical user interface
- proof general : emacs interface
- coqtop and coqide installed on the lab machines

1.5 For reference

- Coq Reference manual: http://coq.inria.fr/V8.1pl3/refman/
- Coq Library doc: http://coq.inria.fr/library-eng.html
- Coq'Art, the book by Yves Bertot and Pierre Casteran (2004). (available in the library!)
- Certified Programming with Dependent Types, by Adam Chlipala (available online)

1.6 Course organisation

- Course page on Moodle
- Coq Labs: every Thursday (1100 - 1300) in A32, starting next week (10/10).
- Weekly coursework, 1st available next Monday (7/10), 25/100
- Use moodle for coursework submissions.
- Tutorials: start next week. See assignment on moodle.
- Online class test on coq in December, 25/100
- Written exam in January, 50/100
- Discussion Forum: Information about coursework, discuss questions Please use it!

Chapter 2

Propositional Logic

`Section` *prop*.

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

We will later introduce ways to construct interesting propositions, but in the moment we will use propositional variables instead. We declare in Coq:

`Variables` P Q R : `Prop`.

This means that the P,Q,R are atomic propositions 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 are going to introduce a number of connectives and logical constants to construct propositions:

- Implication $\rightarrow$, read $P \rightarrow Q$ as if P then Q.
- Conjunction $\wedge$, read $P \wedge Q$ as P and Q.
- Disjunction $\vee$, read $P \vee Q$ as P or Q.
- *False*, read *False* as "Pigs can fly".
- *True*, read *True* as "It sometimes rains in England."
- Negation $\neg$, read $\neg$ P as not P. We define $\neg$ P as $P \rightarrow$ *False*.
- Equivalence, $\leftrightarrow$, read $P \leftrightarrow Q$ as P is equivalent to Q. We define $P \leftrightarrow Q$ as $(P \rightarrow Q)$ $\wedge$ $(Q \rightarrow P)$.

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

- Implication is right associative, i.e. we read $P \rightarrow Q \rightarrow R$ as $P \rightarrow (Q \rightarrow R)$.

- Implication and equivalence bind weaker than conjunction and disjunction. E.g. we read $P \lor Q \to R$ as $(P \lor Q) \to R$.
- Conjunction binds stronger than disjunction. E.g. we read $P \land Q \lor R$ as $(P \land Q) \lor R$.
- Negation binds stronger than all the other connectives, e.g. we read $\neg\ P \land Q$ as $(\tilde{}\ P) \land Q$.

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

We will now discuss how to prove propositions in Coq. 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.

2.1 Our first proof

We start with a very simple tautology $P \to P$, i.e. if P then P. To start a proof we write:

`Lemma` $I : P \to P$.

It is useful to run the source of this document in Coq to see what happens. Coq enters a proof state and shows what we are going to prove under what assumptions. In the moment our assumptions are that P,Q,R are propositions and our goal is $P \to P$. To prove an implication we add the left hand side to the assumptions and continue to prove the right hand side - this is done using the `intro` tactic. We also choose a name for the assumption, let's call it p.

`intro` p.

This changes the proof state: we now have to prove P but we also have a new assumption $p : P$. We can finish the proof by using this assumption. In Coq this can done by using the `exact` tactic.

`exact` p.

This finishes the proof. We only have to instruct Coq to save the proof under the name we have indicated in the beginning, in this case I.

`Qed`.

Qed stands for "Quod erat demonstrandum". This is Latin for "What was to be shown."

2.2 Using assumptions.

Next we will prove another tautology, namely $(P \to Q) \to (Q \to R) \to P \to R$. Try to understand why this is intuitively true for any propositions P,Q and R.

To prove this in Coq we need to know how to use an implication which we have assumed. This can be done using the `apply` tactic: if we have assumed $P \to Q$ and we want to prove Q then we can use the assumption to reduce (hopefully) the problem to proving P. Clearly, using this step is only sensible if P is actually easier to prove than Q. Step through the next proof to see how this works in practice!

`Lemma` $C : (P \to Q) \to (Q \to R) \to P \to R.$

We have to prove an implication, hence we will be using `intro`. Because $\to$ is right associative the proposition can be written as $(P \to Q) \to ((Q \to R) \to P \to R)$. Hence we are going to assume $P \to Q$.

`intro` pq.

we continue assuming...

`intro` qr.
`intro` p.

Now we have three assumptions $P \to Q$, $Q \to R$ and P. It remains to prove R. We cannot use `intro` any more because our goal is not an implication. Instead we need to use our assumptions. The only assumption which could help us to prove R is $Q \to R$. We use the `apply` tactic.

`apply` qr.

Apply uses $Q \to R$ to reduce the problem to prove R to the problem to prove Q. Which in turn can be further reduced to proving P using $P \to Q$.

`apply` pq.

And now it only remains to prove P which is one of our assumptions - hence we can use `exact` again. `exact` p.
`Qed`.

2.3 Introduction and Elimination

We observe that there are two types of proof steps (tactics):

- introduction: How can we prove a proposition? In the case of an implication this is `intro`. To prove $P \to Q$, we assume P and prove Q.

- elimination: How can we use an assumption? In the case of implication this is `apply`. If we know $P \to Q$ and we want to prove Q it is sufficient to prove P.

Actually `apply` is a bit more general: if we know $P1 \to P2 \to ... \to Pn \to Q$ and we want to prove Q then it is sufficient to prove $P1$,$P2$,...,Pn. Indeed the distinction of introduction and elimination steps is applicable to all the connectives we are going to encounter. This is a fundamental symmetry in reasoning.

There is also a 3rd kind of steps: structural steps. An example is `exact` which we can use when we want to refer to an assumption. We can also use `assumption` then we don't even have to give the name of the assumption.

If we want to combine several `intro` steps we can use `intros`. We can also use `intros` without parameters in which case Coq does as many `intro` as possible and invents the names itself.

2.4 Conjunction

How to prove a conjunction? To prove $P \land Q$ we need to prove P and Q. This is achieved using the `split` tactic. We look at a simple example.

`Lemma` *pair* : $P \to Q \to P \land Q$.

On the top level we have to prove an implication.

`intros` *p q*.

now to prove $P \land Q$ we use `split`.

`split`.

This creates two subgoals. We do the first

`exact` *p*.

And then the 2nd

`exact` *q*.
`Qed`.

How do we use an assumption $P \land Q$. We use `destruct` to split it into two assumptions. As an example we prove that $P \land Q \to Q \land P$.

`Lemma` *andCom* : $P \land Q \to Q \land P$.
`intro` *pq*.
`destruct` *pq* `as` [*p q*].
`split`.

Now we need to use the assumption $P \land Q$. We destruct it into two assumptions: P and Q. `destruct` allows us to name the new assumptions.

`exact` *q*.
`exact` *p*.
`Qed`.

Can you see a shorter proof of the same theorem ?

To summarize for conjunction we have:

- introduction: `split`: to prove $P \land Q$ we prove both P and Q.

- elimination: `destruct`: to prove something from $P \wedge Q$ we prove it from assuming both P and Q.

2.5 The currying theorem

Maybe you have already noticed that a statement like $P \to Q \to R$ basically means that R can be proved from assuming both P and Q. Indeed, it is equivalent to $P \wedge Q \to R$. We can show this formally by using $\leftrightarrow$ for the first time.

All the steps we have already explained so I won't comment. It is a good idea to step through the proof using Coq.

```
Lemma curry : (P ∧ Q → R) ↔ (P → Q → R).
unfold iff.
split.
intros H p q.
apply H.
split.
exact p.
exact q.
intros pqr pq.
apply pqr.
destruct pq as [p q].
exact p.
destruct pq as [p q].
exact q.
Qed.
```

I call this the currying theorem, 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. So in Haskell addition has the type $Int \to Int \to Int$.

2.6 Disjunction

To prove a disjunction like $P \vee Q$ we can either prove P or Q. This is done via the tactics *left* and *right*. As an example we prove $P \to P \vee Q$.

```
Lemma inl : P → P ∨ Q.
intros p.
```

Clearly, here we have to use *left*.

```
left.
exact p.
```

```
Qed.
```

To use a disjunction $P \lor Q$ to prove something we have to prove it from both P and Q. The tactic we use is also called `destruct` but in this case `destruct` creates two subgoals. This can be compared to case analysis in functional programming. Indeed we can prove the following theorem.

```
Lemma case : P ∨ Q → (P → R) → (Q → R) → R.
intros pq pr qr.
destruct pq as [p | q].
```

The syntax for `destruct` for disjunction is different if we want to name the assumption we have to separate them with $|$. Indeed each of them will be visible in a different part of the proof. First we assume P.

```
apply pr.
exact p.
```

And then we assume Q

```
apply qr.
exact q.
Qed.
```

So again to summarize: For disjunction we have:

- introduction: there are two ways to prove a disjunction $P \lor Q$. We use *left* to prove it from P and *right* to prove it from Q.
- elimination: If we have assumed $P \lor Q$ then we can use `destruct` to prove our current goal from assuming P and from assuming Q.

2.7 Distributivity

As an example of how to combine the proof steps for conjunction and disjunction we show that distributivity holds, i.e. $P \land (Q \setminus\!/ R)$ is logically equivalent to $(P \land Q) \lor (P \land R)$. This is reminiscent of the principle in algebra that $x \times (y + z) = x \times y + x \times z$.

```
Lemma andOrDistr : P ∧ (Q ∨ R)
            ↔ (P ∧ Q) ∨ (P ∧ R).
split.
intro pqr.
destruct pqr as [p qr].
destruct qr as [q | r].
left.
split.
exact p.
```

```
exact q.
right.
split.
exact p.
exact r.
intro pqpr.
destruct pqpr as [pq | pr].
split.
destruct pq as [p q].
exact p.
left.
destruct pq as [p q].
exact q.
destruct pr as [p r].
split.
exact p.
right.
exact r.
Qed.
```

As before: to understand the working of this script it is advisable to step through it using Coq.

2.8 True and False

True is just a conjunction with no arguments as opposed to $\wedge$ which has two. Similarity *False* is a disjunction with no arguments. As a consequence we already know the proof rules for *True* and *False*.

We can prove *True* without any assumptions.

```
Lemma triv : True.
split.
```

Here we split but instead of two subgoals we get none.

```
Qed.
```

On the other had we can prove anything from *False*. This is called "ex falso quod libet" in Latin.

```
Lemma exFalso : False → P.
intro f.
destruct f.
```

Here instead of two subgoals we get none.

```
Qed.
```

In terms of introduction and elimination steps we may summarize:

- True: There is one introduction rule but no elimination.
- False: There is one elimination rule but no introduction.

2.9 Negation

$\neg P$ is defined as $P \to$ *False*. Using this we can establish some basic theorems about negation. First we show that we cannot have both P and $\neg P$, that is we prove $\neg (P \land \neg P)$.

```
Lemma incons : ¬ ( P ∧ ¬ P).
unfold not.
intro h.
destruct h as [p np].
apply np.
exact p.
Qed.
```

Another example is to show that P implies $\neg \neg P$.

```
Lemma p2nnp : P → ¬ ¬ P.
unfold not.
intros p np.
apply np.
exact p.
Qed.
```

2.10 Classical Reasoning

You may expect that we can also prove the other direction $\neg \neg P \to P$ and that indeed $P \leftrightarrow \neg \neg P$. We can reason that P is either *True* or *False* and in both cases $\neg \neg P$ will be the same. However, this reasoning is not possible using the principles we have introduced so far. The reason is that Coq is based on intuitionistic logic, and the above proposition is not provable intuitionistically.

However, we can use an additional axiom, which corresponds to the principle that every proposition is either *True* or *False*, this is the Principle of the Excluded Middle $P \lor \neg P$. In Coq this can be achieved by:

```
Require Import Coq.Logic.Classical.
```

This means we are now using Classical Logic instead of Intuitionistic Logic. The only difference is that we have an axiom *classic* which proves the principle of the excluded middle for any proposition. We can use this to prove $\neg \neg P \to P$.

`Lemma` *nnpp* : $\tilde{}\tilde{}P \to P$.
`intro` *nnp*.

Here we use a particular instance of *classic* for P.

`destruct` (*classic* P) `as` [p | np].

First case P holds

`exact` p.

2nd case $\neg$ P holds. Here we appeal to *exFalso*.

`apply` *exFalso*.

Notice that we have shown *exFalso* only for P. We should have shown it for any proposition but this would involve quantification over all propositions and we haven't done this yet.

`apply` *nnp*.
`exact` *np*.
`Qed`.

Unless stated otherwise we will try to prove propositions intuitionsitically, that is without using *classic*. An intuitionistic proof provides a positive reason why something is true, while a classical proof may be quite indirect and not so easily acceptable intuitively. Another advantage of intuitionistic reasoning is that it is constructive, that is whenever we prove the existence of a certain object we can also explicitly construct it. This is not true in classical logic. Moreover, in intuitionistic logic we can make differences which disappear when using classical logic. For example we can explicit state when a property is decidable, i.e. can be computed by a computer program.

2.11 The cut rule

This is a good point to introduce another structural rule: the cut rule. Cutting a proof means to introduce an intermediate goal, then you prove your current goal from this intermediate goal, and you prove the intermediate goal. This is particularly useful when you use the intermediate goal several times.

In Coq this can be achieved by using `assert`. `assert` h : H introduces H as a new subgoal and after you have proven this you can use an assumption h : H to prove your original goal.

The following (artificial) example demonstrates the use of `assert`.

`Lemma` *usecut* : $(P \wedge \neg P) \to Q$.
`intro` *pnp*.

If we had a generic version of *exFalso* we could use this. Instead we can introduce *False* as an intermediate goal. `assert` (f : *False*).

which is easy to prove `destruct` *pnp* `as` [p np].
`apply` *np*.

`exact` p.

and using *False* it is easy to prove Q. `destruct` f.

`Qed`.

This example also shows that sometimes we have to cut (i.e. use `assert`) to prove something.

Chapter 3

Predicate Logic

`Section` *pred*.

Predicate logic extends propositional logic: we can talk about sets of things, e.g. numbers and define properties, called predicates and relations. We will soon define some useful sets and ways to define sets but for the moment, we will use set variables as we have used propositional variables before.

In Coq we can declare set variables the same way as we have declared propositional variables:

`Variables` A B : `Set`.

Thus we have declared A and B to be variables for sets. For example think of A=the set of students and B= the set of modules. That is any tautology using set variable remains true if we substitute the set variables with any conrete set (e.g. natural numbers or booleans, etc).

Next we also assume some predicate variables, we let P and Q be properties of A (e.g. P x may mean x is clever and Q x means x is funny).

`Variables` P Q : $A \to$ `Prop`.

Coq views these predicates as functions from A to `Prop`. That is if we have an element of A, e.g. $a : A$, we can apply P to a by writing $P\ a$ to express that a has the property P.

We can also have properties relating several elements, possibly of different sets, these are usually called *relations*. We introduce a relation R, relating A and B by:

`Variable` R : $A \to B \to$ `Prop`.

E.g. R could be the relation "attends" and we would write "R jim g52ifr" to express that Jim attends g52ifr.

3.1 Universal quantification

To say all elements of A have the property P, we write $\forall\ x{:}A,\ P\ x$ more general we can form $\forall\ x{:}A,\ PP$ where PP is a proposition possibly containing the variable x. Another example is $\forall\, x{:}A, P\ x \to Q\ x$ meaning that any element of A that has the property P will also have the property Q. In our example that would mean that any clever student is also funny.

As an example we show that if all elements of A have the property P and that if whenever an element of A has the property P has also the property Q then all alements of A have the property Q. That is if all students are clever, and every clever student is funny, then all students are funny. In predicate logic we write $\forall(\, x{:}A, P\ x) \to \forall(\, x{:}A, P\ x \to Q\ x) \to \forall\, x{:}A,\ Q\ x$.

We introduce some new syntactic conventions: the scope of an forall always goes as far as possible. That is we read $\forall\, x{:}A, P\ x \wedge Q$ as $\forall\, x{:}A,\ (P\ x \wedge Q)$. Given this could we have saved any parentheses in the example above without changing the meaning?

As before we use introduction and elimination steps. Maybe surprisingly the tactics for implication and universal quantification are the same. The reason is that in Coq's internal language implication and universal quantification are actually the same.

`Lemma` *AllMono* : $(\forall\, x{:}A, P\ x) \to (\forall\, x{:}A, P\ x \to Q\ x) \to \forall\, x{:}A,\ Q\ x$.
`intros` *H1 H2*.

To prove $\forall\, x{:}A, Q\ x$ assume that there is an element $a{:}A$ and prove $Q\ a$ We use `intro` a to do this.

`intro` a.

If we know $H2 : \forall\, x{:}A, P\ x \to Q\ x$ and we want to prove $Q\ a$ we can use `apply` *H2* to instantiate the assumption to $P\ a \to Q\ a$ and at the same time eliminate the implication so that it is left to prove $P\ a$.

`apply` *H2*.

Now if we know $H1 : \forall\, x{:}A, P\ x$ and we want to show $P\ a$, we use `apply` *H1* to prove it. After this the goal is completed.

`apply` *H1*.

In the last step we only instantiated the universal quantifier.

`Qed`.

So to summarize:

- introduction for $\forall$: To show $\forall\, x{:}A, P\ x$ we say `intro` a which introduces an assumption $a{:}A$ and it is left to show P where each free occurence of x is replaced by a.

- elimination for $\forall$: We only describe the simplest case: If we know $H : \forall\, x{:}A, P$ and we want to show P where x is replaced by a we use `apply` H to prove $P\ a$.

When I say that each free occurence of x in the proposition P is replaced by a, I mean that occurences of x which are in the scope of another quantifier (these are called bound) are not affected. E.g. if P is $Q\ x \wedge \forall\, x{:}A, R\ x\ x$ then the only free occurence of x is the one in $Q\ x$. That is we obtain $Q\ a \wedge \forall\, x{:}A, R\ x\ x$. The occurences of x in $\forall\, x{:}A, R\ x\ x$ are bound.

We can also use `intros` here. That is if the current goal is $\forall\, x{:}A, P\ x \to Q\ x$ then `intros` $x\ P$ will introduce the assumptions $x{:}A$ and $H{:}P\ x$.

The general case for `apply` is a bit hard to describe. Basically apply may introduce several subgoals if the assumption has a prefix of $\forall$ and $\rightarrow$. E.g. if we have assumed $H : \forall\ x{:}A \forall,\ y{:}B, P\ x \rightarrow Q\ y \rightarrow R\ x\ y$ and our current goal is $R\ a\ b$ then `apply` H will instantiate x with a and y with b and generate the new goals $Q\ b$ and $R\ a\ b$.

Next we are going to show that $\forall$ *commutes with* $\wedge$. That is we are going to show $\forall(\ x{:}A, P\ x \wedge Q\ x) \leftrightarrow \forall(\ x{:}A,\ P\ x) \wedge \forall(\ x{:}A,\ Q\ x)$ that is "all students are clever and funny" is equivalent to "all students are clever" and "all students are funny".

```
Lemma AllAndCom : (∀ x:A,P x ∧ Q x) ↔ (∀ x:A, P x) ∧ (∀ x:A, Q x).
split.
```

Proving $\rightarrow$

```
intro H.
split.
intro a.
assert (pq : P a ∧ Q a).
apply H.
destruct pq as [p q].
exact p.
intro a.
assert (pq : P a ∧ Q a).
apply H.
destruct pq as [p q].
exact q.
```

Proving $\leftarrow$

```
intro H.
destruct H as [p q].
intro a.
split.
apply p.
apply q.
Qed.
```

This proof is quite lengthy and I even had to use `assert`. There is a shorter proof, if we use *edestruct* instead of `destruct`. The "e" version of tactics introduce metavariables (visible as ?x) which are instantiated when we are using them. See the Coq reference manual for details.

I only do the $\rightarrow$ direction using *edestruct*, the other one stays the same.

```
Lemma AllAndComE : (∀ x:A,P x ∧ Q x) → (∀ x:A, P x) ∧ (∀ x:A, Q x).
```

Proving $\rightarrow$

```
intro H.
split.
intro a.
```

edestruct H `as` $[p\ q]$.
`apply` p.
`intro` a.
edestruct H `as` $[p\ q]$.
`apply` q.
`Qed`.

Question: Does $\forall$ also commute with $\vee$? That is does $\forall(\ x{:}A, P\ x \vee Q\ x) \leftrightarrow \forall(\ x{:}A,\ P\ x) \vee \forall(\ x{:}A,\ Q\ x)$ hold? If not, how can you show that?

3.2 Existential quantification

To say that there is an element of A having the property P, we write $\exists\ x{:}A,\ P\ x$ more general we can form $\exists\ x{:}A,\ PP$ where PP is a proposition possibly containing the variable x. Another example is $\exists\ x{:}A, P\ x \wedge Q\ x$ meaning that there is an element of A that has the property P and the property Q. In our example that would mean that there is a student who is both clever and funny.

As an example we show that if there is an element of A having the property P and that if whenever an element of A has the property P has also the property Q then there is an elements of A having the property Q. That is if there is a clever student, and every clever student is funny, then there is a funny student. In predicate logic we write $(\exists\ x{:}A, P\ x) \to \forall(\ x{:}A, P\ x \to Q\ x) \to \exists\ x{:}A,\ Q\ x$.

Btw, we are not changing the 2nd quantifier, it stays $\forall$. What would happen if we would replace it by $\exists$?

The syntactic conventions for $\exists$ are the same as for $\forall$: the scope of an $\exists$ always goes as far as possible. That is we read $\exists\ x{:}A, P\ x \wedge Q$ as $\exists\ x{:}A,\ (P\ x \wedge Q)$.

The tactics for existential quatification are similar to the ones for conjunction. To prove an existential statement $\exists\ x{:}A, PP$ we use $\exists\ a$ where $a : A$ is our *witness*. We then have to prove PP where each free occurence of x is replaced by a. To use an assumption $H : \exists\ x{:}A, PP$ we employ `destruct` H `as` $[a\ p]$ which destructs H into $a : A$ and $p : PP'$ where PP' is PP where all free occurences of x have been replaced by a.

`Lemma` *ExistsMono* : $(\exists\ x{:}A, P\ x) \to (\forall\ x{:}A, P\ x \to Q\ x) \to \exists\ x{:}A,\ Q\ x$.
`intros` *H1 H2*.

We first eliminate or assumption.

`destruct` *H1* `as` $[a\ p]$.

And now we introduce the existential.

$\exists\ a$.
`apply` *H2*.

In the last step we instantiated a universal quantifier.

`exact` p.

`Qed.`

So to summarize:

- introduction for $\exists$ To show $\exists\ x{:}A{,}P$ we say $\exists\ a$ where $a : A$ is any expression of type a. It remains to show P where any free occurence of x is replaced by a.
- elimination for $\exists$ If we know $H : \exists\ x{:}A{,}P$ we can use `destruct` H `as` $[a\ p]$ which destructs H intwo two assumptions: $a : A$ and $p : P'$ where P' is obtained from P by replacing all free occurences of x in P by a.

Next we are going to show that $\exists$ *commutes with* $\vee$. That is we are going to show $(\exists\ x{:}A{,}P\ x \vee Q\ x) \leftrightarrow (\exists\ x{:}A,\ P\ x) \vee (exits\ x{:}A,\ Q\ x)$ that is "there is a student who is clever or funny" is equivalent to "there is a clever student or there is a funny student".

`Lemma` *ExOrCom* : $(\exists\ x{:}A{,}P\ x \vee Q\ x) \leftrightarrow (\exists\ x{:}A,\ P\ x) \vee (\exists\ x{:}A,\ Q\ x)$.
`split`.

Proving $\rightarrow$

`intro` *H*.

It would be too early to use the introduction rules now. We first need to analyze the assumptions. This is a common situation.

`destruct` *H* `as` [*a pq*].
`destruct` *pq* `as` [*p* | *q*].

First case $P\ a$.

left.
$\exists\ a$.
`exact` *p*.

Second case $Q\ a$.

right.
$\exists\ a$.
`exact` *q*.

Proving $\leftarrow$

`intro` *H*.
`destruct` *H* `as` [*p* | *q*].

First case $\exists\ x{:}A{,}P\ x$

`destruct` *p* `as` [*a p*].
$\exists\ a$.
left.
`exact` *p*.

Second case $\exists\ x{:}A{,}Q\ x$

```
destruct q as [a q].
∃ a.
right.
exact q.
Qed.
```

3.3 Another Currying Theorem

There is also a currying theorem in predicate logic which exploits the relation between $\to$ and $\forall$ on the one hand and $\wedge$ and exists on the other. That is we can show that $\forall\ x{:}A{,}P\ x \to S$ is equivalent to $(\exists\ x{:}A{,}P\ x) \to S$. Intuitively, think of S to be "the lecturer is happy". Then the left hand side can be translated as "If there is any student who is clever, then the lecturer is happy" and the right hand side as "If there exists a student who is clever, then the lecturer is happy". The relation to the propositional currying theorem can be seen, when we replace $\forall$ by $\to$ and $\exists$ by $\wedge$.

To prove this tautology we assume an additional proposition.

```
Variable S : Prop.
```

```
Lemma Curry : (∀ x:A,P x → S) ↔ ((∃ x:A,P x) → S).
split.
```

proving $\to$

```
intro H.
intro p.
destruct p as [a p].
```

With our limited knowledge of Coq's tactic language we need to instantiate H using `assert`. There are better ways to do this... We will see later.

```
assert (H' : P a → S).
apply H.
apply H'.
exact p.
```

proving $\leftarrow$.

```
intro H.
intros a p.
apply H.
∃ a.
exact p.
Qed.
```

As before the explicit instantiation using `assert` can be avoided by using the "e" version of a tactic. In this case it is `eapply`. Again, I refer to the Coq reference manual for details. I only do one direction, the other one stays the same.

Lemma $CurryE$: $(\forall\ x{:}A, P\ x \to S) \to ((\exists\ x{:}A, P\ x) \to S)$.

proving $\to$

```
intro H.
intro p.
destruct p as [a p].
eapply H.
apply p.
Qed.
```

3.4 Equality

Predicate logic comes with one generic relation which is defined for all sets: equality (=). Given two expressions $a,b : A$ we write $a = b$: `Prop` for the proposition that a and b are equal, that is they describe the same object.

How can we prove an equality? That is what is the introduction rule for equality? We can prove that every expression is $a : A$ is equal to itself $a = a$ using the tactic `reflexivity`. How can we use an assumption $H : a = b$? That is how can we eliminate equality? If we want to prove a goal P which contains the expression a we can use `rewrite` H to *rewrite* all those as into bs.

To demonstrate how to use these tactics we show that equality is an *equivalence relation* that is, it is:

- reflexive ($\forall\ a{:}A,\ a = a$)
- symmetric ($\forall\ a\ b{:}A,\ a{=}b \to b{=}a$)
- transitive ($\forall\ a\ b\ c{:}A,\ a{=}b \to b{=}c \to a{=}c$.

Lemma eq_refl : $\forall\ a{:}A,\ a = a$.

```
intro a.
```

Here we just invoke the reflexivity tactic.

```
reflexivity.
Qed.
```

Lemma eq_sym : $\forall\ a\ b{:}A,\ a{=}b \to b{=}a$.

```
intros a b H.
```

Here we use rewrite to reduce the goal.

```
rewrite H.
reflexivity.
Qed.
```

Lemma eq_trans : $\forall\ a\ b\ c{:}A,\ a{=}b \to b{=}c \to a{=}c$.

```
intros a b c ab bc.
rewrite ab.
exact bc.
Qed.
```

Do you know any other equivalence relations?

3.5 Classical Predicate Logic

The principle of the excluded middle *classic* $P : P \vee \neg P$ has many important applications in predicate logic. As an example we show that $\exists\, x{:}A, P\ x$ is equivalent to $\neg\, \forall\, x{:}A,\, \neg\ P\ x$.

Instead of using *classic* directly we use the derivable principle $NNPP : \neg\, \neg\ P \rightarrow P$ which is also defined in *Coq.Logic.Classical*.

```
Require Import Coq.Logic.Classical.
```

```
Lemma ex_from_forall : (∃ x:A, P x) ↔ ¬ ∀ x:A, ¬ P x.
split.
```

proving →

```
intro ex.
intro H.
destruct ex as [a p].
assert (npa : ¬ (P a)).
apply H.
apply npa.
exact p.
```

proving ←

```
intro H.
apply NNPP.
```

Instead of proving $\exists\, x{:}A, P\ x$ which is hard, we show ~~ $\exists\, x{:}A, P\ x$ which is easier. `intro` *nex*.

```
apply H.
intros a p.
apply nex.
∃ a.
exact p.
Qed.
```

Chapter 4

Bool

```
Section Bool.
```

4.1 Defining bool and operations

We define *bool* : `Set` as a finite set with two elements: *true* : *bool* and *false* : *bool*. In set theoretic notation we would write *bool* = { *true* , *false* }.

In Coq we write:

```
Inductive bool : Set :=
  | true : bool
  | false: bool.
```

However, we don't need to define *bool* here because it is already defined in the Coq prelude.

The function *negb* : *bool* → *bool* (boolean negation) can be defined by pattern matching using the `match` construct.

```
Definition negb (b:bool) : bool :=
  match b with
  | true ⇒ false
  | false ⇒ true
  end.
```

This should be familiar from g51fun - in Haskell `match` is called `case`. Indeed Haskell offers a more convenient syntax for top-level pattern.

We can evaluate the function using the slightly lengthy phrase `Eval compute in` (...):

```
Eval compute in (negb true).
```

The evaluator replaces

negb true
with
`match` *true* `with` | *true* ⇒ *false* | *false* ⇒ *true* `end`.
which in turn evaluates to
false

`Eval compute in` *negb* (*negb true*).

We know already that *negb true* evaluates to *false* hence *negb* (*negb true*) evaluates to *negb false* which in turn evaluates to *true*.

Other boolean functions can be defined just as easily:

`Definition` *andb*(*b c*:*bool*) : *bool* :=
`if` *b* `then` *c* `else` *false*.

`Definition` *orb* (*b c* : *bool*) : *bool* :=
`if` *b* `then` *true* `else` *c*.

The Coq prelude also defines the infix operators && and || for andb and orb respectively, with && having higher precedence than ||. Note however, that you cannot use ! (for negb) since this is used for other purposes in Coq.

4.2 Reasoning about Bool

We can now use predicate logic to show properties of boolean functions. As a first example we show that the function *negb* run twice is the identity:

∀ *b* :*bool*, *negb* (*negb b*) = *b*

To prove this, the only additional thing we have to know is that we can analyze a boolean variable *b* : *bool* using `destruct` *b* which creates a case for *b* = *true* and one for *b* = *false*.

`Lemma` *negb_twice* : ∀ *b* :*bool*, *negb* (*negb b*) = *b*.
`intro` *b*.
`destruct` *b*.

Case for *b* = *true* Our goal is negb (negb true) = true. As we have already seen *negb* (*negb true*) evaluates to true. Hence this goal can be proven using `reflexivity`. Indeed, we can make this visible by using `simpl`.

`simpl`.
`reflexivity`.

Case for *b* = *false* This case is exactly the same as before.

`simpl`.
`reflexivity`.
`Qed`.

There is a shorter way to write this proof by using ; instead of , after `destruct`. This means that `reflexivity` is used for both cases. We can also omit the `simpl` which we only use for cosmetic reasons.

```
Lemma negb_twice' : ∀ b :bool, negb (negb b) = b.
intro b.
destruct b;
  reflexivity.
Qed.
```

Indeed, proving equalities of boolean functions is very straightforward. All we need is to analyze all cases and then use `reflexivity`. For example to prove that *andb* is commutative, i.e.

$\forall\ x\ y\ :\ bool,\ andb\ x\ y = andb\ y\ x$

(we use the abbrevation: $\forall\ x\ y\ :\ A, \ldots$ is the same as $\forall\ x{:}A\forall,\ y{:}A,\ \ldots$. Note that alas the same shorthand doesn't work for $\exists$ (actually it now does in the latest version of Coq).

```
Lemma andb_comm : ∀ x y : bool, andb x y = andb y x.
intros x y.
destruct x;
  (destruct y;
     reflexivity).
Qed.
```

We can also prove other properties of *bool* not directly related to the functions, for example, we know that every boolean is either true or false. That is

$\forall\ b\ :\ bool,\ b = true \lor b = false$

This is easy to prove:

```
Lemma true_or_false : ∀ b : bool,
          b = true ∨ b = false.
intro b.
destruct b.
    b = true  left.
reflexivity.
    b = false  right.
reflexivity.
Qed.
```

Next we want to prove something which doesn't involve any quantifiers, namely

$\neg\ (true = false)$

This is not so easy, we need a little trick. We need to embed *bool* into `Prop`, mapping *true* to *True* and *false* to *False*. This is achieved via the function Istrue:

```
Definition Istrue (b : bool) : Prop :=
  match b with
  | true ⇒ True
  | false ⇒ False
  end.
```

So *IsTrue* maps *true* to *True* and *false* to *False*. What is the difference between the small and capital versions of true and false?

Now we can prove our property: `Lemma` *diff_true_false* :
$\neg$ (*true* = *false*).

`intro` *h*.

We now need to use a new tactic to replace *False* by *IsTrue false*. This is possible because *IsTrue false* evaluates to *False*. We are using *fold* which is the inverse to `unfold` which we have seen earlier.

fold (*Istrue false*).

Now we can simply apply the equation *h* backwards.

`rewrite`$\leftarrow$ *h*.

Now by unfolding we can replace *Istrue true* by *True*

`unfold` *Istrue*.

Which is easy to prove.

`split`.
`Qed`.

Actually there is a tactic `discriminate` which implements this proof and which allows us to prove directly that any two different constructors (like *true* and *false*) are different. We shall use `discriminate` in future.

`Goal` *true* $\neq$ *false*.
`intro` *h*.
`discriminate` *h*.
`Qed`.

4.3 Reflection

We have `Prop` and *bool* which look very similar. However, an important difference is that an proposition P : `Prop` may have a proof or not but we cannot see this easily. In contrast we cannot "prove" a boolean b : *bool* but we can see wether it is *true* or *false* just by looking at it.

We also have similar operations on `Prop` and *bool*, e.g. there is a logical operator $\wedge$ which acts on `Prop` and a boolean operator *andb* (or &&) which acts on *bool*. How are the two related?

We can use $\wedge$ to specify *andb*, namely we say that *andb* x y = *true* is equivalent to x = *true* and y = *true*. That is we prove:

`Lemma` *and_ok* : $\forall$ x y : *bool*,
andb x y = *true* $\leftrightarrow$ x = *true* $\wedge$ y = *true*.
`intros` x y.
`split`.

$\rightarrow$

```
destruct x.
```

x=true

```
intro h.
split.
reflexivity.
simpl in h.
exact h.
```

Why did the last step work?

x = false

```
intro h.
simpl in h.
discriminate h.
```

←

```
intro h.
destruct h as [hx hy].
rewrite hx.
exact hy.
Qed.
```

The two directions of the proof tell us different things:

- ← tells us that *andb* is sound, if both inputs are true if will return true.
- → tells us that *andb* is complete, it will only return *true* if both inputs are *true*.

What would be implementations of *andb* which are sound but not complete, and complete but not sound?

```
End Bool.
```

Chapter 5

How to make sets

```
Section Sets.
```

Some magic incantations...

```
Open Scope type_scope.
Set Implicit Arguments.
Implicit Arguments inl [A B].
Implicit Arguments inr [A B].
```

5.1 Finite Sets

As we have defined *bool* we can define other finite sets just by enumerating the elements.

Im Mathematics (and conventional Set Theory), we just write C = { c1, c2 , .. , cn } for a finite set.

In Coq we write

`Inductive C : Set := | c1 : C | c2 : C ... | cn : C.`

As a special example we define the empty set:

```
Inductive empty_set : Set := .
```

As an example for finite sets, we consider the game of chess. We need to define the colours, the different type of pieces, and the coordinates.

```
Inductive Colour : Set :=
  | white: Colour
  | black : Colour.
```

```
Inductive Rank : Set :=
  | pawn : Rank
  | rook : Rank
  | knight : Rank
  | bishop : Rank
  | queen : Rank
```

```
  | king : Rank.
Inductive XCoord : Set :=
  | xa : XCoord
  | xb : XCoord
  | xc : XCoord
  | xd : XCoord
  | xe : XCoord
  | xf : XCoord
  | xg : XCoord
  | xh : XCoord.
Inductive YCoord : Set :=
  | y1 : YCoord
  | y2 : YCoord
  | y3 : YCoord
  | y4 : YCoord
  | y5 : YCoord
  | y6 : YCoord
  | y7 : YCoord
  | y8 : YCoord.
```

In practice it is not such a good idea to use different sets for the x and y coordinates. We use this here for illustration and it does reflect the chess notation like e2 - e4 for moving the pawn in front of the king.

We can define operations on finite sets using the `match` construct we have already seen for book. As an example we define the operation $oneUp : YCoord \to YCoord$ which increases the y coordinates by 1. We have to decide what to do when we reach the 8th row. Here we just get stuck.

```
Definition oneUp (y : YCoord) : YCoord :=
  match y with
  | y1 ⇒ y2
  | y2 ⇒ y3
  | y3 ⇒ y4
  | y4 ⇒ y5
  | y5 ⇒ y6
  | y6 ⇒ y7
  | y7 ⇒ y8
  | y8 ⇒ y8
  end.
```

5.2 Products

Given two sets A B : Set we define a new set $A \times B$: Set which is called the *product* of A and B. It is the set of pairs (a,b) where a : A and b : B.

As an example we define the set of chess pieces and coordinates:

Definition $Piece$: Set := $Colour \times Rank$.

Definition $Coord$: Set := $XCoord \times YCoord$.

And for illustration construct some elements:

Definition $blackKnight$: $Piece$:= ($black$, $knight$).

Definition $e2$: $Coord$:= (xe , $y2$).

On Products we have some generic operations called *projections* which extract the components of a product.

```
Definition fst(A B : Set)(p : A × B) : A :=
  match p with
  | (a , b) ⇒ a
  end.
```

```
Definition snd(A B : Set)(p : A × B) : B :=
  match p with
  | (a , b) ⇒ b
  end.
```

```
Eval compute in fst blackKnight.
Eval compute in snd blackKnight.
```

```
Eval compute in (fst blackKnight,snd blackKnight).
```

A general theorem about products is that if we take apart an element using projections and then put it back together again we get the same element. In predicate logic this is:

$\forall\ p : A \times B, (fst\ p\ ,\ snd\ p) = p$

This is called *surjective pairing*. In the actual statement in Coq we also have to quantify over the sets involved (which technically gets us into the realm of higher order logic - but we shall ignore this).

```
Lemma surjective_pairing : ∀ A B : Set,
  ∀ p : prod A B, (fst p , snd p) = p.
intros A B p.
```

The actual proof is rather easy. All that we need to know is that we can take apart a product the same way as we have taken apart conjunctions. `destruct p as [a b].`
`simpl.`

Can you simplify this goal in your head? Yes simpl will do the job but why? `reflexivity.`
`Qed.`

Question: If |A| and |B| are finite sets with |m| and |n| elements respectively, how many elements are in |A * B|?

5.3 Disjoint union

Given two sets $A\ B$: `Set` we define a new set $A + B$: `Set` which is called the *disjoint union* of A and B. Elements of $A + B$ are either *inl* a where $a : A$ or *inr* b where $b : B$. Here *inl* stands for "inject left" and *inr* stands for "inject right".

It is important not to confuse + with the union of sets. The disjoint union of *bool* with *bool* has 4 elements because *inl true* is different from *inr true* while in the union of bool with bool there are only 2 elements since there is only one copy of *true*. Actually, the union of sets does not exist in Coq.

As an example we use disjoint union to define the set field which can either be a piece or empty. The second case is represented by a set with just one element called *Empty* which has just one element *empty*.

```
Inductive Empty : Set :=
  | empty : Empty.
```

```
Definition Field : Set := Piece + Empty.
```

some examples

```
Definition blackKnightHere : Field := inl blackKnight.
```

```
Definition emptyField : Field := inr empty.
```

As an example of a defined operation we define *swap* which maps elements of $A + B$ to $B + A$ by mapping *inl* to *inr* and vice versa.

```
Definition swap(A B : Set)(x : A + B) : B + A :=
  match x with
  | inl a ⇒ inr a
  | inr b ⇒ inl b
  end.
```

The same question as for products: If A has m elements and B has n elements, how many elements are in $A + B$?

Disjoint unions are sometimes called *coproducts* because there are in a sense the mirror image of products. To make this precise we need the language of category theory, which is beyond this course. However, if you are curious look up Category Theory on wikipedia.

5.4 Function sets

Given two sets $A\ B$: `Set` we define a new set $A \to B$: `Set`, the set of functions form A to B. We have already seen one way to define functions, whenever we have defined an operation we have actually defined a function. However, as you have already seen in Haskell, we can define functions directly using lambda abstraction. The syntax is `fun` $x \Rightarrow b$ where b is an expression in B which may refer to $x : A$.

In the case of our chess example we can use functions to define a chess board as a function form *Coord* to *Field*, this function would give us the content of a field for any coordinate.

```
Definition Board : Set := Coord → Field.
```

A particular simple example is the empty board:

```
Definition EmptyBoard : Board := fun x ⇒ emptyField.
```

I leave it as an exercise to construct the initial board for a chess game.

As another example instead of defining *negb* as an operation we could also have used `fun`:

```
Definition negb' : bool → bool
  := fun (b : bool) ⇒ match b with
                        | true ⇒ false
                        | false ⇒ true
                        end.
```

Using `fun` is especially useful when we are dealing with *higher order functions*, i.e. function which take functions as arguments. As an example let us define the function *isConst* which determines wether a given function $f : bool \to bool$ is constant.

```
Open Scope bool_scope.

Definition isConst (f : bool → bool) : bool :=
  (f true) && (f false) || negb (f true) && negb (f false).
```

What will Coq answer when asked to evaluate the terms below. In three cases we are using `fun` to construct the argument. Could we have done this in the 1st case as well?

```
Eval compute in isConst negb.
Eval compute in isConst (fun x ⇒ false).
Eval compute in isConst (fun x ⇒ true).
Eval compute in isConst (fun x ⇒ x).
```

Are there any other cases to consider ?

In general, if A,B are finite sets with m and n elements, how many elements are in $A \to B$? Actually we need to assume the axiom of extensionality to get the right answer. This axiom states that any two functions which are equal for all arguments are equal.

```
Axiom ext : ∀ (A B : Set)
      (f g : A → B),
      (∀ x:A,f x = g x) → f = g.
```

As an example we show that the 2 possible definitions of andb are extensionally equal.

```
Definition andb (a b : bool) : bool :=
  if a then b else false.

Definition andb' (a b : bool) : bool :=
  if b then a else false.
```

Lemma *andbEq* : *andb* = *andb'*.

To show equality of functions we use *ext*. apply *ext*.

intro *a*.

andb a is still a function. More *ext* is needed. apply *ext*.

intro *b*.

Now all is left is reasoning with *bool* destruct *a*; (destruct *b*; reflexivity).

Qed.

5.5 The Curry Howard Correspondence

There is a close correspondence between sets and propositions. We may translate a proposition by the set of its proofs. The question wether a proposition holds corresponds then to finding an element which lives in the corresponding set. Indeed, this is what Coq's proof objects are based upon. For propositional logic the translation works as follows:

- conjunction ($\wedge$) is translated as product ($\times$),
- disjunction ($\vee$) is translated as disjoint union ($+$),
- implication ($\rightarrow$) is translated as function set ($\rightarrow$).

I leave it to you to figure out what to translate *True* and *False* with. As an example we consider the currying theorem for propositional logic. Applying the translation we obtain:

```
Definition curry (A B C : Set) :
    ((A × B → C) → (A → B → C)) :=
  fun f ⇒ fun a ⇒ fun b ⇒ f (a , b).
Definition curry' (A B C : Set) :
    (A → B → C) → (A × B → C) :=
  fun g ⇒ fun p ⇒ g (fst p) (snd p).
```

Indeed, *curry* and *curry'* do not just witness a logical equivalence but they constitute an *isomorphism*. That is if we go back and forth we end up with the element we started. We will need the axiom of extensionality. To make this precise we get:

```
Lemma curryIso1 :
  ∀ A B C : Set,
  ∀ f : A × B → C,
  f = curry' (curry f).
intros A B C f.
```

Here we need to prove that two functions are equal. This is the the time to apply ext.

apply *ext*.

intro *p*.

To make the left hand side reduce we need to replace *p* by an actual pair. destruct *p*.

Let's see what happens if we first unfold *curry'*. unfold *curry'*.

and then *curry* `unfold` *curry*.

ok we just need to compute *fst* and *snd* `simpl`.

`reflexivity`.

`Qed`.

`Lemma` *curryIso2* : $\forall$ A B C : `Set`, $\forall$ g : $A \to B \to C$,
g = *curry* (*curry'* g).

`intros` A B C g.

again we need to show an equality of functions - we need to use *ext*. `apply` *ext*.

`intro` a.

We are not done yet with functions, since g a is still a function. `apply` *ext*.

`intro` b.

Let's first unfold *curry*. `unfold` *curry*.

and then *curry'*. `unfold` *curry'*.

again the rest is just computing with *fst* and *snd*. `simpl`.

`reflexivity`.

`Qed`.

`End` *Sets*.

Chapter 6

Peano Arithmetic

`Section` *Arith.*

6.1 The natural numbers

Guiseppe Peano defined the natural numbers as given by $0 : nat$ and if n is a natural number then $S\ n : nat$ is a natural number called the successor of n. Given this we can construct all the natural numbers, e.g.

- $1 = \mathrm{S}\ 0$
- $2 = \mathrm{S}\ 1 = \mathrm{S}\ (\mathrm{S}\ 0)$
- $3 = \mathrm{S}\ 2 = \mathrm{S}\ (\mathrm{S}\ (\mathrm{S}\ 0))$

Moreover these are all natural numbers (we say they are defined *inductively*). In Coq Peano's natural numbers are defined as follows:

```
Inductive nat : Set :=
| O : nat
| S : nat → nat.
```

However, Coq will automatically translate the usual decimal notation of numbers into Peano numbers, i.e. 3 is translated into $S\ (S\ (S\ 0))$.

Peano went on to represent the fundamental properties of the natural numbers using axioms. Some of the axioms express general properties of equality, which we have already seen. But the following three are specific to the natural numbers. Indeed, they are provable propositions in Coq:

- Axiom 7 : 0 is not the successor of any number. $\forall\ n{:}nat,\ S\ n \neq 0$

- Axiom 8 : If two numbers have the same successor, then they are equal. $\forall\ m\ n{:}nat,\ S\ m = S\ n \rightarrow m = n$
- Axiom 9 : If any property holds for 0, and is closed under successor, then it holds for all natural numbers (principle of induction).

$$\begin{aligned}\forall\ P : nat \rightarrow \texttt{Prop},\ P\ 0 \\ &\rightarrow (\forall\ m : nat,\ P\ m \rightarrow P\ (S\ m)) \\ &\rightarrow \forall\ n : nat,\ P\ n\end{aligned}$$

For illustration we are going to prove these principles:

`Lemma` *peano7* : $\forall\ n{:}nat,\ S\ n \neq 0$.
`intro` *n*.
`intro` *h*.

This is basically the same problem as proving *true* $\neq$ *false*, we could apply the same technique here. To avoid repetetion we just use the `discriminate` tactic.

`discriminate` *h*.
`Qed`.

To prove the next axiom, it is useful to define the inverse to S, the predecessor function pred. We arbitrarily decide that the predecessor of 0 is 0.

`Definition` *pred* $(n : nat) : nat :=$
  `match` *n* `with`
  $\mid 0 \Rightarrow 0$
  $\mid S\ n \Rightarrow n$
  `end`.

`Lemma` *peano8* : $\forall\ m\ n{:}nat,\ S\ m = S\ n \rightarrow m = n$.
`intros` *m n h*.

By folding with *pred* we can change the current goal so that we can apply our hypothesis.

fold (*pred* ($S\ m$)).
`rewrite` *h*.

And now we just have to unfold. simpl would have done the job too.

`unfold` *pred*.
`reflexivity`.
`Qed`.

The 8th axiom says that the successor function is injective. Can we prove the other direction too? $\forall\ m\ n{:}nat,\ m = n \rightarrow S\ m = S\ n$ Does this tell us anything new about the successor function?

The proof of the induction axiom is rather boring. It just uses a tactic which is called `induction`...

`Lemma` *peano9* : $\forall\ P : nat \rightarrow$ `Prop`, $P\ 0$

```
                                 → (∀ m : nat, P m → P (S m))
                                 → ∀ n : nat, P n.
intros P h0 hS n.
induction n.
exact h0.
apply hS.
exact IHn.
Qed.
```

6.2 Primitive recursion and induction

To see the induction principle in action we will look at a simple example: we are going to define a doubling function and then show that it always produces even numbers.

To define the doubling function we first need to introduce another principle *primitive recursion*. To define the doubling function recursively we are using the fact that *double* (S n) is S (S (*double* n), e.g. *double* $3 =$ *double* $(S\ 2) = S\ (S\ ($*double* $2)) = S\ (\ S\ ($*double* $4))$ $= 6$.

To define a function recursively we cannot use the keyword `Definition` because this allows only the definition of of non-recursive functions, but we have to use `Fixpoint` instead.

```
Fixpoint double (m : nat) : nat :=
  match m with
  | 0 ⇒ 0
  | S n ⇒ S (S (double n))
  end.

Eval compute in double 3.
```

double is a fixpoint because it solves an equation of the form *double* $=$ f *double*. I leave it to you to figure out what f is in this case.

Not all recursive definitions have fixpoints, e.g. if we had tried to define

```
Fixpoint double (m : nat) : nat :=
  match m with
  | 0 ⇒ 0
  | S n ⇒ S (S (double m))
  end.
```

then Coq would have reported an error (while Haskell would have just looped). Because Coq is for reasoning there is no space for looping function and Coq only allows terminating functions.

In particular primitive recursive functions are this where the computation of f (S n) only uses f n. All primitive recursive functions are accepted by Coq.

Another example of a primitive recursive function is the function *isEven* below which determines wether a number is even.

```
Fixpoint isEven (m : nat) : bool :=
  match m with
  | 0 ⇒ true
  | S m' ⇒ negb (isEven m')
  end.
```

Having both *double* and *isEven* we can now prove that *double* always produces even numbers. To show this we need to use `induction`.

```
Lemma evenDouble : ∀ n : nat, isEven (double n) = true.
intro n.
```

To show something for all numbers we use `induction`. `induction n.`

We get 2 subgoals: we have to show our goal for 0 and for $S\ n$. What we don't see in the moment is that we can use our goal for n when proving it for $S\ n$.

The 0 case is very straightforward. We only need to compute. `simpl.`
`reflexivity.`

We now see that when priving the property for $S\ n$ we can use the property for n. All we need in this simple exqmple is to compute before we can use the *induction hypothesis*.
`simpl.`

Now *isEven* (*double* n) appears in the goal and we know by induction hypothesis that this is *true*. `rewrite IHn.`

Now it is only a simple calculation with booleans. `simpl.`
`reflexivity.`
`Qed.`

6.3 Addition and multiplication

Peano defined the operations addition and multiplication. These are actually examples of functions defined by *primitive recursion*.

The idea is that we can define addition like this:

- to add 0 to a number is just this number,
- to add one more that n to a number is one more than adding n to the number.

```
Fixpoint plus (m n : nat) {struct m} : nat :=
  match m with
  | 0 ⇒ n
  | S m ⇒ S (plus m n)
  end.
```

```
Eval compute in (plus 2 3).
```

In the Coq library addition is defined using the usual infix notation $+$.

To define multiplication we use primitive recursion again. This time the idea is the following.

- multiplying 0 with a number is just 0.
- multiplying one more than n with a number is obtained by adding the number to multiplying n with the number.

```
Fixpoint mult (m n : nat) {struct m} : nat :=
  match m with
  | 0 ⇒ 0
  | S m ⇒ plus n (mult m n)
  end.
```

```
Eval compute in (mult 2 3).
```

In the Coq library addition is defined using the usual infix notation $+$ and $\times$ with the usual rules of precedence. From now on we shall use the library versions which are defined exactly in the same way as we have defined *plus* and *mult*

6.4 Algebraic properties

Addition and multiplication satisfy a number of important equations:

- 0 is a neutral element for addition $0 + m = m$ and $m + 0 = m$
- Addition is associative. $m + (n + l) = (m + l) + n$
- Addition is commutative. $m + n = n + m$
- 1 is a neutral element for multiplication $1 \times m = m$ and $m \times 1 = m$
- Multiplication is associative. $m \times (n \times l) = (m \times n) \times l$
- Multiplication is commutative. $m \times n = n \times m$
- 0 is a null for multiplication. $m \times 0 = 0$ and $0 \times m = 0$
- Addition distributes over multiplication. $m \times (n + l) = m \times n + m \times l$ and $(m + n) \times l = m \times l + n \times l$

In the language of universal algebra, we say that

- $(+,0)$ is a *commutative monoid*, because 0 is neutral, $+$ is associative and commutative.
- (*,1) is a commutative monoid, because 1 is neutral, $\times$ is associative and commutative.

- (+,0,*,1) is a *commutative semiring* because (+,0) and (*,1) are commutative monoids and 0 is a zero for multiplication and addition distributes over multiplication.

We are going to prove that (+,0) is a commutative monoid and leave the remaining properties as an exercise.

`Lemma` *plus_0_n* : $\forall$ *n*:*nat*, $n = 0 + n$.

This property is very easy to prove. Can you see why? `intro` *n*.

`reflexivity`.

`Qed`.

`Lemma` *plus_n_0* : $\forall$ *n*:*nat*, $n = n + 0$.

`intro` *n*.

This one cannot be proven by reflexivity. So we have to use induction.

`induction` *n*.

n = 0 This is easy.

`simpl`.

`reflexivity`.

We can simplify $S\ n + 0$ using the definition of +

`simpl`.

`rewrite`← *IHn*.

`reflexivity`.

`Qed`.

`Lemma` *plus_assoc* : $\forall$ ($l\ m\ n$:*nat*),$l + (m + n) = (l + m) + n$.

`intros` *l m n*.

There seems to be quite a choice what to do induction over: *l*,*m*,*n* but only one of them works. Why?

`induction` *l*.

`simpl`.

`reflexivity`.

`simpl`.

`rewrite` *IHl*.

`reflexivity`.

`Qed`.

To prove commutativity we first prove a lemma we know already that $0 + m = m = m + 0$ but what about $S\ m + n = S\ (m + n) = m + S\ n$?

`Lemma` *plus_n_Sm* : $\forall$ *n m* : *nat*, $S\ (m + n) = m + S\ n$.

`intros`.

`induction` *m*.

`simpl`.

`reflexivity`.

```
simpl.
rewrite IHm.
reflexivity.
Qed.
```

We are now ready to prove commutativity.

```
Lemma plus_comm : ∀ n m:nat, n + m = m + n.
intros.
induction n.
simpl.
apply plus_n_0.
simpl.
rewrite IHn.
apply plus_n_Sm.
Qed.
```

6.5 Ordering the numbers

We define the relation $\leq$ on natural numbers by saying that $m \leq n$ holds if there is a number k such that $m = k + n$.

```
Definition leq (m n : nat) : Prop :=
  ∃ k : nat, n = k + m.
Notation "m <= n" := (leq m n).
```

We verify some basic properties of $\leq$:

- $\leq$ is reflexive. $\forall\ n{:}nat,\ n \leq n$
- $\leq$ is transitive. $\forall\ l\ m\ n{:}nat,\ l \leq m \to m \leq n \to l \leq n$
- $\leq$ is antisymmetric. $\forall\ l\ m : nat,\ l \leq m \to m \leq l \to m = l$

Any relation which is reflexive, transitive and antisymmetric is a *partial order*. Here the word *partial* is used to differentiate $\leq$ from a total order like $<$. We verify the first two properties in Coq, but leave antisymmetry as an exercise.

```
Lemma le_refl: ∀ n:nat,n ≤ n.
intro n.
∃ 0.
reflexivity.
Qed.
```

```
Lemma le_trans : ∀ (l m n : nat), l ≤ m → m ≤ n → l ≤ n.
intros l m n lm mn.
destruct lm as [k klm].
```

```
destruct mn as [j jmn].
∃ (j+k).
rewrite← plus_assoc.
rewrite← klm.
rewrite← jmn.
reflexivity.
Qed.
```

6.6 Decidable properties

We say a predicate $P : A \to$ Prop is *decidable* if we can define a boolean function $decP : A \to bool$ which agrees with the predicate, i.e. $\forall\ a{:}A,\ P\ a \leftrightarrow decP\ a = true$. This also extends to relations in the obvious way.

We show below that equality on natural numbers is decidable. Do you know any undecidable predicates? Is equality always decidable?

First we define the *decision procedure*. In the case of equality this is quite obvious: we inspect both parameters, if they start with different constructors (i.e. 0 vs S) they are certainly not equal. If they are both 0 they are equal, and if they both start with S then we recursively compare the arguments.

```
Fixpoint eqnat (m n : nat) {struct m} : bool :=
  match m with
  | 0 ⇒ match n with
          | 0 ⇒ true
          | S n' ⇒ false
          end
  | S m' ⇒ match n with
            | 0 ⇒ false
            | S n' ⇒ eqnat m' n'
            end
  end.
```

Now we show both direction seperately. The → direction just boils down to showing that *eqnat* is reflexive. Why?

```
Lemma eqnat_refl : ∀ m : nat, eqnat m m = true.
intro m.
induction m.
reflexivity.
simpl.
exact IHm.
Qed.
```

The other direction is more interesting and requires a *double induction* over m and n.

```
Lemma eqnat_compl : ∀ m n : nat, eqnat m n = true → m = n.
intro m.
```

Here it would have been a mistake to do `intros` m n. Why? m = 0 `induction` m.

```
intro n.
induction n.
```

n = 0 `intro` h.

```
reflexivity.
```

n = S n' `intro` h.

```
simpl in h.
discriminate h.
```

m = S m' `intro` n.

```
induction n.
```

n = 0 `intro` h.

```
discriminate h.
```

n = S n' `intro` h.

```
assert (h' : m = n).
apply IHm.
exact h.
rewrite h'.
reflexivity.
Qed.
```

Finally, we can prove the theorem that equality for natural numbers is decidable.

```
Theorem eqnat_dec : ∀ m n : nat, m = n ↔ eqnat m n = true.
intros m n.
split.
intro h.
rewrite h.
apply eqnat_refl.
apply eqnat_compl.
Qed.

End Arith.
```

Chapter 7

Lists

`Section` Lists.

Lists are the ubiqitous datastructure in functional programming, as you should know from Haskell. Given a set A we define **list** A to be the set of finite sequences of elements of A. E.g. the sequence [1,2,3] is an element of **list** **nat**. We can iterate this process and construct lists of lists, e.g. [[1,2],[3]] is an element of **list** **list**(**nat**). However lists are uniform, that is all elements need to have the same type so we cannot form a list like `true`[1,] or [[1,2],3].

We are going to formally introduce lists using an *inductive definition* which has a lot in common with the definition of the natural numbers in the previous chapter. And indeed the theory of lists has a lot in common with the theory of the natural number, so we can call this *list arithmetic*.

7.1 Arithmetic for lists

`Set Implicit` *Arguments*.
`Load` *Arith*.

We define lists *inductively*. Given a set A a list over A is either the empty list nil or it is the result of putting an element a in fornt of an already constructed list l, we write cons a l. nil and cons are *constructors* of **list** A, as 0 and S (successor) were constructors of **nat**.

`Inductive` **list** $(A$: `Set`) : `Set` :=
 | nil : **list** A
 | cons : $A \rightarrow$ **list** $A \rightarrow$ **list** A.

`Implicit Arguments` nil [A].

In functional programming cons is usually written as an infix operation. In Haskell this is : but since this symbol is used for membership in Coq, we use :: instead. Hence the meaning of : and :: in Coq and Haskell are exactly swapped.

Infix "::" := cons (at *level* 60, right *associativity*).

As an example we can define the list [2,3]

```
Definition l23 : list nat
  := 2 :: 3 :: nil.
```

And by consing another 1 in front we obtain [1,2,3].

```
Definition l123 : list nat
  := 1 :: l23.
```

We are going to prove some basic theorems about lists following the development for natural numbers. There we showed that no successor of a natural number is 0 (peano7), here we show that no cons list is equal to the empty list.

```
Theorem nil_cons : ∀ (A:Set)(x:A) (l:list A),
  nil <> x :: l.
intros.
discriminate.
Qed.
```

The next peano axiom peano8 expressed the injectivity of the successor. We have a similar statement for lists: if two cons lists are equal then their tail is equal. To prove this we define tail as we had define predecessor for numbers.

```
Definition tail (A:Set)(l : list A) : list A :=
  match l with
  | nil ⇒ nil
  | cons a l ⇒ l
  end.
```

The proof follows exactly the one for peano8.

```
Theorem cons_injective :
  ∀ (A : Set)(a b : A)(l m : list A),
    a :: l = b :: m → l = m.
intros A a b l m h.
fold (tail (cons a l)).
rewrite h.
unfold tail.
reflexivity.
Qed.
```

However, unlike S, cons has another argument, the head of the list. We can also show that it is injective in this argument, that is if two cons lists are equal then their head is equal.

There is a slight problem in defining head, we cannot (as in Haskell) define head : list $A \to A$, because it could be that A is empty but there is still nil : list A and what should be the head of this list?

To overcome this issue we define head : $A \to$ list $A \to A$ where the first argument is a *dummy argument* which is returned for the empty list.

```
Definition head (A : Set)(x : A)(l : list A) : A :=
```

```
  match l with
  | nil ⇒ x
  | a :: m ⇒ a
  end.
```

Once we have defined head the proof of injectivity is rather straightforward.

```
Theorem cons_injective' :
  ∀ (A : Set)(a b : A)(l m : list A),
     a :: l = b :: m → a = b.
intros A a b l m h.
fold (head a (a :: l)).
rewrite h.
unfold head.
reflexivity.
Qed.
```

As for natural numbers we have also an induction principle for lists: if a property is true for the empty list, and if it holds for a list l then it also holds for cons a l for any a, then it holds for all lists. In Coq we use the same tactic induction to perform list indiuction.

```
Theorem ind_list : ∀ (A : Set)(P : list A → Prop),
  P nil
  → (∀ (a:A)(l : list A), P l → P (a :: l))
  → ∀ l : list A, P l.
intros A P hnil hcons l.
induction l.
exact hnil.
apply hcons.
exact IHl.
Qed.
```

7.2 Lists form a monoid

Previously, we defined addition and multiplication for numbers. There is a very useful operation resembling addition for lists: append. We define app by *structural recursion* over lists.

The idea is that to append a list to the empty list is just that list, and to append a list to a cons list has the same head as the list and the tail is obtained by recursively appending the list to the tail.

```
Fixpoint app (A : Set)(l m:list A) : list A :=
  match l with
  | nil ⇒ m
  | a :: l' ⇒ a :: (app l' m)
```

```
end.
```

As in Haskell we use the inifx operation ++ to denote append.

```
Infix "++" := app (right associativity, at level 60).
```

As an example we construct the list [2,3,1,2,3] by appending [2,3] and [1,2,3].

```
Eval compute in (l23 ++ l123).
```

We show that **list** A with ++ and nil forms a monoid. Indeed the proofs are basically the same as for (**nat**,+,0).

```
Theorem app_nil_l : ∀ (A : Set)(l : list A),
  nil ++ l = l.
intros A l.
reflexivity.
Qed.
```

```
Theorem app_l_nil : ∀ (A : Set)(l : list A),
  l ++ nil = l.
intros A l.
induction l.
reflexivity.
simpl.
rewrite IHl.
reflexivity.
Qed.
```

```
Theorem assoc_app : ∀ (A : Set)(l m n : list A),
  l ++ (m ++ n) = (l ++ m) ++ n.
intros A l m n.
induction l.
reflexivity.
simpl.
rewrite IHl.
reflexivity.
Qed.
```

7.3 Reverse

While there are many similarities between **nat** and **list** A there are important differences. Commutativity $l ++ m = m ++ l$ does not hold (what would be a counterexample?). Hence (list A,++,nil) is an example of a non-commutative monoid. Since commutativity doesn't hold it makes sense to reverse a list (while it didn't make sense to reverse a number).

To define reverse, we first define the operation snoc which adds an element at the end of a given list. This operation again is defined by primitive recursion.

```
Fixpoint snoc (A:Set)
  (l : list A)(a : A) {struct l} : list A
  := match l with
      | nil ⇒ a :: nil
      | b :: m ⇒ b :: (snoc m a)
      end.
```

There is an alternative way to define snoc just by using ++. Can you see how?

As an example we put 1 at the end of [2,3]

```
Eval compute in (snoc l23 1).
```

Using snoc it is easy to define rev by primitive recursion. The reverse of an empty list is the empty list. To reverse a cons list, reverse its tail and then snoc the head to the end of the result.

```
Fixpoint rev
  (A:Set)(l : list A) : list A :=
  match l with
  | nil ⇒ nil
  | a :: l' ⇒ snoc (rev l') a
  end.
```

This definition of rev is called *naive reverse* and it is rather inefficient. Can you see why? How can it be improved?

Some examples.

```
Eval compute in rev l123.
Eval compute in rev (rev l123).
```

The 2nd example gives rise to a theorem about rev, namely that to reverse twice is the identity (rev rev(l) = l).

To prove it we first prove a lemma about rev and snoc. How did we discover this lemma?

```
Lemma revsnoc : ∀ (A:Set)(l:list A)(a : A),
  rev (snoc l a) = a :: (rev l).
intros A l a.
```

We proceed by induction over l.

```
induction l.
simpl.
reflexivity.
simpl.
rewrite IHl.
simpl.
reflexivity.
Qed.
```

And now we can prove the theorem.

```
Theorem revrev :
  ∀ (A:Set)(l:list A),rev (rev l) = l.
intros A l.
induction l.
simpl.
reflexivity.
simpl.
```

And now it seems that revsnoc is exactly what we need. Lucky that we proved it already.

```
rewrite revsnoc.
rewrite IHl.
reflexivity.
Qed.
```

7.4 Insertion sort

Our next example is sorting: we want to sort a given lists according to an given order. E.g. the list

4 :: 2 :: 3 :: 1 :: nil

should be sorted into

1 :: 2 :: 3 :: 4 :: nil

We will implement and verify "insertion sort". To keep things simple we will sort lists of natural numbers wrt to the <= order. First we implement a boolean function which compares two numbers:

```
Fixpoint leqb (m n : nat) {struct m} : bool :=
  match m with
  | 0 => true
  | S m ⇒ match n with
            | 0 ⇒ false
            | S n ⇒ leqb m n
            end
  end.
```

```
Eval compute in leqb 3 4.
Eval compute in leqb 4 3.
```

Notation "m <= n" := (leq m n).

We just assume that leq decided ≤. I leave it as an exercise to formally prove this, i.e. to replace the axioms by lemmas or theorems.

```
Axiom leq1 : ∀ m n : nat, leqb m n = true → m ≤ n.
Axiom leq2 : ∀ m n : nat, m ≤ n → leqb m n = true.
```

The main function of insertion sort is the function insert which inserts a new element into an already sorted list:

```
Fixpoint insert (n:nat)(ms : list nat) {struct ms} : list nat :=
  match ms with
  | nil ⇒ n::nil
  | m::ms' ⇒ if leqb n m
              then n::ms
              else m::(insert n ms')
  end.
```

```
Eval compute in insert 3 (1::2::4::nil).
```

Now sort builds a sorted list from any list by inserting each element into the empty list.

```
Fixpoint sort (ms : list nat) : list nat :=
  match ms with
  | nil ⇒ nil
  | m::ms' ⇒ insert m (sort ms')
  end.
```

```
Eval compute in sort (4::2::3::1::nil).
```

```
Fixpoint Sorted (l : list nat) : Prop :=
  match l with
  | nil ⇒ True
  | a :: m ⇒ Sorted m ∧ a <= head a m
  end.
```

Here is another assumption about $\leq$ I am not going to prove but leave as an exercise.

```
Axiom total : ∀ m n : nat, m ≤ n ∨ n ≤ m.
```

Our goal is to show that insert preserves sortedness, i.e. Sorted $l \rightarrow$ Sorted insert(n l). To prove this we need to lemmas.

The first one is useful in the case when the new element is not smaller than the current head. In this case we need to know that the head is smaller than the new element so that we can insert it later.

```
Lemma leqFalse : ∀ m n : nat, leqb m n = false → n ≤ m.
intros m n h.
destruct (total m n) as [mn | nm].
assert (mnt : leqb m n = true).
apply leq2.
exact mn.
rewrite h in mnt.
discriminate mnt.
exact nm.
Qed.
```

The other lemma is a little case analysis: the head of the result of insert is either the inserted element or the previous head.

```
Lemma insertSortCase : ∀ (n a : nat)(l : list nat),
  head a (insert n l) = n ∨ head a (insert n l) = head a l.
intros n a l.
```

While we say `induction` we are not going to use the induction hypothesis here. So we could have used `destruct` on lists here.

```
induction l.
left.
simpl.
reflexivity.
simpl.
destruct (leqb n a0).
left.
simpl.
reflexivity.
right.
simpl.
reflexivity.
Qed.
```

We are now able to prove the main lemma on insert.

```
Lemma insertSorted : ∀ (n : nat)(l : list nat),
  Sorted l → Sorted (insert n l).
intros n l.
```

We prove the implication by induction. Why did we not do another `intro`?

```
induction l.
```

The case for the empty list is easy.

```
intro h.
simpl.
split.
split.
apply le_refl.
```

Now the cons case

```
intro h.
simpl.
simpl in h.
destruct h as [sl al].
```

We now analyze the result of the comparison.

```
case_eq (leqb n a).
```

First case leqb n a = true, that is the element is put in front.

```
intro na.
```

```
simpl.
split.
split.
exact sl.
exact al.
```

Here we need the correctness of leq wrt $\leq$.

```
apply leq1.
exact na.
```

Second case leqb n a = false so we insert a in the tail Here we need our lemmas.

```
intro na.
simpl.
split.
apply IHl.
exact sl.
```

Here we have to reason about the head of insert n l, so we use our lemma.

```
destruct (insertSortCase n a l) as [H1 | H2].
```

First case: it is the new element.

```
rewrite H1.
apply leqFalse.
exact na.
```

Second case: it is the old head.

```
rewrite H2.
exact al.
Qed.
```

using the previous lemma it is easy to prove our main theorem.

```
Theorem sortSorted : ∀ ms:list nat,Sorted (sort ms).
induction ms.
```

case ms=nil:

```
  simpl.
  split.
```

case a::ms

```
  simpl.
  apply insertSorted.
  exact IHms.
Qed.
```

Is this enough? No, we could have implemented a function with the property sort_ok by always returning the empty list. Another important property of a sorting function is that it returns a permutation of the input. I leave this as an exercise.

`End` Lists.

Chapter 8

Compiling expressions

```
Section Expr.
```

We are going to use the standard library for lists.

```
Require Import Coq.Lists.List.

Set Implicit Arguments.
```

8.1 Evaluating expressions.

We define a simple language of expressions over natural numbers: only containing numeric constants and addition. This is already a useful abstraction over the one-dimensional view of a program as a sequence of symbols, i.e. we don't care about precedence or balanced bracktes.

However, this means that at some point we'd have to implement a parser and verify it.

```
Inductive Expr : Set :=
  | Const : nat → Expr
  | Plus : Expr → Expr → Expr.
```

The expression "(3 + 5) + 2" is represented by the following tree:

```
Definition e1 : Expr := Plus (Plus (Const 3) (Const 5)) (Const 2).
```

We give a "denotational" semantics to our expressions by recursively assigning a value (their denotation). This process is called evaluation - hence the function is called *eval*. It is defined by structural recursion over the structure of the expression tree.

```
Fixpoint eval (e:Expr) {struct e} : nat :=
  match e with
  | Const n ⇒ n
  | Plus e1 e2 ⇒ (eval e1) + (eval e2)
  end.
```

Let's evaluate our example expression:

```
Eval compute in (eval e1).
```

8.2 A stack machine

We are going to describe how to calculate the value of an expression on a simple stack machine - thus giving rise to an "operational semantics".

First we specify the operation of our machine, there are only two operations :

```
Inductive Op : Set :=
  | Push : nat → Op

  | PlusC : Op.
Definition Code := list Op.
Definition Stack := list nat.
```

We define recursively how to execute code wrt any given stack. This function proceeds by linear recursion over the stack and could be easily implemented as a "machine".

```
Fixpoint run (st:Stack)(p:Code) : nat := match p with
    | nil ⇒ match st with
              | nil ⇒ 0
              | n :: st' ⇒ n
              end
    | op :: p' ⇒
        match op with
        | Push n ⇒ run (n :: st) p'
        | PlusC ⇒ match st with
                    | nil ⇒ 0
                    | n :: nil ⇒ 0
                    | n1 :: n2 :: st' ⇒
                        run ((n2 + n1) :: st') p'
                    end
        end
    end.
```

We run a piece of code by starting with the empty stack.

```
Definition c1 : Code
  := Push 2 :: Push 3 :: PlusC :: nil.
Eval compute in (run nil c1).
```

8.3 A simple compiler

We implement a simple compiler which translates an expression into code for the stack machine.

A naive implementation looks like this:

```
Fixpoint compile (e:Expr) : list Op :=
  match e with
  | Const n ⇒ (Push n) :: nil
  | Plus e1 e2 ⇒ (compile e1)++
                    (compile e2)++
                    (PlusC::nil)
```

Why do we need to do this in this order?

```
  end.
```

We test the compiler `Eval compute in` *compile e1*.

```
Eval compute in run nil (compile e1).
Eval compute in eval e1.
```

The agreement of the last two lines is not a coincidence. Indeed, our compiler is correct because compile and run produces the same result as evaluation, i.e. we want to prove ∀ *e* : *Expr*, *run nil* (*compile e*) = *eval e* We will be using induction over trees here. However, in this form the result won't go through because the stack will change during the proof. We have to generalize the statement:

```
Lemma compile_lem :
  ∀ (e:Expr)(st : Stack)(p : Code),
    run st ((compile e)++p) = run (eval e :: st) p.
intros e.
```

It would be wrong to do additional intros because both *st* and *p* are going to vary during the proof. Here we are going to use induction over trees. `induction` *e*.

The Const case is straightforward `intros` *p st*.

```
simpl.
reflexivity.
```

The Plus case is more intresting. Note that we have two induction hypotheses: one for each of the subtrees. `intros` *p st*.

```
simpl.
```

We need to reorganize the code argument to be able to apply the induction hypothesis.

```
rewrite app_ass.
rewrite IHe1.
```

And again for the 2nd induction hypotehsis. `rewrite` *app_ass*.

```
rewrite IHe2.
simpl.
reflexivity.
```

```
Qed.
```

We are now ready to prove compiler correctness.

```
Theorem compile_ok :
  ∀ e : Expr, run nil (compile e) = eval e.
intro e.
pattern (compile e).
```

To be able to apply the lemma we need to use the fact that $e{+}{+}nil{=}e$. `rewrite app_nil_end.`

```
rewrite compile_lem.
simpl.
reflexivity.
Qed.
```

8.4 A continuation based compiler

A better alternative both in terms of efficiency and verification is a "continuation based" compiler. We compile an expression e wrt a continuation p, some code which is going to be run after it.

```
Fixpoint compile_cont (e:Expr) (p:Code) : Code := match e with
   | Const n ⇒ Push n :: p
   | Plus e1 e2 ⇒ compile_cont e1
                        (compile_cont e2 (PlusC :: p))
   end.
```

Test the compiler

```
Eval compute in compile_cont e1 nil.
```

And run the compiled code:

```
Eval compute in run nil (compile_cont e1 nil).
```

As before we prove a lemma to show compiler correctness. Note that we don't need to use ++ anymore.

```
Lemma compile_cont_lem : ∀ (e:Expr)(p:Code)(st:Stack),
  run st (compile_cont e p) = run ((eval e)::st) p.
induction e.
intros p st.
simpl.
reflexivity.
simpl.
intros.
```

Even better: no need to appeal to associativity of ++ here. `rewrite IHe1.`

Or here. `rewrite IHe2.`

```
simpl.
reflexivity.
Qed.
```

The main theorem is a simple application of the previous lemma:

Theorem *compile_cont_ok* : ∀ *e*:*Expr*,
  run nil (*compile_cont e nil*) = *eval e*.
intro *e*.

No need to use $l{+}{+}nil{=}l$ here. apply *compile_cont_lem*.

Qed.

To summarize: the continuation based proof results in a more efficient program and in a simpler proof.

End *Expr*.