Introduction to Formal Reasoning (G52IFR)

Thorsten Altenkirch

November 14, 2011
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.
- 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 Library doc: http://coq.inria.fr/library-eng.html
- Coq’Art, the book by Yves Bertot and Pierre Casteran (2004). (available in the library!)

1.6 Course organisation

- Course page: http://www.cs.nott.ac.uk/~txa/g52ifr/
- Coq Labs: every Tuesday (1600 - 1800) in A32, starting next week (11/10).
- Weekly coursework, 1st available next Monday.
- Use online submission system (cw).
- Tutorials: start next week. See assignment on webpage.
- Online class test on coq in December.
- Online Forum: Information about coursework, discuss questions Please subscribe!
Chapter 2

Propositional Logic

Section \textit{prop}.

A proposition is a definitive statement which we may be able to prove. In Coq we write $P : \text{Prop}$ 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:

\textbf{Variables} $P Q R : \text{Prop}$.

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 $\land$, read $P \land Q$ as $P$ and $Q$.
- Disjunction $\lor$, read $P \lor Q$ as $P$ or $Q$.
- \textit{False}, read \textit{False} as "Pigs can fly".
- \textit{True}, read \textit{True} as "It sometimes rains in England."
- Negation $\neg$, read $\neg P$ as not $P$. We define $\neg P$ as $P \rightarrow \text{False}$.
- Equivalence, $\leftrightarrow$, read $P \leftrightarrow Q$ as $P$ is equivalent to $Q$. We define $P \leftrightarrow Q$ as $(P \rightarrow Q) \land (Q \rightarrow P)$.

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 $P \lor Q \rightarrow R$ as $((P \lor Q) \rightarrow R)$.

• 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 $(\neg P) \land Q$.

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 $P \rightarrow P$, i.e. if $P$ then $P$. To start a proof we write:

```
Lemma I : P \rightarrow 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 $P \rightarrow P$. 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 $(P \rightarrow Q) \rightarrow (Q \rightarrow R) \rightarrow P \rightarrow R$. 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 \texttt{apply} tactic: if we have assumed \( P \rightarrow Q \) 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** \( C : (P \rightarrow Q) \rightarrow (Q \rightarrow R) \rightarrow P \rightarrow R. \)

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

\begin{verbatim}
intro pq.
we continue assuming...
intro qr.
intro p.

Now we have three assumptions \( P \rightarrow Q \), \( Q \rightarrow R \) and \( P \). It remains to prove \( R \). We cannot use \texttt{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 \( Q \rightarrow R \). We use the \texttt{apply} tactic.

\texttt{apply qr}.

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

\texttt{apply pq}.

And now it only remains to prove \( P \) which is one of our assumptions - hence we can use \texttt{exact} again. \texttt{exact p}.

\texttt{Qed}.
\end{verbatim}

### 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 \texttt{intro}. To prove \( P \rightarrow Q \), we assume \( P \) and prove \( Q \).

- **elimination**: How can we use an assumption? In the case of implication this is \texttt{apply}. If we know \( P \rightarrow Q \) and we want to prove \( Q \) it is sufficient to prove \( P \).

Actually \texttt{apply} is a bit more general: if we know \( P_1 \rightarrow P_2 \rightarrow ... \rightarrow P_n \rightarrow Q \) and we want to prove \( Q \) then it is sufficient to prove \( P_1,P_2,...,P_n \). 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 : $P \rightarrow Q \rightarrow P \land Q$.

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 $P \land Q \rightarrow Q \land P$.

Lemma andCom : $P \land Q \rightarrow Q \land P$.
intropq.

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 $P \land Q$ we prove it from assuming both $P$ and $Q$.

### 2.5 The currying theorem

Maybe you have already noticed that a statement like $P \rightarrow Q \rightarrow R$ basically means that $R$ can be proved from assuming both $P$ and $Q$. Indeed, it is equivalent to $P \land Q \rightarrow R$. 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**: $(P \land Q \rightarrow R) \leftrightarrow (P \rightarrow Q \rightarrow R)$.

- 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 $Int \rightarrow Int \rightarrow Int$.

### 2.6 Disjunction

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

**Lemma inl**: $P \rightarrow P \lor Q$.

- intros $p$.

Clearly, here we have to use **left**.

**left**.
- exact $p$. 

7
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 \texttt{destruct} but in this case \texttt{destruct} creates two subgoals. This can be compared to case analysis in functional programming. Indeed we can prove the following theorem.

**Lemma case**: $P \lor Q \rightarrow (P \rightarrow R) \rightarrow (Q \rightarrow R) \rightarrow R$.

```plaintext
intros pq pr qr.
destruct pq as [p | q].
```

The syntax for \texttt{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 $P$.

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

And then we assume $Q$

```plaintext
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 \texttt{left} to prove it from $P$ and \texttt{right} to prove it from $Q$.

- **elimination**: If we have assumed $P \lor Q$ then we can use \texttt{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. $P \land (Q \lor R)$ 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**: $P \land (Q \lor R) \leftrightarrow (P \land Q) \lor (P \land R)$.

```plaintext
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 \( \land \) 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 \( \rightarrow \) 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 \( P \rightarrow \text{False} \). 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** \( \text{incons} : \neg (P \land \neg P) \).

\[
\begin{align*}
\text{unfold} & \quad \text{not.} \\
\text{intro} & \quad h. \\
\text{destruct} & \quad h \text{ as } [p \ np]. \\
\text{apply} & \quad np. \\
\text{exact} & \quad p. \\
\text{Qed.}
\end{align*}
\]

Another example is to show that \( P \) implies \( \neg \neg P \).

**Lemma** \( p\negnp : P \rightarrow \neg \neg P \).

\[
\begin{align*}
\text{unfold} & \quad \text{not.} \\
\text{intros} & \quad p \ np. \\
\text{apply} & \quad np. \\
\text{exact} & \quad p. \\
\text{Qed.}
\end{align*}
\]

### 2.10 Classical Reasoning

You may expect that we can also prove the other direction \( \neg \neg P \rightarrow P \) and that indeed \( P \leftrightarrow \neg \neg P \). We can reason that \( P \) is either \textbf{True} or \textbf{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 \textbf{True} or \textbf{False}, this is the Principle of the Excluded Middle \( P \lor \neg P \). In Coq this can be achieved by:

**Require Import** \textit{Coq.Logic.Classical}.

This means we are now using Classical Logic instead of Intuitionistic Logic. The only difference is that we have an axiom \textit{classic} which proves the principle of the excluded middle for any proposition. We can use this to prove \( \neg \neg P \rightarrow P \).
Lemma \( nnpp : \sim\neg P \rightarrow P. \)
intro \( nnp. \)

Here we use a particular instance of \textit{classic} for \( P. \)

\texttt{destruct (classic } \( P \texttt{) as } [p \mid np]. \)

First case \( P \) holds

\texttt{exact } \( p. \)

2nd case \( \neg P \) holds. Here we appeal to \textit{exFalso}.

\texttt{apply exFalso.}

Notice that we have shown \textit{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.

\texttt{apply } \( nnp. \)

\texttt{exact } \( np. \)

Qed.

Unless stated otherwise we will try to prove propositions intuitionsitically, that is without using \textit{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 intuitionistic 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.

\textbf{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 \texttt{assert}. \texttt{assert } \( h : H \texttt{ introduces } H \texttt{ as a new subgoal and after you have proven this you can use an assumption } h : H \texttt{ to prove your original goal.}

The following (artificial) example demonstrates the use of \texttt{assert}.

\textbf{Lemma usecut} : \( (P \land \neg P) \rightarrow Q. \)
intro \( pnp. \)

If we had a generic version of \textit{exFalso} we could use this. Instead we can introduce \textit{False} as an intermediate goal. \texttt{assert } \( f : False). \)

which is easy to prove \texttt{destruct } \( pnp \texttt{ as } [p \mid np] \).

\texttt{apply } \( np. \)
exact $p$.

and using $False$ it is easy to prove $Q$. \texttt{destruct} $f$.

Qed.

This example also shows that sometimes we have to cut (i.e. use \texttt{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 $A \ B : \text{Set}$.

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 concrete 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. $P \ x \ \text{may mean} \ P \ \text{is clever and} \ Q \ x \ \text{means} \ x \ \text{is funny}$).

Variables $P \ Q : A \rightarrow \text{Prop}$.

Coq views these predicates as functions from $A$ to $\text{Prop}$. 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 $R : A \rightarrow B \rightarrow \text{Prop}$.

E.g. $R$ could be the relation "attends" and we would write "$R \ \text{jim} \ \text{g52ifr}$" 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 $\forall x:A, P x \rightarrow Q x$ 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 elements 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 $\forall (x:A, P x) \rightarrow \forall (x:A, P x \rightarrow Q x) \rightarrow \forall x:A, Q x$.

We introduce some new syntactic conventions: the scope of an forall always goes as far as possible. That is we read $\forall x:A, P x \land Q$ as $\forall x:A, (P x \land Q)$. 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 :** $(\forall x:A, P x) \rightarrow (\forall x:A, P x \rightarrow Q x) \rightarrow \forall x:A, Q x$.

**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 $H2 : \forall x:A, P x \rightarrow Q x$ and we want to prove $Q a$ we can use **apply H2** to instantiate the assumption to $P a \rightarrow Q a$ 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 occurrence of $x$ is replaced by $a$.

- **elimination for $\forall$**: We only describe the simplest case: If we know $H : \forall x:A, P x$ and we want to show $P a$, we use **apply H** to prove $P a$.

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

We can also use **intros** here. That is if the current goal is $\forall x:A, P x \rightarrow Q x$ then **intros** $x P$ will introduce the assumptions $x:A$ and $H:P x$. 14
The general case for **apply** is a bit hard to describe. Basically apply may introduce several subgoals if the assumption has a prefix of ∀ and →. 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 ∀ commutes with ∧. That is we are going to show \( \forall( x:A, P x \land Q x) \leftrightarrow \forall( x:A, P x) \land \forall( x:A, Q x) \) that is "all students are clever and funny" is equivalent to "all students are clever" and "all students are funny".

**Lemma AllAndCom** : \( (\forall x:A, P x \land Q x) \leftrightarrow (\forall x:A, P x) \land (\forall x:A, Q x) \).

**split.**

Proving \( \rightarrow \)

```
intro H.
split.
intro a.
assert (pq : P a \land Q a).
apply H.
destruct pq as [p q].
extact p.
intro a.
assert (pq : P a \land Q a).
apply H.
destruct pq as [p q].
extact 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** : \( (\forall x:A, P x \land Q x) \rightarrow (\forall x:A, P x) \land (\forall x:A, Q x) \).

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 \( \lor \)? That is does \( \forall( x:A, P \lor Q \ x) \leftrightarrow \forall( x:A, P \ x) \lor \forall( x:A, Q \ x) \) 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 \( \exists \ x:A, P \ x \land Q \ x \) 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 \( (\exists x:A, P \ x) \rightarrow (\forall x:A, P \ x) \rightarrow Q x \) \( \rightarrow (\exists x:A, Q x) \).

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 \( \exists x:A, P \ x \land Q \) as \( \exists x:A, (P \ x \land Q) \).

The tactics for existential quantification 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 occurrence of \( x \) is replaced by \( a \). To use an assumption \( H : \exists x:A, PP \) we employ \texttt{destruct} \( H \) as \([a \ p]\) which destructs \( H \) into \( a : A \) and \( p : PP' \) where \( PP' \) is \( PP \) where all free occurrences of \( x \) have been replaced by \( a \).

**Lemma** \texttt{ExistsMono} : \( (\exists x:A, P \ x) \rightarrow (\forall x:A, P \ x \rightarrow Q x) \rightarrow (\exists x:A, Q x) \).

\texttt{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.

\texttt{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 occurrence 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 $H$ into two assumptions: $a : A$ and $p : P'$ where $P'$ is obtained from $P$ by replacing all free occurrences of $x$ in $P$ by $a$.

Next we are going to show that $\exists$ commutes with $\lor$. That is we are going to show $(\exists x:A, P x \lor Q x) \leftrightarrow (\exists x:A, P x) \lor (\exists x:A, Q x)$ 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 $ExOrCom : (\exists x:A, P x \lor Q x) \leftrightarrow (\exists x:A, P x) \lor (\exists x:A, Q x)$.

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 $\exists x:A, Q x$
3.3 Another Currying Theorem

There is also a currying theorem in predicate logic which exploits the relation between $\rightarrow$ and $\forall$ on the one hand and $\land$ and exists on the other. That is we can show that $\forall x: A, P x \rightarrow S$ is equivalent to $(\exists x: A, P x) \rightarrow S$. 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 $\rightarrow$ and $\exists$ by $\land$.

To prove this tautology we assume an additional proposition.

Variable $S : \text{Prop}$.

Lemma Curry : $(\forall x : A, P x \rightarrow S) \leftrightarrow ((\exists x : A, P x) \rightarrow S)$.

split.

proving $\rightarrow$

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 ($H' : P a \rightarrow S$).

apply $H$.

apply $H'$.

exact $p$.

proving $\leftarrow$.

intro $H$.

intros $a$ $p$.

apply $H$.

$\exists a$.

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 : (∀ x:A, P x → S) → ((∃ x:A, P x) → S).

    proving →
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 (∀ a:A, a = a)
- symmetric (∀ a b:A, a=b → b=a)
- transitive (∀ a b c:A, a=b → b=c → a=c.

Lemma eq_refl : ∀ a:A, a = a.
intro a.
Here we just invoke the reflexivity tactic.
reflexivity.
Qed.

Lemma eq_sym : ∀ a b:A, a=b → b=a.
intros a b H.
Here we use rewrite to reduce the goal.
rewrite H.
reflexivity.
Qed.

Lemma eq_trans : ∀ a b c:A, a=b → b=c → 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 $P : P \lor \neg P$ 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.


Lemma ex_from_forall : ($\exists x : A, P x$) $\iff \neg \forall x : A, \neg P x$.

split.

proving $\rightarrow$
intro ex.
intro H.
destruct ex as [a p].
assert (npa : $\neg (P a)$).
apply H.
apply npa.
exact p.

proving $\leftarrow$
intro H.
apply NNPP.

Instead of proving $\exists x : A, P x$ which is hard, we show $\neg \neg \exists x : A, P x$ which is easier. intronex.
apply H.
intros a p.
apply nex.
exists a.
exact p.
Qed.
Chapter 4

Bool

Section Bool.

4.1 Defining bool and operations

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

The function \( \text{negb} : \text{bool} \rightarrow \text{bool} \) (boolean negation) can be defined by pattern matching using the match construct.

Definition \( \text{negb}(b: \text{bool}) : \text{bool} := \)

\[
\begin{align*}
\text{match } b \text{ with} \\
\quad | \text{true} &\Rightarrow \text{false} \\
\quad | \text{false} &\Rightarrow \text{true}
\end{align*}
\]

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 \( \text{negb true} \).

The evaluator replaces

\( \text{negb true} \)

with

\[
\text{match } \text{true} \text{ with } | \text{true} \Rightarrow \text{false} | \text{false} \Rightarrow \text{true} \text{ end.}
\]

which in turn evaluates to

\( \text{false} \)

Eval compute in \( \text{negb (negb true)} \).

We know already that \( \text{negb true} \) evaluates to \( \text{false} \) hence \( \text{negb (negb true)} \) evaluates to \( \text{negb false} \) which in turn evaluates to \( \text{true} \).
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` is idempotent, that is

`∀ 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_idem : ∀ b : bool, negb (negb b) = b.`

```coq
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`.

```coq
simpl.
reflexivity.
```

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

```coq
simpl.
reflexivity.
```

Qed.

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

`∀ x y : bool, andb x y = andb y x`
(we use the abbreviation: \( \forall x \ y : A, \ldots \) is the same as \( \forall x: A \forall y: A, \ldots \))

Lemma \textit{andb\_comm} : \( \forall x \ y : \text{bool}, \text{andb} \ x \ y = \text{andb} \ y \ x \).

intros \( x \ y \).
destruct \( x \);
   (destruct \( y \);
     reflexivity).
Qed.

We can also prove other properties of \textit{bool} not directly related to the functions, for example, we know that every boolean is either true or false. That is
\[ \forall b : \text{bool}, b = \text{true} \lor b = \text{false} \]

This is easy to prove:

Lemma \textit{true\_or\_false} : \( \forall b : \text{bool}, b = \text{true} \lor b = \text{false} \).

intro \( b \).
destruct \( b \).
   \( b = \text{true} \) \textit{left}.
reflexivity.
   \( b = \text{false} \) \textit{right}.
reflexivity.
Qed.

Next we want to prove something which doesn’t involve any quantifiers, namely
\[ \neg (\text{true} = \text{false}) \]

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

Definition \textit{Istrue} \( (b : \text{bool}) : \text{Prop} := \)
match \( b \) with
| \text{true} \Rightarrow \text{True} 
| \text{false} \Rightarrow \text{False} 
end.

Lemma \textit{diff\_true\_false} :
\[ \neg (\text{true} = \text{false}). \]
intro \( h \).

We now need to use a new tactic to replace \textit{False} by \textit{Istrue False}. This is possible because \textit{Istrue False} evaluates to \text{false}. We are using \textit{fold} which is the inverse to \textit{unfold} which we have seen earlier.

fold \( (\text{Istrue false}) \).

Now we can simply apply the equation \( h \) backwards.

rewrite\textsuperscript{←} \( h \).

Now by unfolding we can replace \textit{Istrue true} by \text{True}
unfold \textit{Itrue}.

Which is easy to prove.

\texttt{split. Qed.}

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

4.3 Reflection

We notice that there is a logical operator $\land$ which acts on \texttt{Prop} and a boolean operator \texttt{andb} (or \\&\\) which acts on \texttt{bool}. How are the two related?

We can use $\land$ to specify \texttt{andb}, namely we say that \texttt{andb x y = true} is equivalent to $x = \textit{true}$ and $y = \textit{true}$. That is we prove:

\begin{verbatim}
Lemma \_ok : \forall x y : bool, andb x y = true \iff x = true \land y = true.
intros x y.
split.
→
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.
End Bool.
\end{verbatim}
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.

In Mathematics (and conventional Set Theory), we just write \( C = \{ c_1, c_2, \ldots, c_n \} \) for a finite set.

In Coq we write

```coq
Inductive C : Set := c1 : C | c2 : C | \ldots | cn : C.
```

As a special example we define the empty set:

```coq
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.

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

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

**Inductive** \( XCoord : \text{Set} := \)
- \( xa : XCoord \)
- \( xb : XCoord \)
- \( xc : XCoord \)
- \( xd : XCoord \)
- \( xe : XCoord \)
- \( xf : XCoord \)
- \( xg : XCoord \)
- \( xh : XCoord \).

**Inductive** \( YCoord : \text{Set} := \)
- \( y1 : YCoord \)
- \( y2 : YCoord \)
- \( y3 : YCoord \)
- \( y4 : YCoord \)
- \( y5 : YCoord \)
- \( y6 : YCoord \)
- \( y7 : YCoord \)
- \( y8 : YCoord \).

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 \( \text{oneUp} : YCoord \rightarrow YCoord \) 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** \( \text{oneUp} (y : YCoord) : YCoord := \)
```
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 $A, B : \text{Set}$ we define a new set $A \times B : \text{Set}$ 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($A B : \text{Set}$)($p : A \times B$) : $A$ :=

\[
\text{match } p \text{ with } \\
| (a , b) \Rightarrow a \\
\end{match}
\]

**Definition** snd($A B : \text{Set}$)($p : A \times B$) : $B$ :=

\[
\text{match } p \text{ with } \\
| (a , b) \Rightarrow b \\
\end{match}
\]

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:

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

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 : $\forall A B : \text{Set},$

$\forall p : \text{prod } A B, (\text{fst } p , \text{snd } p) = p.$

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. destructor $p$ 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 $|A \ast B|$?

27
5.3 Disjoint union

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

It is important not to confuse $+$ with the union of sets. The disjoint union of $\text{bool}$ with $\text{bool}$ has 4 elements because $\text{inl true}$ is different from $\text{inr true}$ while in the union of $\text{bool}$ with $\text{bool}$ there are only 2 elements since there is only one copy of $\text{true}$. 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 $\text{Empty}$ which has just one element $\text{empty}$.

\begin{verbatim}
Inductive Empty : Set :=
  | empty : Empty.

Definition Field : Set := Piece + Empty.
\end{verbatim}

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 $\text{Empty}$ which has just one element $\text{empty}$.

\begin{verbatim}
Definition blackKnightHere : Field := inl blackKnight.
Definition emptyField : Field := inr empty.
\end{verbatim}

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

\begin{verbatim}
Definition swap(A B : Set)(x : A + B) : B + A :=
  match x with
  | inl a ⇒ inr a
  | inr b ⇒ inl b
  end.
\end{verbatim}

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 $A$ and $B$ we define a new set $A \rightarrow B$ which is the set of functions from $A$ 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 $\text{fun } x \Rightarrow b$ where $b$ is an expression in $B$ which may refer to $x : A$. 

28
In the case of our chess example we can use functions to define a chess board as a function form \textit{Coord} to \textit{Field}, this function would give us the content of a field for any coordinate.

\begin{verbatim}
Definition Board : Set := Coord \rightarrow Field.
\end{verbatim}

A particular simple example is the empty board:

\begin{verbatim}
Definition EmptyBoard : Board := fun x \Rightarrow emptyField.
\end{verbatim}

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

As another example instead of defining \textit{negb} as an operation we could also have used \textit{fun}:

\begin{verbatim}
Definition negb' : bool \rightarrow bool :=
  fun (b : bool) \Rightarrow match b with
  | true \Rightarrow false
  | false \Rightarrow true
end.
\end{verbatim}

Using \textit{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 \textit{isConst} which determines wether a given function \( f : \text{bool} \rightarrow \text{bool} \) is constant.

\begin{verbatim}
Open Scope bool_scope.

Definition isConst (f : bool \rightarrow bool) : bool :=
  (f true) \&\& (f false) || negb (f true) \&\& negb (f false).
\end{verbatim}

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

\begin{verbatim}
Eval compute in isConst negb.
Eval compute in isConst (fun x \Rightarrow false).
Eval compute in isConst (fun x \Rightarrow true).
Eval compute in isConst (fun x \Rightarrow x).
\end{verbatim}

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 \( A \rightarrow B \)? 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.

\begin{verbatim}
Axiom ext : \forall (A B : Set)(f g : A \rightarrow B), (\forall x:A,f x = g x) \rightarrow f = g.
\end{verbatim}

\section*{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 ($\land$) is translated as product ($\times$),
• disjunction ($\lor$) is translated as disjoint union (+),
• implication ($\to$) is translated as function set ($\to$).

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** ($A \ B \ C : \text{Set}$) : ($A \times B \to C$) $\to$ ($A \to B \to C$) := 
fun $f$ $\mapsto$ fun $a$ $\mapsto$ fun $b$ $\mapsto$ $f$ ($a$, $b$).

**Definition curry’** ($A \ B \ C : \text{Set}$) : ($A \to B \to C$) $\to$ ($A \times B \to C$) := 
fun $g$ $\mapsto$ fun $p$ $\mapsto$ $g$ (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** : $\forall$ $A \ B \ C : \text{Set}$, $\forall$ $f : A \times B \to C$,
$f = (\text{curry’}(\text{curry } f))$.

intros $A \ B \ C \ f$.
apply ext.
intro $p$.
destruct $p$.
reflexivity.
Qed.

**Lemma curryIso2** : $\forall$ $A \ B \ C : \text{Set}$, $\forall$ $g : A \to B \to C$,
$g = (\text{curry}(\text{curry’ } g))$.

intros $A \ B \ C \ g$.
apply ext.
intro $a$.
apply ext.
intro $b$.
reflexivity.
Qed.

End Sets.
Chapter 6

Peano Arithmetic

Section Arith.

6.1 The natural numbers

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

- $1 = S\ 0$
- $2 = S\ 1 = S\ (S\ 0)$
- $3 = S\ 2 = S\ (S\ (S\ 0))$

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

For illustration we are going to prove these principles:

Lemma peano7: $\forall\ n:\text{nat}, S\ n \neq 0$. 

31
intro \( n \).
intro \( h \).

This is basically the same problem as proving \( true \neq false \), we could apply the same technique here. To avoid repetition we just use the \texttt{discriminate} tactic.

discriminate \( h \).
Qed.

To prove the next axiom, it is useful to define the inverse to \( S \), the predecessor function \( \text{pred} \). We arbitrarily decide that the predecessor of \( 0 \) is \( 0 \).

\textbf{Definition} \( \texttt{pred} \) : \( n : \text{nat} \mapsto \text{nat} \) :=
match \( n \) with
| \( 0 \) \( \Rightarrow 0 \)
| \( S \, n \) \( \Rightarrow n \)
end.

\textbf{Lemma} \( \texttt{peano8} \) : \( \forall \, m \, n : \text{nat}, S \, m = S \, n \rightarrow m = n \).
intros \( m \, n \, h \).

By folding with \( \texttt{pred} \) we can change the current goal so that we can apply our hypothesis.

\( \text{fold} \, (\text{pred} \, (S \, m)) \).
\( \text{rewrite} \, h \).

And now we just have to unfold. \texttt{simpl} would have done the job too.

\( \text{unfold} \, \texttt{pred} \).
\( \text{reflexivity} \).
Qed.

The 8th axiom says that the successor function is injective. Can we prove the other direction too? \( \forall \, m \, n : \text{nat}, m = n \rightarrow S \, m = S \, n \) 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 \texttt{induction}...

\textbf{Lemma} \( \texttt{peano9} \) : \( \forall \, P : \text{nat} \rightarrow \text{Prop}, P \, 0 \rightarrow (\forall \, m : \text{nat}, P \, m \rightarrow P \, (S \, m)) \rightarrow \forall \, n : \text{nat}, P \, n \).
intros \( P \, h0 \, hS \, n \).
\( \text{induction} \, n \).
exact \( h0 \).
apply \( hS \).
exact \( \text{IHn} \).
Qed.
6.2 Addition and multiplication

Peano defined the operations addition and multiplication. These are actually examples of functions defined by primitive recursion a general scheme which can be used to define many other functions. A function is definable by primitive recursion if we can give a case for 0 and reduce the computation for the value at \( S \ n \) to the value at \( n \). In Coq we have to use the keyword `fixpoint` instead of `definition` and we have to indicate on which argument we want to do 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.

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

```coq
Eval compute in (add 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.

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

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

In the Coq library addition is defined using the usual infix notation \(+\) and \(×\) 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 `add` and `mult`
6.3 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.
- $(\times,1)$ is a commutative monoid, because 1 is neutral, $\times$ is associative and commutative.
- $(+,0,\times,1)$ is a **commutative semiring** because $(+,0)$ and $(\times,1)$ 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_O n : ∀ n:nat, n = 0 + n.`

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

`reflexivity.`

Qed.

**Lemma** `plus_n_O : ∀ 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 $+$

34
Lemma \( \text{plus\_assoc} : \forall (l m n:\text{nat}), l + (m + n) = (l + m) + n. \)

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 \( \text{IHl}. \)

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 \( \text{plus\_n\_Sm} : \forall n m : \text{nat}, S (m + n) = m + S n. \)

intros.

induction \( m. \)

simpl.

reflexivity.

simpl.

rewrite \( \text{IHm}. \)

reflexivity.

Qed.

We are now ready to prove commutativity.

Lemma \( \text{plus\_comm} : \forall n m : \text{nat}, n + m = m + n. \)

intros.

induction \( n. \)

simpl.

apply \( \text{plus\_n\_O}. \)

simpl.

rewrite \( \text{IHn}. \)

apply \( \text{plus\_n\_Sm}. \)

Qed.
6.4 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** \textit{leq} $(m \: n : \text{nat}) : \text{Prop} :=$

$\exists \: k : \text{nat}, \: n = k + m.$

**Notation** "$m \leq n$" := $(\text{leq} \: m \: n)$.

We verify some basic properties of $\leq$:

- $\leq$ is reflexive. $\forall \: n : \text{nat}, \: n \leq n$

- $\leq$ is transitive. $\forall \: l \: m \: n : \text{nat}, \: l \leq m \implies m \leq n \implies l \leq n$

- $\leq$ is antisymmetric. $\forall \: l \: m : \text{nat}, \: l \leq m \implies m \leq l \implies m = l$

Any relation which is reflexive, transitive and antisymmetric is a \textit{partial order}. Here the word \textit{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** \textit{le_refl}: $\forall \: n : \text{nat}, n \leq n$.

**intro n.**

$\exists \: 0.$

**reflexivity.**

**Qed.**

**Lemma** \textit{le_trans} : $\forall \: (l \: m \: n : \text{nat}), \: l \leq m \implies m \leq n \implies l \leq n$.

**intros l m n lm mn.**

**destruct lm as [k klm].**

**destruct mn as [j jmn].**

$\exists \: (j+k).$

**rewrite← plus_assoc.**

**rewrite← klm.**

**rewrite← jmn.**

**reflexivity.**

**Qed.**

6.5 Decidable properties

We say a predicate is $P : A \to \text{Prop}$ \textit{decidable} if we can define a boolean function $\text{decP} : A \to \text{bool}$ which agrees with the predicate, i.e. $\forall \: a : A, \: P \: a \leftrightarrow \text{decP} \: a = \text{true}$. 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 \( (m \ n : \text{nat}) \{\text{struct} \ m \} : \text{bool} \) :=
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 \( \rightarrow \) direction just boils down to showing that eqnat is reflexive. Why?

Lemma eqnat_refl : \( \forall \ m : \text{nat}, \text{eqnat} \ m \ m = \text{true} \).
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 : \( \forall \ m \ n : \text{nat}, \text{eqnat} \ m \ n = \text{true} \rightarrow m = n \).
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.
\( n = \text{S} \ n' \) intro \( h \).
simpl in \( h \).
discriminate \( h \).
\( m = \text{S} \ m' \) intro \( n \).
induction \( n \).
\( n = 0 \) intro \( h \).
discriminate \( h \).
\( n = \text{S} \ n' \) intro \( h \).
assert \( (h' : m = n) \).
apply IHm.
extact h.
rewrite h'.
reflexivity.
Qed.

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

**Theorem** \( \text{eqnat}\_\text{dec} : \forall m \, n : \text{nat}, \, m = n \leftrightarrow \text{eqnat} \, m \, n = \text{true} \).  

intros \( m \) \( n \). 
split. 
intro \( h \). 
rewrite \( h \). 
apply \( \text{eqnat}\_\text{refl} \). 
apply \( \text{eqnat}\_\text{compl} \). 
Qed.

End \textit{Arith}.
Chapter 7

Lists

Section Lists.

Lists are the ubiquitous data structure 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 $[1,\text{true}]$ 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 front 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 $(A : \text{Set}) : \text{Set} :=$
| nil : list $A$
| cons : $A \to \text{list } A \to \text{list } A$.

Implicit Arguments nil [A].

In functional programming cons is usually written as an infix operation. In Haskell this
is $\cdot$ 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 \( l_{23} \): list nat

:= 2 :: 3 :: nil.

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

Definition \( l_{123} \): list nat

:= 1 :: \( l_{23} \).

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

**Theorem** \( \text{nil} \_ \text{cons} \): \( \forall (A:\text{Set})(x:A) (l:\text{list } A), \) \( \text{nil} \neq x :: l. \)

*intros.*

*discriminate.*

*Qed.*

The next peano axiom \( \text{peano8} \) 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 \( \text{tail} \) as we had define predecessor for numbers.

**Definition** \( \text{tail} \):(A:\text{Set})(l:\text{list } A) : \text{list } A :=

\[
\text{match } l \text{ with}
| \text{nil } \Rightarrow \text{nil}
| \text{cons } a \ l \Rightarrow l
\text{end}.
\]

The proof follows exactly the one for \( \text{peano8} \).

**Theorem** \( \text{cons} \_ \text{injective} \):

\( \forall (A : \text{Set})(a \ b : A)(l \ m : \text{list } A), \)

\( a :: l = b :: m \rightarrow l = m. \)

*intros* \( A \ a \ b \ l \ m \ h.\)

*fold* (\( \text{tail} \ (\text{cons } a \ l) \)).

*rewrite* \( h.\)

*unfold* \( \text{tail}.\)

*reflexivity.*

*Qed.*

However, unlike \( S \), \( \text{cons} \) 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 there head is equal.

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

To overcome this issue we define \( \text{head} : A \rightarrow \text{list } A \rightarrow A \) where the first argument is a dummy argument which is returned for the empty list.

**Definition** \( \text{head} \):(A : \text{Set})(x : A)(l : \text{list } A) : A :=

\[
\text{match } l \text{ with}
| \text{nil } \Rightarrow x
| \text{cons } a \ l \Rightarrow a
\text{end}.
\]
Once we have defined \textit{head} the proof of injectivity is rather straightforward.

\textbf{Theorem cons\textsubscript{injective}' :}

\begin{align*}
\forall (A : \text{Set})(a b : A)(l m : \text{list } A), \\
& a :: l = b :: m \rightarrow a = b.
\end{align*}

\textit{intros} \ A \ a \ b \ l \ m \ h.
\textit{fold} (\textit{head} a (a :: l)).
\textit{rewrite} h.
\textit{unfold} head.
\textit{reflexivity}.
\textit{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 \textit{cons} \(a\ \ l\) for any \(a\), then it holds for all lists. In Coq we use the same tactic \textit{induction} to perform list induction.

\textbf{Theorem ind\textsubscript{list} : \forall (A : \text{Set})(P : \text{list } A \rightarrow \text{Prop}),}

\begin{align*}
P \text{ nil} \\
& \rightarrow (\forall (a:A)(l : \text{list } A), P \ l \rightarrow P \ (a :: l)) \\
& \rightarrow \forall \ l : \text{list } A, P \ l.
\end{align*}

\textit{intros} \ A \ P \ hnil \ hcons \ l.
\textit{induction} \ l.
\textit{exact} \ hnil.
\textit{apply} \ hcons.
\textit{exact} \ IHl.
\textit{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 \textit{app} by \textit{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.

\textbf{Fixpoint app} \ (A : \text{Set})(l m : \text{list } A) : \text{list } A :=

\begin{align*}
& \begin{array}{l}
| \text{nil} \Rightarrow m \\
| a :: l' \Rightarrow a :: (\text{app} \ l' \ m)
\end{array}
\end{align*}

end.
As in Haskell we use the infix 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]`.

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),
  
  \( \text{nil} ++ l = l \).

**intros A l.**

**reflexivity.**

Qed.

**Theorem app-l-nil** : ∀ (A : Set)(l : list A),
  
  \( l ++ \text{nil} = l \).

**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 \((list A, ++, \text{nil})\) is an example of a non-commutative monoid. Since we 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 \( \text{snoc} \) just by using ++. Can you see how?

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

Eval compute in \((\text{snoc} \ l 23 \ 1)\).

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

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

This definition of \( \text{rev} \) is called \textit{naive reverse} and it is rather inefficient. Can you see why?

How can it be improved?

Some examples.

Eval compute in \( \text{rev} \ l 123 \).

Eval compute in \( \text{rev} \ (\text{rev} \ l 123) \).

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

To prove it we first prove a lemma about \( \text{rev} \) and \( \text{snoc} \). How did we discover this lemma?

Lemma revsnoc : \( \forall \ (A:\text{Set})(l:\text{list} \ A)(a : A), \text{rev} \ (\text{snoc} \ l \ a) = a :: (\text{rev} \ l) \).
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:
\[ \forall (A: \text{Set})(l: \text{list } A), \text{rev (rev } l) = l. \]
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
\[ 4 :: 2 :: 3 :: 1 :: \text{nil} \]
should be sorted into
\[ 1 :: 2 :: 3 :: 4 :: \text{nil} \]

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

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

Eval compute in leqb 3 4.
Eval compute in leqb 4 3.
Notation "m \leq n" := (leq m n).

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

Axiom leq1 : \(\forall m n : \text{nat}, \text{leqb } m n = \text{true} \rightarrow m \leq n. \)
Axiom leq2 : \(\forall m n : \text{nat}, m \leq n \rightarrow \text{leqb } m n = \text{true}. \)

The main function of insertion sort is the function insert which inserts a new element into an already sorted list:
Fixpoint \textit{insert} \((n: \text{nat})(ms : \text{list nat}) \{\text{struct} ms\} : \text{list nat} :=
\begin{align*}
\text{match} \ ms \ \text{with} \\
| \ nil &\Rightarrow n::nil \\
| m::ms' &\Rightarrow \text{if leqb} \ n \ m \\
&\quad\text{then} n::ms \\
&\quad\text{else} m::(\text{insert} \ n \ ms') \\
\end{align*}
\end{align*}.

Eval compute in \textit{insert} 3 (1::2::4::nil).

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

Fixpoint \textit{sort} \((ms : \text{list nat}) : \text{list nat} :=
\begin{align*}
\text{match} \ ms \ \text{with} \\
| \ nil &\Rightarrow \nil \\
| m::ms' &\Rightarrow \text{insert} \ m \ (\text{sort} \ ms') \\
\end{align*}
\end{align*}.

Eval compute in \textit{sort} (4::2::3::1::nil).

Fixpoint \textit{Sorted} \((l : \text{list nat}) : \text{Prop} :=
\begin{align*}
\text{match} \ l \ \text{with} \\
| \ nil &\Rightarrow \text{True} \\
| \ a :: m &\Rightarrow \text{Sorted} \ m \land a \leq \text{head} \ a \ m \\
\end{align*}
\end{align*}.

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

Axiom total : \(\forall m n : \text{nat}, m \leq n \lor n \leq m\).

Our goal is to show that \textit{insert} preserves sortedness, i.e. \(\text{Sorted} \ l \to \text{Sorted} \ (\text{insert} \ n \ l)\).
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 \textit{leqFalse} : \(\forall m n : \text{nat}, \text{leqb} \ m \ n = \text{false} \rightarrow n \leq m\).
intrs \(m n h\).
destruct (total \(m n\)) as \([mn \mid nm]\).
assert \(mnt : \text{leqb} \ m \ n = \text{true}\).
apply leq2.
extact \(mn\).
rewrite \(h\) in \(mnt\).
discriminate \(mnt\).
extact \(nm\).
Qed.

The other lemma is a little case analysis: the head of the result of \textit{insert} is either the inserted element or the previous head.
Lemma `insertSortCase : \forall (n a : nat)(l : list nat),
head a (insert n l) = n \lor 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 : \forall (n : nat)(l : list nat),
Sorted l \rightarrow 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.
simpl.
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.
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 \( sortSorted : \forall\ ms:list\ nat,Sorted\ (sort\ ms)\).

induction \( ms \).

    case ms=nil:
        simpl.
split.

    case a::ms
        simpl.
        apply \( insertSorted \).
        exact \( IHms \).
Qed.

47
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

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:
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

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 runAux (st:Stack)(p:Code) {struct p} : nat := match p with
  | nil ⇒ match st with
    | nil ⇒ 0
    | n :: st' ⇒ n
    end
  | op :: p' ⇒
    match op with
    | Push n ⇒ runAux (n :: st) p'
    | PlusC ⇒ match st with
      | nil ⇒ 0
      | n :: nil ⇒ 0
      | n1 :: n2 :: st' ⇒
        runAux ((n1 + n2) :: 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 c1).

A simple compiler

We implement a simple compiler which translates an expression into code for the stack machine.
A naive implementation looks like this:

```ocaml
Fixpoint compile naive (e:Expr) \{ struct e \} : list Op :=
match e with
| Const n => (Push n) :: nil
| Plus e1 e2 => (compile naive e2)++
  (compile naive e1)++
  (PlusC::nil)
end.
```

Why do we need to do this in this order?

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.

```ocaml
Fixpoint compileAux (e:Expr) (p:Code) \{ struct e \} : Code := match e with
| Const n => Push n :: p
| Plus e1 e2 => compileAux e2
  (compileAux e1 (PlusC :: p))
end.
```

The top level compiler just uses the empty continuation.

```ocaml
Definition compile (e:Expr) : Code := compileAux e nil.
```

Test the compiler

```ocaml
Eval compute in compile e1.
```

And run the compiled code:

```ocaml
Eval compute in run (compile e1).
```

### 8.3 Compiler correctness

Compiler correctness means that the operational semantics of the compiled code agrees with its denotational semantics.

forall e:Expr, run (compile e) = eval e.

However, to prove this we have to show a more general lemma about the auxiliary functions.

```ocaml
Lemma compileLem : \forall (e:Expr)(p:Code)(st:Stack),
runAux st (compileAux e p) = runAux ((eval e)::st) p.
induction e.
intros p st.
simpl.
reflexivity.
simpl.
```
intros.
rewrite \textit{IH}e_2.
rewrite \textit{IH}e_1.
simpl.
reflexivity.
Qed.

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

\textbf{Theorem } \textit{compileOk} : \forall e:\textit{Expr},
\textit{run} (\textit{compile} \ e) = \textit{eval} \ e.

intro \ e.
unfold \textit{run}.
unfold \textit{compile}.
rewrite \textit{compileLem}.
simpl.
reflexivity.
Qed.

End \textit{Expr}.
Chapter 9

Coq in Coq

Section Meta.

Require Import Coq.Lists.List.

Set Implicit Arguments.

This chapter is about using Coq to reason about its own logic. This was the title of a paper by Bruno Barras who managed to develop the theory of Coq inside Coq.

Obviously, we won’t be able to do this here so we are going to focus on a more modest goal: we are limiting ourselves to propositional logic and to keep things short we will look at propositional logic with implication only.

We are going to develop this logic inside coq using natural deduction. This is very close to the actual Coq proof objects.

An alternative would be to use combinatory logic. We are going to compare these two approaches and we will show that they are equivalent.

9.1 Formulas as trees

We are representing logical formulas as trees. Variables are just represented as strings.

Inductive Formula : Set :=
| Var : string → Formula
| Impl : Formula → Formula → Formula.

Notation "x ==> y" := (Impl x y) (at level 30, right associativity).

As examples we are going to use three propositions, all of them are tautologies. Two of them will show up as the basic combinators of combinatoric logic later.

The identity combinator "I".

Definition I (P : Formula) : Formula := P ==> P.
The constant combinator "K".

**Definition** $K (P, Q : \text{Formula}) : \text{Formula} := P \implies Q \implies P$.

The (mysterious) combinator "S".

**Definition** $S (P, Q, R : \text{Formula}) : \text{Formula} :=$

- $(P \implies Q \implies R)$
- $\implies (P \implies Q)$
- $\implies P \implies R$.

We represent Hypotheses as a list of formula.

**Definition** $\text{Hyps} : \text{Set} := \text{list Formula}$.

### 9.2 Natural Deduction

We are going to represent the proposition that a formula $P$ is provable from a list of assumptions $Hs$ as $\text{ND}_\text{Proof} \ Hs \ P$. This is an inductive definition, the constructors are nodes in the proof tree.

**Inductive** $\text{ND}_\text{Proof} : \text{Hyps} \to \text{Formula} \to \text{Prop} :=$

The first constructor $\text{nd}\_\text{ass}$ allows us to use the last hypothesis from our list of hypotheses (which appears at the head of the list).

- $\text{nd}\_\text{ass} : \forall (Hs : \text{Hyps}) (P : \text{Formula}),$
  - $\text{ND}_\text{Proof} \ (P :: Hs) P$

To be able to access earlier hypothesis we use $\text{nd}\_\text{weak}$ which allows us to ignore the last hypothesis (i.e. the head of the list).

- $\text{nd}\_\text{weak} : \forall (Hs : \text{Hyps}) (P, Q : \text{Formula}),$
  - $\text{ND}_\text{Proof} \ Hs \ P \to \text{ND}_\text{Proof} \ (Q :: Hs) P$

The next constructor $\text{nd}\_\text{intro}$ corresponds to the intro tactic in Coq: to prove $P \implies Q$ we assume $P$, i.e. we add it to the list of assumptions, and continue to prove $Q$.

- $\text{nd}\_\text{intro} : \forall (Hs : \text{Hyps}) (P, Q : \text{Formula}),$
  - $\text{ND}_\text{Proof} \ (P :: Hs) Q \to \text{ND}_\text{Proof} \ Hs \ (P \implies Q)$

The elimination for application is slightly different from the one in Coq which is hard to state precisely. The rule $\text{nd}\_\text{apply}$ corresponds to *modus ponens*: if you can prove $P \implies Q$ and also $P$ then you can also prove $Q$.

- $\text{nd}\_\text{apply} : \forall (Hs : \text{Hyps}) (P, Q : \text{Formula}),$
  - $\text{ND}_\text{Proof} \ Hs \ (P \implies Q) \to \text{ND}_\text{Proof} \ Hs \ P \to \text{ND}_\text{Proof} \ Hs \ Q$.

As examples we are going to prove that the examples $I, K$ and $S$ are provable.

The proof for $I$ follows almost exactly the proof of the same tautology in Coq.

**Lemma** $\text{nd}\_I : \forall (Hs : \text{Hyps}) (P : \text{Formula}),$

- $\text{ND}_\text{Proof} \ Hs \ (I \ P)$.

intros $Hs\ P$. unfold $I$.  

54
To prove $K$ we need to use $\text{weak}$.

**Lemma** $\text{nd}_K : \forall (Hs : \text{Hyps})(P \ Q : \text{Formula}),$

$$\text{ND}_\text{Proof} \ Hs \ (K \ P \ Q).$$

intros $Hs \ P \ Q$.
unfold $K$.
apply $\text{nd}_\text{intro}$.
apply $\text{nd}_\text{intro}$.
apply $\text{nd}_\text{weak}$.
apply $\text{nd}_\text{ass}$.
Qed.

The proof of $S$ uses $\text{nd}_\text{apply}$. It also shows that modens ponens isn’t so suitable to interactive proof, because we need some hindsight how to apply it.

**Lemma** $\text{nd}_S : \forall (Hs : \text{Hyps})(P \ Q \ R : \text{Formula}),$

$$\text{ND}_\text{Proof} \ Hs \ (S \ P \ Q \ R).$$

intros $Hs \ P \ Q \ R$.
unfold $S$.
apply $\text{nd}_\text{intro}$.
apply $\text{nd}_\text{intro}$.
eapply $\text{nd}_\text{apply}$. eapply $\text{nd}_\text{apply}$.
apply $\text{nd}_\text{weak}$. apply $\text{nd}_\text{weak}$. apply $\text{nd}_\text{ass}$.
apply $\text{nd}_\text{ass}$.
eapply $\text{nd}_\text{apply}$.
apply $\text{nd}_\text{weak}$. apply $\text{nd}_\text{ass}$.
apply $\text{nd}_\text{ass}$.
Qed.

### 9.3 Combinatory logic.

Combinatory logic (also sometimes called "Hilbert style logic") is based on the maybe surprising observation that we can replace $\text{nd}_\text{intro}$ by adding $K$ and $S$ as axioms. This leads to a variable free representation of logic. However, to be able to compare natural deduction and combinatory logic we will consider combinatory logic with variables here. However, if the list of hypotheses is empty we will never need variables unlike natural deduction where the $\text{nd}_\text{intro}$ rule introduces variables.

We define $\text{CL}_\text{Proof} \ Hs \ P$ to mean that $P$ is provable from $Hs$ in combinatory logic.

**Inductive** $\text{CL}_\text{Proof} : \text{Hyps} \rightarrow \text{Formula} \rightarrow \text{Prop} :=$

55
The rules relating to hypothesis are exactly the same as the ones for natural deduction.

\[
\begin{align*}
| \text{cl\_ass} & : \forall (Hs : Hyps)(P : Formula), \quad CL\_Proof (P :: Hs) P \\
| \text{cl\_weak} & : \forall (Hs : Hyps)(P Q : Formula), \quad CL\_Proof Hs P \rightarrow CL\_Proof (Q :: Hs) P
\end{align*}
\]

We are adding proofs for K and S as axioms.

\[
\begin{align*}
| \text{cl\_K} & : \forall (Hs : Hyps)(P Q : Formula), \quad CL\_Proof Hs (K P Q) \\
| \text{cl\_S} & : \forall (Hs : Hyps)(P Q R : Formula), \quad CL\_Proof Hs (S P Q R)
\end{align*}
\]

Modus ponens \text{cl\_apply} is the same rule as for natural deduction.

\[
\begin{align*}
| \text{cl\_apply} & : \forall (Hs : Hyps)(P Q : Formula), \quad CL\_Proof Hs (P \rightarrow Q) \rightarrow CL\_Proof Hs P \rightarrow CL\_Proof Hs Q.
\end{align*}
\]

We can actually prove \(I\) from \(S\) and \(K\).

**Lemma** \text{cl\_I} : \(\forall (Hs : Hyps)(P : Formula), \quad CL\_Proof Hs (I P).

\begin{verbatim}
intros Hs P.
unfold I.
eapply cl\_apply.
eapply cl\_apply.
apply cl\_S.
apply cl\_K.
\end{verbatim}

We need to instantiate one of the meta variables by hand. This is how we do this in Coq - please check the manual.

\begin{verbatim}
instantiate (1:= P \rightarrow P ).
apply cl\_K.
Qed.
\end{verbatim}

Since we did already prove \(K\) and \(S\) using natural deduction, we can show that every proof in combinatory logic can be turned into one in natural deduction. We prove this by induction over the derivation trees.

Basically we are showing that each node in an CL proof tree can be replaced by a ND tree by replacing the axioms \(K\) and \(S\) by the corresponding proofs.

**Lemma** \text{cl\_2nd} : \(\forall (Hs : Hyps)(P : Formula), \quad CL\_Proof Hs P \rightarrow ND\_Proof Hs P.

\begin{verbatim}
intros Hs P H.
\end{verbatim}

Since the derivation trees are dependent, i.e. they depend on the choice of hypotheses and proposition we need to invoke the tactic \text{dependent induction}.
dependent induction \( H \).

We have now to provide a translation for each case.

\( \text{ass}_{\text{cl}} \) is translated by \( \text{nd}_{\text{ass}} \).

apply \( \text{nd}_{\text{ass}} \).

And \( \text{weak}_{\text{cl}} \) by \( \text{nd}_{\text{weak}} \). Here we have to use the induction hypothesis to recursively translate the rest of the proof.

apply \( \text{nd}_{\text{weak}} \).

exact \( \text{IHCL} \_\text{Proof} \).

\( \text{cl}_{\text{K}} \) is translated as \( \text{nd}_{\text{K}} \). Here on axiom is replaced by a small proof tree.

apply \( \text{nd}_{\text{K}} \).

\( \text{cl}_{\text{S}} \) is translated as \( \text{nd}_{\text{S}} \).

apply \( \text{nd}_{\text{S}} \).

\( \text{cl}_{\text{apply}} \) is translated by \( \text{nd}_{\text{apply}} \). Since there are two subproofs we have to translate them recursively by using the induction hypotheses.

eapply \( \text{nd}_{\text{apply}} \).
apply \( \text{IHCL} \_\text{Proof} \_1 \).
apply \( \text{IHCL} \_\text{Proof} \_2 \).
Qed.

### 9.4 The deduction theorem

The main ingredient to prove the other direction of the equivalence, i.e. that it is possible to simulate natural deduction proofs in combinatory logic, is to show that combinatory logic is closed under the \( \text{intro} \) rule. This is usually called \textit{the deduction theorem}.

**Lemma** \( \text{cl}_{\text{intro}} \) : \( \forall \ (Hs : \text{Hyps})(P \ Q : \text{Formula}), \)

\[ \text{CL} \_\text{Proof} \ (P :: Hs) \ Q \rightarrow \text{CL} \_\text{Proof} \ Hs \ (P \Rightarrow \ Q) \]

intros \( Hs \ P \ Q \ H \).

to prove this we need to perform an induction over the proof tree showing \( \text{CL} \_\text{Proof} \ (P :: Hs) \ Q \).

dependent induction \( H \).

The case for \( \text{cl}_{\text{ass}} \) can be proven using the identity proof \( \text{cl}_{\text{I}} \) which we have already derived.

apply \( \text{cl}_{\text{I}} \).

In the case for \( \text{cl}_{\text{weak}} \) we need to use \( \text{cl}_{\text{K}} \).

eapply \( \text{cl}_{\text{apply}} \).
apply \( \text{cl}_{\text{K}} \).
exact $H$.

The case for $cl\_K$ can be derived by using $cl\_K$ once to ignore the argument and a 2nd time to provide the constant to be actually used.

eapply $cl\_apply$.
apply $cl\_K$.
apply $cl\_K$.

The case for $cl\_S$ is similar only that we use $cl\_S$ the 2nd time.

eapply $cl\_apply$.
apply $cl\_K$.
apply $cl\_S$.

The case for $cl\_app$ is the most interesting one. It finally lifts the mystery about $S$. It is actually what we need to translate this case, i.e. to shift abstraction over an application.

eapply $cl\_apply$.
eapply $cl\_apply$.
apply $cl\_S$.
apply $IHCL\_Proof1$.
reflexivity.
apply $IHCL\_Proof2$.
reflexivity.
Qed.

9.5 Equivalence of natural deduction and combinatory logic.

We have now all the ingredients together to show that natural deduction and combinatory logic prove exactly the same propositions.

To prove the other direction we only need to appeal to the deduction theorem $cl\_intro$ when translating $nd\_intro$.

Lemma $nd2cl : \forall (Hs : Hyps) (P : Formula),
\quad ND\_Proof Hs P \rightarrow CL\_Proof Hs P$.

intros $Hs\ P\ H$.
dependent induction $H$.
apply $cl\_ass$.
apply $cl\_weak$.
exact $IHND\_Proof$.
apply $cl\_intro$. exact $IHND\_Proof$.
eapply $cl\_apply$.
apply $IHND\_Proof1$.
exact $IHND\_Proof2$.
Qed.
The final theorem

**Theorem** \( \text{ndcl} : \forall (\text{Hs} : \text{Hyps})(\text{P} : \text{Formula}), \)
\[
\text{ND} \_ \text{Proof Hs P} \leftrightarrow \text{CL} \_ \text{Proof Hs P}.
\]

intros \text{Hs} \text{P}.
split.
apply \text{nd2cl}.
apply \text{cl2nd}.
Qed.