Discrete Structures

Lecture Notes for CSE 191

Matthew G. Knepley
Feng-Mao Tsai

University at Buffalo
Department of Computer Science and Engineering
University At Buffalo
April 15, 2024

I dedicate these notes to my wonderful wife Margarete, without whose patience and help they could never have been written. –*Matt*

Acknowledgements

TBD

Contents

Chapter 1

Propositional Logic

. . . an incorrect theory, even if it cannot be inhibited by any contradiction that would refute it, is none the less incorrect, just as a criminal policy is none the less criminal even if it cannot be inhibited by any court that would curb it.

— L. E. J. Brouwer

Science is what we understand well enough to explain to a computer, Art is all the rest.

— Donald E. Knuth

1.1 Whirlwind Inroduction to Propositional Calculus

Do you consider proofs mysterious and forbidding? Does logic seem remote from the programming that you enjoy? In course, through the magic of the Curry-Howard isomorphism, we will demonstrate that proving theorems and writing programs are actually two sides of the same coin! We will need just one sophisticated programming notion, namely that functions can be data. We will allow our functions to take in other functions as input and return them as output. And not just functions, we will also allow types themselves to be data.

Now, suppose I asked you to write a function that took in data of some type and returned the exact same thing. You might write something like

```
foo(Type P, P p) {
  return p;
}
```

where we need to allow our function `foo` to return different types depending on the input `tp`. In the notation of the language Coq that we will use for the course, this becomes

```
fun (P : Prop) (p : P) => p
    : forall P : Prop, P -> P
```

We can translate all the parts of this definition. The `fun` keyword defines a function. The `Prop` type means that we have passed in a type as data, what we use `Type` for above. Type declarations use the colon operator, so `p : P` means we have a variable `p` of type `P`. The `=>` operator is the same as `return` in C. So the first line contains our function definition, with the arguments in parentheses, and the second line contains the equivalent statement of propositional calculus, namely that any proposition P implies itself. We have just completed our first proof doing nothing but programming!

Talk about P, P -¿ Q

Talk about P -¿ Q, Q -¿ R, R -¿ S

1.2 Propositions

A *proposition* is a declarative statement. It is not an opinion (normative statement), a question (interogative statement), command (imperative statement), or a parameterized statement with indeterminate truth value. An example of a parameterized statement would be "n^2 is even", for some indeterminate natural number n. Propositional calculus is a set of rules for manipulating propositions so that if we start with valid propositions, we can produce other valid propositions. Why would we use propositional calculus?

- To compose propositions
- So we can get from facts we collect about the world, to conclusions about it or decisions about our behavior, in an *automated* way.
- To verify that computer programs/algorithms produce the correct output for all possible input values.
- To establish the security of systems.

Propositional calculus, or Boolean algebra, was the creation of self-taught mathematician George Boole in his book The Laws of Thought. We will call a statement of the propositional calculus a *Boolean expression*.

Classically, a proposition could be either true or false, which leads to the Law of Excluded Middle, $\P \vee \neg P$. In this book, we take the computational perspective. A proposition is valid sentence of propositional calculus that might be proved.

1.3 Operators and Truth Tables

There are two non-trivial unary Boolean operators. The first, *identity*, just returns its input. The other, *negation*, reverses the truth value of its argument. So, if P is true, then $\neg P$ is false, and vice versa.

P	id$P$
T	T
F	F

P	$\neg P$
T	F
F	T

There must be four unary operators in total, but the other two are constant, always returning true or always returning false.

Implication is a binary Boolean operator, $P \implies Q$, which encasulates the common verbal form "if P, then Q". The first part P is called the assumption, or *antecedent*, and the second part is called the conclusion, or *consequent*. An implication is only false if a true antecedent leads to a false consequent, say "If 2 is even then 2^2 is odd". We can illustrate this using a *truth table*, which is a device that assigns truth values to a combination of propositions based on the truth values of the propositions themselves. For implication,

P	Q	$P \implies Q$
T	T	T
T	F	F
F	T	T
F	F	T

Because implication statements play such an essential role in mathematics, a variety of terminology is used to express $p \implies q$,

- if p, then q
- q, if p
- p, only if q
- p implies q
- p is sufficient for q
- q is necessary for p
- q follows from p

A common argument technique involving implication is called *contraposition* or employing the *contrapositive*. An argument using the contrapositive replaces an implication

$$p \implies q$$

with its contraposition

$$\neg q \implies \neg p$$

since in classical logic, the two forms are equivalent. We can see this using a truth table.

P	Q	$P \implies Q$	$\neg Q \implies \neg P$
T	T	T	T
T	F	F	F
F	T	T	T
F	F	T	T

However, they are not equivalent in constructive logic since it relies on the Law of Excluded Middle.

It turns out that $\{\implies, \neg\}$ constitute a complete logical system, in the sense that we can construct any Boolean sentence using only these operators. But what does that mean? It must mean that we can generate any given truth table from a set of inputs $P_1, \ldots, P_n$. Emil Post characterized functional completeness of logical operators. The NAND and NOR operators (see Problem 5) are the only functionally complete singletons. The sets $\{\wedge, \neg\}$, $\{\vee, \neg\}$, and $\{\implies, \neg\}$ are all complete, along with 18 more.

Using the same reasoning as above, we can conclude that there are 16 possible binary logical operators, since there are 4 possible input combinations for each one, and two possible responses for each input combination, $2^4 = 16$. They are not strictly necessary, but we define a few more logical operators below for convenience in writing Boolean expressions. Conjuction, $P \wedge Q$, formalizes the verbal construction “P and Q are true”,

P	Q	$P \wedge Q$
T	T	T
T	F	F
F	T	F
F	F	F

so that it is only true when both arguments are true. A kind of inverse of this is disjunction, $P \vee Q$, from the verbal form “P or Q is true”,

P	Q	$P \vee Q$
T	T	T
T	F	T
F	T	T
F	F	F

which is true if either of the arguments is true. We can formalize this idea of an *inverse*, by saying that the inverse negates all the arguments, so

$$\neg P \wedge \neg Q = \neg(P \vee Q) \tag{1.1}$$

which is one of Augustus De Morgan’s Laws, the other being

$$\neg P \vee \neg Q = \neg(P \wedge Q) \tag{1.2}$$

which can also be verified using the truth tables. The exclusive-or operator, $P \oplus Q$, also called XOR, is like logical *or* except that it is false when both arguments are true, which is the exclusion.

P	Q	$P \oplus Q$
T	T	F
T	F	T
F	T	T
F	F	F

Another way of seeing this, is that the operator is true when the arguments are different. We could imagine an operator that was true when both arguments were the same. This happens when we have implication in both directions, namely that $P \implies Q$ and $Q \implies P$, which we will call logical equivalence or bidirectional implication.

P	Q	$P \iff Q$
T	T	T
T	F	F
F	T	F
F	F	T

We can also get this using our inverse operation

$$\neg P \oplus \neg Q = \neg(P \iff Q) \tag{1.3}$$

which we can verify with a truth table

P	Q	$\neg P \oplus \neg Q$	$\neg(P \iff Q)$
T	T	F	F
T	F	T	T
F	T	T	T
F	F	F	F

In this way, truth tables give us a way to interpret the meaning of any compound logical expression. First, we substitute a truth value for each atomic proposition. Then for each operator, we replace it with the truth value from its table. How many truth values do we need for an operator or expression? If it has n inputs, then we will need 2^n truth values, one for each combination of input truth values. And it is here that we arrive at the true goal of propositional calculus, that is to automate deduction and the use of logic to make decisions. Be very suspicious of any activity for which you must be smart. We instead want automation, which allows dumb things, such as computers, to perform flawlessly. There is a fly in the ointment above, since we see that using truth tables in a brute force manner will require an exponential table size. Later on we will look at another way to formalize proofs in propositional calculus.

An underlying assumption in this use of truth tables is the Law of Excluded Middle which says that either P or its negation $\neg P$ *must* be true. We can verify this with a truth table.

P	$\neg P$	$P \vee \neg P$
T	F	T
T	F	T
F	T	T
F	T	T

Logic admitting this proof will be called classical logic, as opposed to intuitionistic logic in which we cannot prove $P \vee \neg P$ or $((P \implies Q) \implies P) \implies P$ (Pierce's Law).

An example from number theory of an argument that depends on the law of excluded middle would be to prove that there exist two irrational numbers a and b such that a^b is rational. It is known that $\sqrt{2}$ is irrational (we will prove this later in the course). Consider the number

$$\sqrt{2}^{\sqrt{2}} \tag{1.4}$$

By the law of excluded middle, this number is either rational or irrational. If it is rational, the proof is complete, giving

$$a = \sqrt{2} \qquad \text{and} \qquad b = \sqrt{2}. \tag{1.5}$$

However, if $\sqrt{2}^{\sqrt{2}}$ is irrational, then let

$$a = \sqrt{2}^{\sqrt{2}} \qquad \text{and} \qquad b = \sqrt{2}, \tag{1.6}$$

so that

$$a^b = \left(\sqrt{2}^{\sqrt{2}}\right)^{\sqrt{2}} = \sqrt{2}^2 = 2, \tag{1.7}$$

and 2 is certainly rational, which concludes the proof. The above proof is non-constructive because it doesn't give specific numbers a and b that satisfy the theorem but only two separate possibilities, one of which must work. An *intuitionist* mathematician would require a explicit proof that a was irrational instead. Actually $\sqrt{2}^{\sqrt{2}}$ is irrational, but there is no known easy proof of this fact.

This may seem like a silly objection (a chair is either red, or its not red), but it becomes more serious when we talk about infinite objects or infinite processes. In this case, an intuitionist would be object that such an infinite object cannot be created or the process cannot be finished, and thus I am not justified in assuming that it is. For example, suppose that we are talking about a program. I could define the proposition $P =$ (the program halts), and say the program either halts or it does not halt. However, we have no way of verifying this. We cannot wait long enough to see if the program halts. In fact, this was the subject of one of Kolmogorov's earliest papers (Kolmogorov 1925). In Coq, those propositions for which the Law of Excluded Middle holds are called decidable propositions.

The split between classical and intuitionistic logic also arises when we try to automate the process of proving theorems. One of the most popular strategies for proving theorems of propositional calculus makes use of the Curry-Howard Correspondence, which states that a proof in intuitionistic logic is equivalent to a properly typed program of a certain language, an approach pioneered by Heyting.

1.4 Proving Logical Equivalence

A *tautology* is a statement which is always true. We have already seen an example of this, $P \vee \neg P$. If two propositional statements s_1 and s_2 have the same truth values for all inputs, then $s_1 \iff s_2$ is a tautology since bidirectional implication is true when its arguments are the same. We can show this using a truth table, just as we did above. For example, let us validate one of De Morgan's Laws,

P	Q	$\neg(P \wedge Q)$	$\neg P \vee \neg Q$	$\neg(P \wedge Q) \iff \neg P \vee \neg Q$
T	T	F	F	T
T	F	T	T	T
F	T	T	T	T
F	F	T	T	T

1.5 Using Coq and Writing Proofs by Hand

Coq is a *proof assistant*, meaning a program which helps a user construct and check proofs. In some instances, it may even carry out the entire proof automatically. Other interactive theorem provers exist, such as HOL and Isabelle. I have chosen to use Coq because it is full-featured and well supported, as shown in the comparison of proof assistants here. Coq has an excellent reference manual, as well as companion book, and its library of proofs is available online at https://coq.inria.fr/library.

Installation instructions for Coq can be found at here. We will use it for the entire course, in order to guarantee correctness, but all steps can be done equally well by hand. Let us begin with a very simple proof, namely that $P \implies P$ is a tautology. First, we enter the proof assistant,

```
/home/knepley> coqtop
Welcome to Coq 8.6 (December 2017)

Coq <
```

Most presentations of logic apply inference rules to the hypotheses to generate conclusions. We will also do this, but typically using a reduced set of rules that correspond to type inference. This makes it easier to automate, but we could of course prove all the inference rules shown in traditional texts and use them in our proof assistant. One of the most common inference rules is known as *modus ponens* and also sometimes called implication elimination. It says that given a proof of P and a proof of $P \implies Q$, we can construct a proof of Q. In Coq, modus ponens corresponds to function evaluation, and is effected with the `apply` tactic, or proof operation, which eliminates a hypothesis. If instead the implication $P \implies Q$ is our goal, we can introduce a hypothesis P and make our goal Q. This makes sense because the content of $P \implies Q$, according to Heyting, is that given a proof of P (hypothesis) we can generate a proof of Q.

For our example, $P \implies P$, we first declare the proof, using the `Lemma` or `Theorem` statement, id_P

```
Coq < Lemma id_P : forall P : Prop, P->P.
1 subgoal

  ============================
  forall P : Prop, P -> P
```

The `forall` part, called universal quantification, will be discussed in Chapter 2. However, for now we will understand it as giving the type of our argument P, which is a proposition of type `Prop`. We put this type declaration in the hypothesis set using the `intro` tactic, which introduces a new hypothesis.

```
id_P < intro P.
1 subgoal

  P : Prop
  ============================
  P -> P
```

In the same way, we can introduce the antecedent as a hypothesis

```
id_P < intro p.
1 subgoal

  P : Prop
  p : P
  ============================
  P
```

and then note that our goal is actually one of our hypotheses, named `p`.

```
id_P < exact p.
No more subgoals.

id_P < Qed.
intro P.
intro p.
exact p.

Qed.
id_P is defined
```

You can get the IDE to print out a summary of your proof using `Show Script`. or the Debug menu in the ProofWeb interface. After that, you use `Qed` so that Coq will print out a summary of the proof, check it, and register the lemma internally. We could have done this proof using a truth table,

P	$P \implies P$
T	T
F	T

or more traditional inference rules.

$$P \implies P \equiv \neg P \vee P \qquad \text{(Conditional Identity)} \qquad (1.8)$$
$$\equiv T \qquad \text{(Complement Law)} \qquad (1.9)$$

However, using a proof assistant, we are guaranteed that our conclusion is correct and it removes the sometimes confusing choice of law to apply.

We can equivalently write this proof by hand, mirroring the steps we use in the proof assistant. We write the each step, the application of a tactic, in a list on the right. We write the hypotheses, indicated by H, above the goals indicated by G, separated by a line. We use the step number to number the goals, and we number the hypotheses sequentially.

$H : P$

$G0 : P \implies P$

$G1 : P$

Step	Tactic
1	intro H
2	assumption

Lets look at a slightly more advanced example, in which we use modus ponens, proving that $(P \implies Q) \implies (Q \implies R) \implies P \implies R$. We begin by generating three hypotheses, the antecedents of our implications, and also making the three type declarations into hypotheses, imp_trans

```
Coq < Lemma imp_trans : forall P Q R : Prop, (P->Q)->(Q->R)->P->R.
1 subgoal

  ============================
  forall P Q R : Prop, (P -> Q) -> (Q -> R) -> P -> R

imp_trans < intros P Q R.
1 subgoal

  P, Q, R : Prop
  ============================
  (P -> Q) -> (Q -> R) -> P -> R

imp_trans < intros pimpq qimpr p.
1 subgoal

  P, Q, R : Prop
  pimpq : P -> Q
  qimpr : Q -> R
  p : P
  ============================
  R
```

Now, we apply hypothesis `qimpr`. This hypothesis says that if we can prove Q, then we can prove R. So if R is our current goal, we only really need to prove Q, so that is our new goal.

```
imp_trans < apply qimpr.
1 subgoal

  P, Q, R : Prop
  pimpq : P -> Q
  qimpr : Q -> R
  p : P
  ============================
  Q
```

We can do the same thing again using hypothesis `pimpq`.

```
imp_trans < apply pimpq.
1 subgoal

  P, Q, R : Prop
  pimpq : P -> Q
  qimpr : Q -> R
  p : P
  ============================
  P
```

Now it is a simple matter to say that our goal is one of our assumptions, and the proof is complete.

```
imp_trans < exact p.
No more subgoals.

imp_trans < Qed.
(intros P Q R).
(intros pimpq qimpr p).
(apply qimpr).
(apply pimpq).
exact p.

Qed.
imp_trans is defined
```

We write this proof by hand below.

$$\begin{aligned} H &: P \implies Q \\ H2 &: Q \implies R \\ p &: P \end{aligned}$$

$$\begin{aligned} G0 &: (P \implies Q) \implies (Q \implies R) \implies P \implies R \\ G1 &: (Q \implies R) \implies P \implies R \\ G2 &: P \implies R \\ G3 &: R \\ G4 &: Q \\ G5 &: P \end{aligned}$$

Step	Tactic
1	intros H
2	intros H2
3	intros p
4	apply H2
5	apply H
6	assumption

We have similar tactics for the other logical operators. If my goal is to prove $P \wedge Q$, then I need a proof for P and a proof for Q. The `split` introduction tactic in Coq generates these two subgoals. On the other hand, if we have a hypothesis $P \wedge Q$, then we can turn this into two hypotheses P and Q, which is done by the `destruct` elimination tactic in Coq. For example, and_comm

```
Coq < Lemma and_comm : forall P Q : Prop, (P /\ Q) -> (Q /\ P).
1 subgoal

  ============================
  forall P Q : Prop, P /\ Q -> Q /\ P

and_comm < intros P Q.
1 subgoal

  P, Q : Prop
  ============================
  P /\ Q -> Q /\ P

and_comm < intro pq.
1 subgoal

  P, Q : Prop
  pq : P /\ Q
  ============================
  Q /\ P

and_comm < destruct pq as [p q].
1 subgoal

  P, Q : Prop
  p : P
  q : Q
  ============================
  Q /\ P

and_comm < split.
2 subgoals

  P, Q : Prop
  p : P
  q : Q
  ============================
  Q

subgoal 2 is:
 P

and_comm < exact q.
1 subgoal

  P, Q : Prop
  p : P
  q : Q
  ============================
  P

and_comm < exact p.
No more subgoals.

and_comm < Qed.
(intros P Q).
intro pq.
(destruct pq as [p q]).
split.
 exact q.

 exact p.

Qed.
and_comm is defined
```

which can be written

$$
\begin{array}{rl}
H: & P \wedge Q \\
p: & P \\
q: & Q \\
\hline
G0: & P \wedge Q \implies Q \wedge P \\
G1: & Q \wedge P \\
G3a: & Q \\
G3b: & P
\end{array}
$$

Step	Tactic
1	intros H
2	destruct H
3	split
4a	assumption
4b	assumption

Similarly for disjunction, if we have the goal to prove $P \vee Q$, then we can either prove P or prove Q. The tactic `left` in Coq makes P a subgoal, and the tactic `right` makes Q a subgoal. Likewise, if we have a hypothesis $P \vee Q$, then `destruct` makes a subgoal with hypothesis P and one with hypothesis Q, which mirrors the *proof by cases* strategy. For example, we can try using the introduction tactic first, or_comm

```
Coq < Lemma or_comm : forall P Q : Prop, (P \/ Q) -> (Q \/ P).
1 subgoal

  ============================
  forall P Q : Prop, P \/ Q -> Q \/ P

or_comm < intros P Q.
1 subgoal

  P, Q : Prop
  ============================
  P \/ Q -> Q \/ P

or_comm < intro pq.
1 subgoal

  P, Q : Prop
  pq : P \/ Q
  ============================
  Q \/ P

or_comm < left.
1 subgoal

  P, Q : Prop
  pq : P \/ Q
  ============================
  Q

or_comm < destruct pq as [p | q].
2 subgoals

  P, Q : Prop
  p : P
  ============================
  Q

subgoal 2 is:
 Q
```

but that is a deadend. We cannot prove `Q` since we only know `P`. We made our choice in the goal too early. Instead, we should execute the proof-by-cases for the hypothesis first and then choose which goal we can prove. Thus we undo that step, and use the elimination tactic instead.

```
or_comm < Undo.
1 subgoal

  P, Q : Prop
  pq : P \/ Q
  ============================
  Q

or_comm < Undo.
1 subgoal

  P, Q : Prop
  pq : P \/ Q
  ============================
  Q \/ P
```

```
or_comm < destruct pq as [p | q].
2 subgoals

  P, Q : Prop
  p : P
  ============================
  Q \/ P

subgoal 2 is:
 Q \/ P

or_comm < right.
2 subgoals

  P, Q : Prop
  p : P
  ============================
  P

subgoal 2 is:
 Q \/ P

or_comm < exact p.
1 subgoal

  P, Q : Prop
  q : Q
  ============================
  Q \/ P

or_comm < left.
1 subgoal

  P, Q : Prop
  q : Q
  ============================
  Q

or_comm < exact q.
No more subgoals.

or_comm < Qed.
(intros P Q).
intro pq.
(destruct pq as [p| q]).
 right.
 exact p.

 left.
 exact q.

Qed.
or_comm is defined
```

which can be written

$$
\begin{aligned}
H :& P \vee Q \\
Ha :& \quad P \\
Hb :& \quad Q \\
\hline
G0 :& P \vee Q \implies Q \vee P \\
G1 :& Q \vee P \\
G3a :& P \\
G3b :& Q
\end{aligned}
$$

Step	Tactic
1	intros H
2	destruct H
3a	right
4a	assumption
3b	left
4b	assumption

The negation operator is a trickier case. When $\neg P$ is the goal, we use the equivalence $P \implies F$ and the introduction tactic for implication. When $\neg P$ is a hypothesis, we use `destruct`, which introduces a subgoal of P, which is a form of proof by contradiction as we show below.

- First we assume our hypothesis ($\neg P$) is false, so our subgoal is P.
- Next, we show that $P \implies \neg P$.
- Since P and $\neg P$ cannot both be true, the assumption must be wrong and $\neg P$ must be true.

Lastly, for logical equivalence, we use `split` to turn a goal into two subgoals with the implication each way, and `destruct` to eliminate a hypothesis in favor of two hypotheses with both implications. All of these tactics and proof strategies are expounded in this excellent lecture by Pierre Castéran. In Table 1.1 we summarize the tactics used to deal with each logical operator and quantifier, and then in Table 1.2 we give the Coq commands which carry out these operations. Note that every tactic which takes a hypothesis argument can also take the name of a lemma or theorem.

A basic example employing negation is the *Law of Noncontradiction* in constructive logic. We start by using `unfold` to replace negation by the equivalent implication form. noncon

```
Coq < Lemma noncon : forall P : Prop, ~(P /\ ~P).
1 subgoal

  ============================
  forall P : Prop, ~ (P /\ ~ P)

noncon < intro P.
1 subgoal

  P : Prop
  ============================
  ~ (P /\ ~ P)

noncon < unfold not.
1 subgoal

  P : Prop
  ============================
  P /\ (P -> False) -> False

noncon < intro H.
1 subgoal

  P : Prop
  H : P /\ (P -> False)
  ============================
  False
```

Now we can separate `H` into two hypotheses by elimination the conjunction,

```
noncon < destruct H as [p np].
1 subgoal

  P : Prop
  p : P
  np : P -> False
  ============================
  False
```

and finally apply the `np` implication to complete the proof.

```
noncon < apply np.
1 subgoal

  P : Prop
  p : P
  np : P -> False
  ============================
  P

noncon < exact p.
No more subgoals.

noncon < Qed.
intro P.
(unfold not).
intro H.
(destruct H as [p np]).
(apply np).
exact p.

Qed.
noncon is defined
```

The Law of Noncontradiction $\neg(P \wedge \neg P)$ expands to $(P \wedge (P \implies F)) \implies F$:

- A proof of $(P \wedge (P \implies F)) \implies F$ is a function f that converts a proof of $(P \wedge (P \implies F))$ into a proof of F.
- A proof of $(P \wedge (P \implies F))$ is a pair of proofs (a, b), where a is a proof of P, and b is a proof of $P \implies F$.
- A proof of $P \implies F$ is a function that converts a proof of P into a proof of F.

Putting it all together, a proof of $(P \wedge (P \implies F)) \implies F$ is a function f that converts a pair (a, b), where a is a proof of P, and b is a function that converts a proof of P into a proof of F, into a proof of F. There is a function f that does this, where $f(a, b) = b(a)$, proving the law of non-contradiction, no matter what P is.

```
Coq < Print noncon.
noncon =
fun (P : Prop) (H : P /\ (P -> False)) =>
match H with
| conj p np => np p
end
     : forall P : Prop, ~ (P /\ ~ P)
```

In fact, this same strategy can provide a proof for $(P \wedge (P \implies Q)) \implies Q$ for any proposition Q. This rule of inference is classically called modus ponens, first described by Theophrastus (Bobzien 2016). We can prove *modus ponens* two different ways. First, the functional way that we are used to modus_ponens

```
Coq < Lemma modus_ponens : forall P Q : Prop, P -> (P -> Q) -> Q.
1 subgoal

  ============================
  forall P Q : Prop, P -> (P -> Q) -> Q

modus_ponens < intros P Q.
1 subgoal

  P, Q : Prop
  ============================
  P -> (P -> Q) -> Q

modus_ponens < intros p pimpq.
1 subgoal

  P, Q : Prop
  p : P
  pimpq : P -> Q
  ============================
  Q

modus_ponens < apply pimpq.
1 subgoal

  P, Q : Prop
  p : P
  pimpq : P -> Q
  ============================
  P

modus_ponens < exact p.
No more subgoals.

modus_ponens < Qed.
(intros P Q).
(intros p pimpq).
(apply pimpq).
exact p.

Qed.
modus_ponens is defined
```

and we see that the proof term mirrors our discussion above of the BHK interpretation of negation

```
Coq < Print modus_ponens.
modus_ponens =
fun (P Q : Prop) (p : P) (pimpq : P -> Q) => pimpq p
     : forall P Q : Prop, P -> (P -> Q) -> Q
```

but we can also explicitly construct the proof term, in a more imperative style,

```
Coq < Lemma modus_ponens_2 : forall P Q : Prop, P -> (P -> Q) -> Q.
1 subgoal

  ============================
  forall P Q : Prop, P -> (P -> Q) -> Q

modus_ponens_2 < intros P Q p pimpq.
1 subgoal

  P, Q : Prop
  p : P
  pimpq : P -> Q
  ============================
  Q

modus_ponens_2 < apply pimpq in p as q.
1 subgoal

  P, Q : Prop
  p : P
  q : Q
  pimpq : P -> Q
  ============================
  Q

modus_ponens_2 < exact q.
No more subgoals.

modus_ponens_2 < Qed.
(intros P Q p pimpq).
(apply pimpq in p as q).
exact q.

Qed.
modus_ponens_2 is defined
```

which has a different looking proof term

```
Coq < Print modus_ponens_2.
modus_ponens_2 =
fun (P Q : Prop) (p : P) (pimpq : P -> Q) => let q := pimpq p : Q in q
     : forall P Q : Prop, P -> (P -> Q) -> Q
```

Another strategy for proving statements using negation is the construction of a contradiction, meaning a proof of `F`. Any proof of false means that the theory is inconsistent and can prove any statement, rendering it trivial. If we know the theory is consistent, then we conclude that the original assumption was in error. This is called *Proof by Negation*. More precisely, the Principle of Explosion is the induction rule for `F`, namely

$$\forall P : Prop, F \implies P$$

An informal presentation of this could be

- Replace `F` with a contradiction, such as "The sky is blue *and* the sky is not blue".
- Then we have the statement "The sky is blue", assumed to be true.
- Therefore, the two-part statement "The sky is blue *or* unicorns exist" must also be true, since the first part is true.

- However, since we know that "The sky is not blue", as this has also been assumed, the first part is false, and hence the second part must be true, i.e. unicorns exist.

We can use this strategy to prove the Law of Noncontradiction with a contradiction,

```
Coq < Lemma noncon3 : forall P : Prop, ~(P /\ ~P).
1 subgoal

  ============================
  forall P : Prop, ~ (P /\ ~ P)

noncon3 < intros P H.
1 subgoal

  P : Prop
  H : P /\ ~ P
  ============================
  False

noncon3 < destruct H as [p np].
1 subgoal

  P : Prop
  p : P
  np : ~ P
  ============================
  False

noncon3 < contradiction.
No more subgoals.

noncon3 < Qed.
(intros P H).
(destruct H as [p np]).
contradiction.

Qed.
noncon3 is defined

Coq < Print noncon3.
noncon3 =
fun (P : Prop) (H : P /\ ~ P) =>
match H with
| conj p np => False_ind False (np p)
end
     : forall P : Prop, ~ (P /\ ~ P)
```

and we can see from the proof term that we use the induction rule for `F`. We can get the same effect from our first proof by using elimination rather than application to get rid of the negation of `P`,

```
Coq < Lemma noncon4 : forall P : Prop, ~(P /\ ~P).
1 subgoal

  ============================
  forall P : Prop, ~ (P /\ ~ P)

noncon4 < intros P H.
1 subgoal

  P : Prop
  H : P /\ ~ P
  ============================
  False
```

```
noncon4 < destruct H as [p np].
1 subgoal

  P : Prop
  p : P
  np : ~ P
  ============================
  False

noncon4 < elim np.
1 subgoal

  P : Prop
  p : P
  np : ~ P
  ============================
  P

noncon4 < exact p.
No more subgoals.

noncon4 < Qed.
(intros P H).
(destruct H as [p np]).
(elim np).
exact p.

Qed.
noncon4 is defined

Coq < Print noncon4.
noncon4 =
fun (P : Prop) (H : P /\ ~ P) =>
match H with
| conj p np => False_ind False (np p)
end
     : forall P : Prop, ~ (P /\ ~ P)
```

Note that this is different from the classical *Proof by Contradiction*. In that strategy, we assume the negation of our goal and show it leads to contradiction, which amounts to double negation, $\neg\neg P \implies P$. This is not valid on constructive logic, and thus we use Proof by Negation instead.

As a last example, we look at the *Conditional Identity*. This is true in classical logic, but only partially true in intuitionistic logic. We can prove the forward implication, cond_ident

```
Coq < Lemma cond_ident : forall P Q : Prop, (~P \/ Q) -> (P -> Q).
1 subgoal

  ============================
  forall P Q : Prop, ~ P \/ Q -> P -> Q

cond_ident < intros P Q H p.
1 subgoal

  P, Q : Prop
  H : ~ P \/ Q
  p : P
  ============================
  Q

cond_ident < destruct H as [np | q].
2 subgoals
```

```
  P, Q : Prop
  np : ~ P
  p : P
  ============================
  Q

subgoal 2 is:
 Q

cond_ident < contradiction.
1 subgoal

  P, Q : Prop
  q : Q
  p : P
  ============================
  Q

cond_ident < exact q.
No more subgoals.

cond_ident < Qed.
(intros P Q H p).
(destruct H as [np| q]).
 contradiction.

 exact q.

Qed.
cond_ident is defined

Coq < Print cond_ident.
cond_ident =
fun (P Q : Prop) (H : ~ P \/ Q) (p : P) =>
match H with
| or_introl np => False_ind Q (np p)
| or_intror q => q
end
     : forall P Q : Prop, ~ P \/ Q -> P -> Q
```

We used our Proof by Contradiction strategy in one branch, as we can also see from the proof term. The reverse implication is not provable constructively since it makes use of the Law of Excluded Middle.

In fact, we can show that there are many equivalent statements which distinguish classical logic. For example, the Law of Excluded Middle, the Conditional Identity, DeMorgan's relation, and Peirce's formula are all equivalent statements. As a start, we prove that the Law of Excluded Middle implies the Conditional Inequality.

```
Coq < Lemma test : forall P Q : Prop, (P \/ ~P) -> ((P -> Q) <-> (~P \/ Q)).
1 subgoal

  ============================
  forall P Q : Prop, P \/ ~ P -> (P -> Q) <-> ~ P \/ Q

test < intros P Q.
1 subgoal

  P, Q : Prop
  ============================
  P \/ ~ P -> (P -> Q) <-> ~ P \/ Q

test < intro em.
1 subgoal
```

```
  P, Q : Prop
  em : P \/ ~ P
  ============================
  (P -> Q) <-> ~ P \/ Q
```

We split the b-implication and prove both directions. Only the forward direction will require the Law of Excluded Middle.

```
test < split.
2 subgoals

  P, Q : Prop
  em : P \/ ~ P
  ============================
  (P -> Q) -> ~ P \/ Q

subgoal 2 is:
 ~ P \/ Q -> P -> Q

test < intro H.
2 subgoals

  P, Q : Prop
  em : P \/ ~ P
  H : P -> Q
  ============================
  ~ P \/ Q

subgoal 2 is:
 ~ P \/ Q -> P -> Q
```

We destruct the excluded middle statement, using P to prove right half of our goal, and $\neg P$ to prove the left half.

```
test < destruct em as [p | np].
3 subgoals

  P, Q : Prop
  p : P
  H : P -> Q
  ============================
  ~ P \/ Q

subgoal 2 is:
 ~ P \/ Q
subgoal 3 is:
 ~ P \/ Q -> P -> Q

test < right.
3 subgoals

  P, Q : Prop
  p : P
  H : P -> Q
  ============================
  Q

subgoal 2 is:
 ~ P \/ Q
subgoal 3 is:
 ~ P \/ Q -> P -> Q

test < apply H.
3 subgoals

  P, Q : Prop
  p : P
  H : P -> Q
  ============================
  P

subgoal 2 is:
 ~ P \/ Q
subgoal 3 is:
 ~ P \/ Q -> P -> Q

test < exact p.
2 subgoals

  P, Q : Prop
  np : ~ P
  H : P -> Q
  ============================
  ~ P \/ Q

subgoal 2 is:
 ~ P \/ Q -> P -> Q

test < left.
2 subgoals

  P, Q : Prop
  np : ~ P
  H : P -> Q
  ============================
  ~ P

subgoal 2 is:
 ~ P \/ Q -> P -> Q

test < exact np.
1 subgoal

  P, Q : Prop
  em : P \/ ~ P
  ============================
  ~ P \/ Q -> P -> Q
```

Now we can prove the reverse implication without using our premise.

```
test < intro ci.
1 subgoal

  P, Q : Prop
  em : P \/ ~ P
  ci : ~ P \/ Q
  ============================
  P -> Q

test < intro p.
1 subgoal

  P, Q : Prop
  em : P \/ ~ P
  ci : ~ P \/ Q
  p : P
  ============================
  Q

test < destruct ci as [np | q].
2 subgoals

  P, Q : Prop
  em : P \/ ~ P
  np : ~ P
  p : P
  ============================
  Q

subgoal 2 is:
 Q

test < contradiction.
1 subgoal

  P, Q : Prop
  em : P \/ ~ P
  q : Q
  p : P
  ============================
  Q

test < exact q.
No more subgoals.

test < Qed.
(intros P Q).
intro em.
split.
 intro H.
 (destruct em as [p| np]).
  right.
  (apply H).
  exact p.

  left.
  exact np.

 intro ci.
 intro p.
 (destruct ci as [np| q]).
  contradiction.

  exact q.

Qed.
test is defined
```

1.5.1 More Examples

For this proof, we use tactics to eliminate the conjuction and disjunction.

```
Coq < Lemma prob3a : forall P Q R : Prop, (P /\ Q) -> (P \/ R).
1 subgoal

  ============================
  forall P Q R : Prop, P /\ Q -> P \/ R

prob3a < intros P Q R pq.
1 subgoal

  P, Q, R : Prop
  pq : P /\ Q
  ============================
  P \/ R

prob3a < destruct pq as [p q].
1 subgoal

  P, Q, R : Prop
  p : P
```

	$\neg P$	$P \implies Q$
Hypothesis `H`	Prove P instead of False	Prove P instead of Q
Conclusion	Assume P, prove False	Assume P, prove Q
	$P \vee Q$	$P \wedge Q$
Hypothesis `H`	Prove using hyp. P then using hyp. Q	Replace with hyp. P and hyp. Q
Conclusion	Prove P or prove Q	Prove P, then prove Q
	$\forall x \in X, P(x)$	$\exists x \in X, P(x)$
Hypothesis `H`	Proves $P(y)$ for $y \in X$	Produce a witness $y \in X$ and the hyp. $P(y)$
Conclusion	Prove $P(x)$	Give a witness w
	$x = y$	F
Hypothesis `H`	Substitute y for x Substitute x for y	False hyp. implies anything
Conclusion	A thing equals itself	

Table 1.1: Tactics in Plain English for Propositional and Predicate Calculus (hyp. = hypothesis)

	$\neg$	$\wedge$	$\vee$
Hypothesis `H`	`elim H`	`elim H`	`elim H`
	`case H`	`case H`	`case H`
		`destruct H as [H1 H2]`	`destruct H as [H1 | H2]`
Conclusion	`intros H`	`split`	`left` or `right`
	$\implies$	$\forall$	$\exists$
Hypothesis `H`	`apply H`	`elim H`	`elim H`
		`apply H`	`case H`
			`destruct H as [x H1]`
Conclusion	`intros H`	`intros H`	`exists` w
	$=$	F	
Hypothesis `H`	`rewrite H`	`elim H`	
	`rewrite <- H`	`case H`	
Conclusion	`reflexivity`		
	`ring`		

Table 1.2: Summary of Coq tactics for Propositional and Predicate Calculus

```
  q : Q
  ============================
  P \/ R

prob3a < left.
1 subgoal

  P, Q, R : Prop
  p : P
  q : Q
  ============================
  P

prob3a < exact p.
No more subgoals.

prob3a < Qed.
(intros P Q R pq).
(destruct pq as [p q]).
left.
exact p.

Qed.
prob3a is defined
```

$$\begin{array}{r l}
H : & P \wedge Q \\
p : & P \\
q : & Q \\
\hline
G0 : & P \wedge Q \implies P \vee R \\
G1 : & P \vee R \\
G3 : & P
\end{array}$$

Step	Tactic
1	intros H
2	destruct H
3	left
4	assumption

When we deal with implication, we use the introduction tactic to assume the antecdent.

```
Coq < Lemma prob3b : forall P R : Prop, P->(R->P).
1 subgoal

  ============================
  forall P R : Prop, P -> R -> P

prob3b < intros P R p r.
1 subgoal

  P, R : Prop
  p : P
  r : R
  ============================
  P

prob3b < exact p.
No more subgoals.

prob3b < Qed.
(intros P R p r).
exact p.

Qed.
prob3b is defined
```

$$\begin{array}{l} p : P \\ r : R \\ \hline G0 : P \implies R \implies P \\ G1 : R \implies P \\ G3 : P \end{array}$$

Step	Tactic
1	intros p
2	intros r
3	assumption

1.5.2 Submitting Coq Homework

All proof will be submitted as Coq source files, which use the extension `.v`. The automated system will compile your code and then check that it proves the desired theorem. A Coq source file consists essentially of a set of proofs. Each proof has a statement, followed by a list of tactics. These tactics are what is often printed when `Qed` is executed, or by running the `Show Script` command during the proof. For example, if we were to prove that $P \implies P$, our source file `easy.v` would contain

```
Lemma easy : forall P : Prop, P -> P.
Proof.
intros P p.
exact p.
Qed.
```

You would then submit the `easy.v` to Autograder. Grading should happen immediately upon submission, so that you can check your score and resubmit if necessary.

1.5.3 Compiling Coq Files

We can develop libraries of proofs in Coq by using the compiler. We begin by creating a source file, `identity.v`, of the commands we used to prove our simple theorem. The file contents are shown below.

```
Section Logic_Examples.
Lemma id_P : forall P : Prop, P->P.
intros P p.
exact p.
Qed.
End Logic_Examples.
```

Now we compile this file using the Coq compiler `coqc`. If we put our source code in the `code` directory, then we want to map that directory to the `code` namespace in Coq using the `-R` flag,

```
  coqc -R $PWD/code code code/identity.v
```

which creates an object file, `code/identity.vo`, as well as an auxiliary file `code/identity.glob` that describes the compilation process. We can check this file using the proof checker

```
  coqchk -R $PWD/code code code.identity
```

which produces

```
Checking library: identity
  checking cst: identity.id_P
Modules were successfully checked
```

Now we can use our compiled library in the toplevel interpreter

```
> coqtop -R $PWD/code code

Welcome to Coq 8.6 (December 2017)

Coq < Require Import code.identity.

Coq < Print code.identity.id_P.
Fetching opaque proofs from disk for code.identity
id_P = fun (P : Prop) (p : P) => p
     : forall P : Prop, P -> P

Argument scopes are [type_scope _]
```

Using this process, you should be able to verify that your homework submissions compile before you submit.

1.6 Commonsense Interpretation

Coq does not automate the process of proving theorems, but rather assists the prover by clarifying the alternatives and checking the results. Thus, we will look for commonsense proof techniques which we can use when proving by hand, which mirror the tactics in Coq. For example, if I need to prove

$$P \wedge Q$$

then I would first prove P and then prove Q. This is exactly what the `split` tactic does. On the other hand, if I have a hypothesis $P \wedge Q$, so it is true, then it must mean that both P and Q are true, which is exactly what the `destruct` tactic conveys.

Similarly, if I want to prove

$$P \vee Q$$

then I must either prove P, or prove Q, which I can choose with the `left` or `right` tactics. If I have a hypothesis $P \vee Q$, then I have a case where P is true (one proof) and then another case where Q is true (another proof), which is what `destruct` gives me.

When I have an implication

$$P \implies Q$$

it is necessarily true when P is false, so the only case I need to be concerned with is P true. Thus I add P as a hypothesis and try to prove Q, which is accomplished with the `intro` tactic. If $P \implies Q$ is a hypothesis, then I can think of it as a machine for turning a proof for P into a proof for Q. Thus if I have a goal Q, I can replace that with a goal P since I can always turn P into Q in some sense. Moreover, we often write multiple hypotheses for a proof as

$$P \implies R \implies S \implies Q$$

instead of

$$(P \wedge R \wedge S) \implies Q.$$

We can look at the truth table

P	R	S	Q	$P \implies (R \implies (S \implies Q))$	$(P \wedge R \wedge S) \implies Q$
T	T	T	T	T	T
T	T	T	F	F	F
T	T	F	T	T	T
T	T	F	F	T	T
T	F	T	T	T	T
T	F	T	F	T	T
T	F	F	T	T	T
T	F	F	F	T	T
F	T	T	T	T	T
F	T	T	F	T	T
F	T	F	T	T	T
F	T	F	F	T	T
F	F	T	T	T	T
F	F	T	F	T	T
F	F	F	T	T	T
F	F	F	F	T	T

which shows us that these are identical statements.

Lastly, proof incorporating negation can always be done by contradiction, which is how the proof assistant handles them. If I want to prove $\neg P$, I start by assuming P and showing that a contradiction results. This strategy follows from the equivalence

$$\neg P \iff P \implies \text{F}. \tag{1.10}$$

The introduction tactic for the implication is the initial assumption in a proof by contradiction, and the goal of False is the contradiction itself.

1.7 Problems

Problem I.1 This written problem will familiarize you with the grading system that we use at UB. Follow the steps below to ensure that your Autolab account is working correctly.

1. Create your account at https://autograder.cse.buffalo.edu using your UB email address.

2. An account may have been created for you if you enrolled before you had an account. If Autolab says that you already have an account, click "Forgot your password?" and enter your email address. Follow instructions to reset your password.

3. Ensure that you are registered for the course: CSE191: Discrete Structures

4. Submit a pdf to Homework 0 with the following information:
   - Name
   - Person number
   - A programming language you have used, or None

The best way to create PDF from LaTeX is to use `pdflatex`,

```
pdflatex essay.tex
bibtex essay
pdflatex essay.tex
pdflatex essay.tex
```

where the repetition is necessay to assure that the metadata stored in auxiliary files is consistent. This process can be handled in an elegant way by using the `latexmk` program,

```
latexmk -pdf essay.tex
```

If you rely on TeX source or BibTeX files in other locations, you can use

```
TEXINPUTS=${TEXINPUTS}:/path/to/tex BIBINPUTS=${BIBINPUTS}:/path/to/bib
  latexmk -pdf essay.tex
```

Problem I.2 This problem will teach you to submit a Coq proof for automatic grading by the Autograder system. We will prove the simplest proposition we have seen

$$\forall P : Prop, P \implies P.$$

You will prepare a text file named *ChPropositionalLogic.v* which contains the statement and tactics for the proof. You must name the lemma *identity*

```
Lemma identity : forall P : Prop, P -> P.
Proof.
intros P p.
exact p.
Qed.
```

You then submit this file to Autograder, just as you submitted your PDF in the last assignment.

Problem I.3 Indicate whether the statement is a proposition. If not, what kind of statement is it?

1. Have a nice day.

2. The patient has diabetes.

3. The light is on.
4. It's a beautiful day.
5. Do you like my new shoes?
6. The sky is purple.
7. $2 + 3 = 6$
8. Every prime number is even.
9. There is a number that is larger than 17.

Problem I.4 Indicate whether the statement is a proposition. If not, what kind of statement is it?

1. $4^2 = 16$
2. The GDP is $10 trillion dollars.
3. Have a great day.
4. The light is off.
5. Every prime number is odd.
6. Did you vote this year?
7. Music is nowhere near as good as in the 80s.
8. The moon is made of cheese.
9. There is a number that is divisible by 17.

Problem I.5 Look up the NAND and NOR operations, and write the truth table for both.

Problem I.6 Assume the propositions p, q, r, and s have the following truth values:

- $p :$ True
- $q :$ False
- $r :$ True
- $s :$ False

Provide the truth value for each of the following compound propositions. Show your work.

1. $\neg p \vee (q \wedge s)$

2. $\neg(p \land q) \lor r$
3. $(p \lor q) \land \neg(p \lor q)$
4. $(p \land q) \implies \neg(r \lor s)$
5. $(r \implies s) \iff (q \lor \neg p)$

Problem I.7 Assume the propositions p, q, r, and s have the following truth values:

- p : False
- q : True
- r : True
- s : False

Provide the truth value for each of the following compound propositions. Show your work.

1. $(p \land q) \lor \neg(p \land q)$
2. $(r \land s) \lor \neg p$
3. $(r \land q) \implies \neg(p \land s)$
4. $\neg(s \land p) \land r$
5. $(p \implies q) \iff (\neg r \lor s)$

Problem I.8 Define the following propositions:

- j : a person is a Jedi
- p : a person is not a Padawan
- l : a person is allowed to use a lightsaber

Express each of the following English sentences with a logical expression:

1. A person is allowed to use a lightsaber only if they are a Jedi and not a Padawan.
2. A person is allowed to use a lightsaber if they are a Jedi or a Padawan.
3. Not being a Padawan is a necessary condition for being allowed to use a lightsaber.
4. A person is allowed to use a lightsaber if and only if the person is a Jedi and is not a Padawan.
5. Being allowed to use a lightsaber implies that the person is either a Jedi or a Padawan.

Problem I.9 Define the following propositions:

- g : a person is in Gryffindor House
- s : a person is in Slytherin House
- w : a person won the Quidditch match

Express each of the following English sentences with a logical expression:

1. If she won the match, she is either in Gryffindor or Slytherin House.
2. He won the match if he is in Gryffindor and not in Slytherin House.
3. Gryffindor House never wins at Quidditch.
4. Winning today implies that she isn't in Gryffindor or Slytherin House.
5. She is in Gryffindor if and only if she is not in Slytherin House.

Problem I.10 Below we tabulate laws of logical equivalence. Verify the Distributive, De Morgan's, and Absorption laws using a truth table.

Equivalence	Name
$p \wedge \mathrm{T} \equiv p, \quad p \vee \mathrm{F} \equiv p$	Identity laws
$p \vee \mathrm{T} \equiv \mathrm{T}, \quad p \wedge \mathrm{F} \equiv \mathrm{F}$	Domination laws
$p \vee p \equiv p, \quad p \wedge p \equiv p$	Idempotent laws
$\neg(\neg p) \equiv p$	Double negation law
$p \vee q \equiv q \vee p$ $p \wedge q \equiv q \wedge p$	Commutative laws
$(p \vee q) \vee r \equiv p \vee (q \vee r)$ $(p \wedge q) \wedge r \equiv p \wedge (q \wedge r)$	Associative laws
$p \vee (q \wedge r) \equiv (p \vee q) \wedge (p \vee r)$ $p \wedge (q \vee r) \equiv (p \wedge q) \vee (p \wedge r)$	Distributive laws
$\neg(p \vee q) \equiv \neg p \wedge \neg q$ $\neg(p \wedge q) \equiv \neg p \vee \neg q$	De Morgan's laws
$p \vee (p \wedge q) \equiv p$ $p \wedge (p \vee q) \equiv p$	Absorption laws
$p \vee \neg p \equiv \mathrm{T}, \quad p \wedge \neg p \equiv \mathrm{F}$	Complement laws

Problem I.11 The laws of intuitionistic logic are not equivalent to classical logic, and catching the differences can be tricky. For example, the contrapositive inference rule,

$$P \implies Q \iff \neg Q \implies \neg P \tag{1.11}$$

cannot be proven in intuitionistic logic. Using constructive logic in Coq, generate **Proof:** *contrap* for

$$(P \implies Q) \implies (\neg Q \implies \neg P) \tag{1.12}$$

which would be

```
Lemma contrap : forall P Q : Prop, (P -> Q) -> (~Q -> ~P).
```

and also **Proof:** *revcontrap*

$$(\neg Q \implies \neg P) \implies (P \implies \neg\neg Q). \tag{1.13}$$

which would be

```
Lemma revcontrap : forall P Q : Prop, (~Q -> ~P) -> (P -> ~~Q).
```

Follow the guidelines for submitting constructive proofs using Coq.

Problem I.12 Generate **Proof:** *add_iff* for

$$\forall PQ : Prop, (Q \vee P) \wedge (\neg P \implies Q) \iff (P \vee Q). \tag{1.14}$$

which would be

```
Lemma add_iff : forall P Q : Prop, (Q \/ P) /\ (~P -> Q) <-> (P \/ Q).
```

using constructive logic. Follow the guidelines for submitting constructive proofs using Coq.

Problem I.13 Generate **Proof:** *weak_pierce* for the Weak Pierce relation

$$\forall PQ : Prop, ((((P \implies Q) \implies P) \implies P) \implies Q) \implies Q. \tag{1.15}$$

which would be

```
Lemma weak_pierce : forall P Q : Prop, (((P -> Q) -> P) -> P) -> Q) ->Q.
```

using constructive logic. Follow the guidelines for submitting constructive proofs using Coq.

Problem I.14 Generate **Proof:** *neg_imp* for

$$P \implies (Q \wedge R) \tag{1.16}$$

$$\neg Q \tag{1.17}$$

$$\therefore \neg P \tag{1.18}$$

which is equivalent to

$$\forall PQR : Prop, (P \implies (Q \wedge R)) \implies (\neg Q) \implies (\neg P)$$

which would be

```
Lemma neg_imp : forall P Q R : Prop, (P -> (Q /\ R)) -> ~Q -> ~P.
```

using constructive logic. Follow the guidelines for submitting constructive proofs using Coq.

Problem I.15 Generate **Proof:** *double_modus_ponens* for

$$P \implies Q \tag{1.19}$$
$$R \implies S \tag{1.20}$$
$$P \wedge R \tag{1.21}$$
$$\overline{}$$
$$\therefore Q \wedge S \tag{1.22}$$

which is equivalent to

$$\forall PQRS : Prop, (P \implies Q) \implies (R \implies S) \implies (P \wedge R) \implies (Q \wedge S)$$

which would be

```
Lemma double_modus_ponens : forall P Q R S : Prop, (P->Q) -> (R->S) -> (P /\ R) -> (Q /\ S).
```

using constructive logic. Follow the guidelines for submitting constructive proofs using Coq.

Problem I.16 Prove that

$$P \implies Q \implies (P \iff Q) \tag{1.23}$$

using the Coq syntax

```
Lemma bothtrue : forall P Q : Prop, P -> Q -> (P <-> Q).
```

Problem I.17 Prove that

$$\neg P \implies \neg Q \implies (P \iff Q) \tag{1.24}$$

using the Coq syntax

```
Lemma bothfalse : forall P Q : Prop, ~P -> ~Q -> (P <-> Q).
```

Problem I.18 Prove the transitivity of logical equivalence

$$(P \iff Q) \implies (Q \iff R) \implies (P \iff R) \tag{1.25}$$

using the Coq syntax

```
Lemma iff_trans : forall P Q R : Prop, (P <-> Q) -> (Q <-> R) -> (P <-> R).
```

Problem I.19 Prove that functions with multiple arguments can be turned into a series of function taking two arguments, one of which is another function,

$$((P \wedge Q) \implies R) \implies (P \implies (Q \implies R)) \tag{1.26}$$

using the Coq syntax

```
Lemma curry : forall P Q R : Prop, ((P /\ Q) -> R) -> (P -> (Q -> R)).
```

Also, prove the converse

```
Lemma uncurry : forall P Q R : Prop, (P -> (Q -> R)) -> ((P /\ Q) -> R).
```

Problem I.20 Prove that disjunction in an implication can be split into two simpler implications,

$$((P \vee Q) \implies R) \iff ((P \implies R) \wedge (Q \implies R)) \tag{1.27}$$

using the Coq syntax

```
Lemma or_uni : forall P Q R : Prop, ((P \/ Q) -> R) <-> ((P -> R) /\ (Q -> R)).
```

Problem I.21 Prove that conjunction in an implication can be split into two simpler implications,

$$(P \implies (Q \wedge R)) \iff ((P \implies Q) \wedge (P \implies R)) \tag{1.28}$$

using the Coq syntax

```
Lemma and_uni : forall P Q R : Prop, (P -> (Q /\ R)) <-> ((P -> Q) /\ (P -> R)).
```

Problem I.22 Prove that equality is a symmetric relation,

$$P = Q \implies Q = P \tag{1.29}$$

using the Coq syntax

```
Lemma eq_symmetric : forall P Q : Prop, P = Q -> Q = P.
```

Problem I.23 Prove that equality is a symmetric relation,

$$P = Q \implies Q = P \tag{1.30}$$

using the Coq syntax

```
Lemma eq_transitive : forall P Q R : Prop, P = Q -> Q = R -> P = R.
```

References

Kolmogorov, A. N. (1925). “On the principle ”tertium non datur””. In: *Mathematicheskiĭ Sbornik* 32. English translation, On the principle of excluded middle, in van Heijenoort J., *From Frege to Gödel: a source-book in mathematical logic, 1879–1931*, 1967, pp. 416–437, pp. 646–667.

Bobzien, Susanne (2016). “Ancient Logic”. In: *The Stanford Encyclopedia of Philosophy*. Ed. by Edward N. Zalta. Winter 2016. Metaphysics Research Lab, Stanford University.

Chapter 2

Predicate Logic

Suppose that you want to teach the 'cat' concept to a very young child. Do you explain that a cat is a relatively small, primarily carnivorous mammal with retractible claws, a distinctive sonic output, etc.? I'll bet not. You probably show the kid a lot of different cats, saying 'kitty' each time, until it gets the idea. To put it more generally, generalizations are best made by abstraction from experience.

— R. P. Boas

2.1 Predicates and Quantifiers

A *predicate* is a function from some domain $\mathcal{D}$ to the type Prop, or we can think of it as a function which returns a proposition. We normally denote a predicate as $P(x)$. For example, the predicate $P(x) : x > 5$ applies to the domain of natural numbers. A predicate may have any number of arguments, such as $P(x, y, z) : x + y > 5z$.

In order to derive truth values from predicates, we constrain or quantify the input values. If we want to know if a predicate P is true for every element of the domain, we would use the *universal quantifier*

$$\forall x \in \mathcal{D}, P(x). \tag{2.1}$$

which is translated as "for all x in $\mathcal{D}$, P is true". If instead we wanted to know if P was true for any of the elements in the domain, we would use the *existential quantifier*

$$\exists x \in \mathcal{D}, P(x). \tag{2.2}$$

which is translated as "there exists an x in $\mathcal{D}$ such that P is true". We often omit the domain from the quantifier if it is clear from the context.

We can express the truth of a statement of predicate logic using a truth table in the same way we treated propositional logic. Suppose that for the domain $\mathcal{D} = \{a, b, c\}$, $P(x, y)$ has truth values

P	a	b	c
a	T	T	F
b	F	F	T
c	T	F	T

where the first argument indicates the row (vertical), and the second indicates the column (horizontal). The statement $\exists x \exists y, P(x, y)$ asks whether any entry in this table is true. Since $P(a, a)$ is true, the statement is true. The element $P(a, a)$ is called a *witness*, which is what is needed to establish the truth of an existential statement. The statement $\forall x \forall y, P(x, y)$ asks whether all the entries in the table are true. Since $P(b, b)$ is false, the statement is false. The element $P(b, b)$ is called a *counterexample*, which is what is needed to establish the falsity of a universal statement. We can also use P to demonstrate that the universal and existential quantifiers do not commute, although existential quantifiers commute among themselves, as do universal quantifiers. The statement $\exists x \forall y, P(x, y)$ asks whether there is a row where all entries are true. Since no row has all trues, this statement is false. However $\forall y \exists x, P(x, y)$ asks whether each column has a true entry, which is true.

In order to understand the negation of the existential quantifier, $\neg\exists x, P(x)$, we recall that the existence of a witness $P(a)$ proves the statement. Thus, the negation means that no witness can be found, so that all the evaluations of P are false, or in predicate logic terms $\forall x, \neg P(x)$. Similarly, if we negate the universal quantifier, $\neg\forall x, P(x)$, it means that a counterexample can be found, or there exists an x which contradicts P, $\exists x, \neg P(x)$. This is the analogue of the De Morgan relation for quantifiers.

We can imagine the action of quantifiers by explicitly expanding them for a given domain. Suppose that we are using the domain $\mathcal{D}$ from above with three elements. Then the statement

$$\forall x : \mathcal{D}, P(x)$$

would be equivalent to the chain of conjunctions

$$P(a) \wedge P(b) \wedge P(c).$$

If we had an infinite type, like the natural numbers,

$$\forall n : \mathbb{N}, P(n)$$

then the chain would be infinite.

$$P(0) \wedge P(1) \wedge P(2) \ldots \wedge P(n) \wedge \ldots$$

The existential quantifier

$$\exists x : \mathcal{D}, P(x)$$

behaves like a chain of disjunctions

$$P(a) \vee P(b) \vee P(c).$$

since it needs to pick out just one witness, and the infinite version

$$\exists n : \mathbb{N}, P(n)$$

is similar

$$P(0) \vee P(1) \vee P(2) \ldots \vee P(n) \vee \ldots$$

We can reinterpret quantifiers from the point of view of our constructive system by thinking of them as functions. Suppose that we know that a universal statement is true,

$$\forall n : \mathbb{N}, P(n).$$

That means that we can produce a proof of $P(n)$ for any natural number n. Thus we can think of the universal quantifier as a function which takes in a predicate P and any member n of the domain, producing a proof of $P(n)$. In this sense, universal quantification is just like implication, and we use exactly the same tactics to handle it as we do for implication. When we prove a universal statement by induction, we are constructing an infinite tower of implications, as we will see in Section 4.2. In Coq, if a universal statement is in our hypotheses, we can apply it to any member of the domain to generate a specific proof.

If an existential statement is true

$$\exists n : \mathbb{N}, P(n),$$

it means that there is some n for which we can prove $P(n)$. Thus we can think of the existential quantifier as a function that takes in a predicate P and produces a member of the domain n, the witness, and a proof of $P(n)$. When an existence statement is one of our hypotheses, we can extract the witness using the `destruct` tactic.

2.2 Using Coq

We must first setup our proof environment, initializing variables with the correct types.

```
Coq < Section Pred_Examples.

Coq < Variables A : Set.
A is declared

Coq < Variables P Q : A->Prop.
P is declared
Q is declared
```

We see that the predicates P and Q map set elements to propositions, so that if $x \in A$, $P(x)$ is a proposition. Since the proposition has a truth value, we can also think of P as a Boolean function $A \to \{\mathrm{T}, \mathrm{F}\}$.

We will try and prove that the existential quantifier distributes over disjunction, meaning

$$\exists x : A, P(x) \vee Q(x) \iff (\exists x : A, P(x)) \vee (\exists x : A, Q(x))\,. \tag{2.3}$$

We can state the forward implication, ex_dist_or_for, use the introduction rule for implication, and then eliminate the quantifier in the hypothesis.

```
Coq < Lemma ex_dist_or_for : (exists x:A, P x \/ Q x) -> (ex P) \/ (ex Q).
1 subgoal

  A : Set
  P, Q : A -> Prop
  ============================
  (exists x : A, P x \/ Q x) -> (exists y, P y) \/ (exists y, Q y)

ex_dist_or_for < intro H.
1 subgoal

  A : Set
  P, Q : A -> Prop
  H : exists x : A, P x \/ Q x
  ============================
  (exists y, P y) \/ (exists y, Q y)

ex_dist_or_for < destruct H.
1 subgoal

  A : Set
  P, Q : A -> Prop
  x : A
  H : P x \/ Q x
  ============================
  (exists y, P y) \/ (exists y, Q y)
```

As a counterpart to the introduction rule, eliminating a hypothesis containing an existential quantification produces a variable that satisfies the quantified formula, or witness, which here is x. Now we can treat the new hypothesis H just as we did in propositional logic and do a proof by cases.

```
ex_dist_or_for < destruct H.
2 subgoals

  A : Set
  P, Q : A -> Prop
  x : A
  H : P x
  ============================
  (exists y, P y) \/ (exists y, Q y)

subgoal 2 is:
 (exists y, P y) \/ (exists y, Q y)
```

For the first case, we use the introduction tactic for disjunction, and then a new introduction tactic for the witness called `exists` that allows us to satisfy the existence clause.

```
ex_dist_or_for < left.
2 subgoals

  A : Set
  P, Q : A -> Prop
  x : A
  H : P x
  ============================
  exists y, P y

subgoal 2 is:
 (exists y, P y) \/ (exists y, Q y)

ex_dist_or_for < exists x.
2 subgoals

  A : Set
  P, Q : A -> Prop
  x : A
  H : P x
  ============================
  P x

subgoal 2 is:
 (exists y, P y) \/ (exists y, Q y)

ex_dist_or_for < assumption.
1 subgoal

  A : Set
  P, Q : A -> Prop
  x : A
  H : Q x
  ============================
  (exists y, P y) \/ (exists y, Q y)
```

We do the same thing for the other case, and the lemma is proved.

```
ex_dist_or_for < right.
1 subgoal

  A : Set
  P, Q : A -> Prop
  x : A
  H : Q x
  ============================
  exists y, Q y

ex_dist_or_for < exists x.
1 subgoal

  A : Set
  P, Q : A -> Prop
  x : A
  H : Q x
  ============================
  Q x

ex_dist_or_for < assumption.
No more subgoals.

ex_dist_or_for < Qed.
intro H.
(destruct H).
(destruct H).
 left.
 exists x.
 assumption.

 right.
 exists x.
 assumption.

Qed.
ex_dist_or_for is defined
```

```
Coq <
```

The reverse implication is proved in Problem 3.

We can also try to prove the DeMorgan relation for the existential quantifier, demorgan_forall. The forward implication is straightforward. We first use introduction to eliminate the implication. Then we use the `unfold` tactic to replace $\neg P$ with $P \implies \mathrm{F}$,

```
Coq < Lemma demorgan_forall : (forall x : A, P x) -> ~ (exists y : A, ~P y).
1 subgoal

  A : Set
  P, Q : A -> Prop
  ============================
  (forall x : A, P x) -> ~ (exists y : A, ~ P y)

demorgan_forall < intro H.
1 subgoal

  A : Set
  P, Q : A -> Prop
  H : forall x : A, P x
  ============================
  ~ (exists y : A, ~ P y)

demorgan_forall < unfold not.
1 subgoal

  A : Set
  P, Q : A -> Prop
  H : forall x : A, P x
  ============================
  (exists y : A, P y -> False) -> False
```

We eliminate the implication, and then use the `destruct` tactic to produce a witness for the existential hypothesis.

```
demorgan_forall < intro.
1 subgoal

  A : Set
  P, Q : A -> Prop
  H : forall x : A, P x
  H0 : exists y : A, P y -> False
  ============================
  False

demorgan_forall < destruct H0.
1 subgoal

  A : Set
  P, Q : A -> Prop
  H : forall x : A, P x
  x : A
  H0 : P x -> False
  ============================
  False
```

Now we apply hypothesis `H0` to generate the goal $P(x)$. Finally we apply H, since it says that $P(z)$ is true for all z, so $P(x)$ must be true.

```
demorgan_forall < apply H0.
1 subgoal
```

```
  A : Set
  P, Q : A -> Prop
  H : forall x : A, P x
  x : A
  H0 : P x -> False
  ============================
  P x

demorgan_forall < apply H.
No more subgoals.

demorgan_forall < Qed.
intro H.
(unfold not).
intro.
(destruct H0).
(apply H0).
(apply H).

Qed.
demorgan_forall is defined
```

The proof done with shorthand, without unfolding the negation statements, is given below.

```
Coq < Lemma demorgan_forall : (forall x : A, P x) -> ~ (exists y : A, ~P y).
1 subgoal

  A : Set
  P, Q : A -> Prop
  ============================
  (forall x : A, P x) -> ~ (exists y : A, ~ P y)

demorgan_forall < intro H.
1 subgoal

  A : Set
  P, Q : A -> Prop
  H : forall x : A, P x
  ============================
  ~ (exists y : A, ~ P y)

demorgan_forall < intro H1.
1 subgoal

  A : Set
  P, Q : A -> Prop
  H : forall x : A, P x
  H1 : exists y : A, ~ P y
  ============================
  False

demorgan_forall < destruct H1.
1 subgoal

  A : Set
  P, Q : A -> Prop
  H : forall x : A, P x
  x : A
  H0 : ~ P x
  ============================
  False

demorgan_forall < destruct H0.
1 subgoal

  A : Set
  P, Q : A -> Prop
  H : forall x : A, P x
  x : A
  ============================
  P x

demorgan_forall < apply H.
No more subgoals.

demorgan_forall < Qed.
intro H.
intro H1.
(destruct H1).
(destruct H0).
(apply H).

Qed.
demorgan_forall is defined

Coq <
```

The reverse implication, however, cannot be proved in intuitionistic logic, for the same reason that we could not prove the converse of the contrapositive law. In Problem 5, we construct a weaker version which can be proved.

2.3 Inductive Types

Induction is reasoning from the specific to the general, as opposed to its dual *deduction*, which reasons from the general to the specific. In essence, induction is no more than reasoning by cases, which is clear from finite induction. Infinite induction makes use of chains of implications to fill out the entire universe of cases. In the end, induction generates the general statement from which deduction begins. In fact, deduction is just the application of a universal quantification to a specific proposition.

Inductive types are used by Coq to subsume the different type definitions found in conventional programming languages. Each inductive type corresponds directly to a computation, based on pattern matching and recursion. These computation structures provide the basis for recursive programming and induction in Coq.

The simplest inductive type is *False*, which has no members, and we say that the type is not *inhabited*. It is the analogue of the empty set in type theory. When we reason inductively using this type, there are no cases to consider as we see from its definition,

```
Coq < Print False.
Inductive False : Prop :=

Coq < Print False_ind.
False_ind = fun P : Prop => False_rect P
    : forall P : Prop, False -> P
```

which means that $F \implies P$ is true for any P. However, we do not actually use the rule, since the type is not inhabited, unless we are doing a proof by contradiction.

The type with only one member we call *True*, and its one member we can call I, or sometimes *triv* for trivial. When using induction with this type, there is only one case to check, namely that the proposition itself can be proved. Thus, $T \implies P$ is true if and only if P can be proved.

```
Coq < Print True.
Inductive True : Prop := I : True

Coq < Print True_ind.
True_ind =
fun P : Prop => True_rect (P:=P)
    : forall P : Prop, P -> True -> P
```

The type *Bool* has two members, true and false. Note that these are just names, and do not relate to proving propositions. When using finite induction for a predicate over the bool domain, we must check two cases, namely $P(\text{true})$ and $P(\text{false})$. If these are both provable, then we have our general statement $\forall b : \mathit{bool}, P(b)$.

```
Coq < Print bool.
Inductive bool : Set := true : bool | false : bool

Coq < Print bool_ind.
bool_ind =
fun P : bool -> Prop => bool_rect P
    : forall P : bool -> Prop, P true -> P false -> forall b : bool, P b
```

2.3.1 Types without Recursion

Inductive types are capable of representing types without recursion, akin to the record types with variants that are represented in C using `struct` and `union` types. For example, for the boolean type in Coq

```
Coq < Print bool.
Inductive bool : Set := true : bool | false : bool
```

and we could imagine more elaborate enumerated types, such as

```
Inductive pdetype : Set :=
  elliptic : pdetype | parabolic : pdetype | hyperbolic : pdetype.
```

The elements of the type are called *constructors* of the type, in analogy with the constructors for objects of a given type.

In order to reason about and compute on data of this type, Coq adds induction theorems automatically when the type is defined. For the boolean type, the first theorem is called `bool_ind`, which encapsulates the induction principle associated with the inductive definition,

```
Coq < Check bool_ind.
bool_ind
     : forall P : bool -> Prop, P true -> P false -> forall b : bool, P b
```

What is this induction theorem saying? If we take any predicate P over the booleans, then proving that $P(\mathrm{T})$ is true and $P(\mathrm{F})$ is true implies that P holds over all booleans. So induction over a finite set just means testing the predicate on every member of the set. Therefore, a similar theorem exists for the PDE type

```
Coq < Check pdetype_ind.
pdetype_ind
     : forall P : pdetype -> Prop,
       P elliptic -> P parabolic -> P hyperbolic -> forall p : pdetype, P p
```

where again we see that the content of the theorem is that in order for some predicate to hold for all members of the inductive type, it must hold for each one individually. Coq also create variants `pdetype_rec` and `pdetype_rect` which replace the `Prop` sort in the predicate with `Set` and `Type` sorts respectively.

We can do a simple proof by induction for the `pdetype` type, pde_equal

```
Coq < Theorem pde_equal : forall p : pdetype, p = parabolic \/ p = elliptic \/ p = hyperbolic.
1 subgoal

  ============================
  forall p : pdetype, p = parabolic \/ p = elliptic \/ p = hyperbolic

pde_equal < induction p.
3 subgoals

  ============================
  elliptic = parabolic \/ elliptic = elliptic \/ elliptic = hyperbolic

subgoal 2 is:
 parabolic = parabolic \/ parabolic = elliptic \/ parabolic = hyperbolic
subgoal 3 is:
 hyperbolic = parabolic \/ hyperbolic = elliptic \/ hyperbolic = hyperbolic

pde_equal < right.
3 subgoals

  ============================
  elliptic = elliptic \/ elliptic = hyperbolic

subgoal 2 is:
 parabolic = parabolic \/ parabolic = elliptic \/ parabolic = hyperbolic
subgoal 3 is:
 hyperbolic = parabolic \/ hyperbolic = elliptic \/ hyperbolic = hyperbolic

pde_equal < left.
3 subgoals

  ============================
  elliptic = elliptic

subgoal 2 is:
 parabolic = parabolic \/ parabolic = elliptic \/ parabolic = hyperbolic
subgoal 3 is:
 hyperbolic = parabolic \/ hyperbolic = elliptic \/ hyperbolic = hyperbolic

pde_equal < reflexivity.
2 subgoals

  ============================
  parabolic = parabolic \/ parabolic = elliptic \/ parabolic = hyperbolic

subgoal 2 is:
 hyperbolic = parabolic \/ hyperbolic = elliptic \/ hyperbolic = hyperbolic

pde_equal < left.
2 subgoals

  ============================
  parabolic = parabolic

subgoal 2 is:
 hyperbolic = parabolic \/ hyperbolic = elliptic \/ hyperbolic = hyperbolic

pde_equal < reflexivity.
1 subgoal

  ============================
  hyperbolic = parabolic \/ hyperbolic = elliptic \/ hyperbolic = hyperbolic

pde_equal < right.
1 subgoal

  ============================
  hyperbolic = elliptic \/ hyperbolic = hyperbolic

pde_equal < right.
1 subgoal

  ============================
  hyperbolic = hyperbolic

pde_equal < reflexivity.
No more subgoals.

pde_equal < Qed.
(induction p).
 right.
 left.
 reflexivity.

 left.
 reflexivity.

 right.
 right.
 reflexivity.

Qed.
pde_equal is defined
```

We can analyze exactly what `induction` is doing here by going through the steps by hand. First we introduce a hypothesis for the quantification,

```
Coq < Theorem pde_equal : forall p : pdetype, p = parabolic \/ p = elliptic \/ p = hyperbolic.
1 subgoal

  ============================
  forall p : pdetype, p = parabolic \/ p = elliptic \/ p = hyperbolic

pde_equal < intro p.
1 subgoal

  p : pdetype
  ============================
  p = parabolic \/ p = elliptic \/ p = hyperbolic
```

Then we use the `pattern` tactic to make our goal a function application. This gives us a predicate instead of a raw statement,

```
pde_equal < pattern p.
1 subgoal

  p : pdetype
  ============================
  (fun p0 : pdetype => p0 = parabolic \/ p0 = elliptic \/ p0 = hyperbolic) p
```

Finally, we apply the induction principle for this type,

```
pde_equal < apply pdetype_ind.
3 subgoals

  p : pdetype
  ============================
  elliptic = parabolic \/ elliptic = elliptic \/ elliptic = hyperbolic
```

```
subgoal 2 is:
 parabolic = parabolic \/ parabolic = elliptic \/ parabolic = hyperbolic
subgoal 3 is:
 hyperbolic = parabolic \/ hyperbolic = elliptic \/ hyperbolic = hyperbolic
```

We could also accomplish the same thing using the `elim` tactic, which uses `pattern` and `apply` behind the scenes. An even lower level tactic, `case`, takes a goal and replaces it with several goals, each one derived from one part of the inductive type.

We can make functions that act on an inductive type using pattern matching. For example,

```
  Definition mg_works (p : pdetype) :=
match p with
| hyperbolic => False
| other => True
end.
```

and evaluate them on input data

```
Coq < Eval compute in (mg_works elliptic).
     = True
     : Prop
```

We can use the `simpl` tactic to replace a function call with the result when using inductive types. For example, we can use it to prove the simple statement below, no_hyp_mg

```
Coq < Theorem no_hyp_mg : mg_works hyperbolic = False.
1 subgoal

  ============================
  mg_works hyperbolic = False

no_hyp_mg < simpl.
1 subgoal

  ============================
  False = False

no_hyp_mg < reflexivity.
No more subgoals.

no_hyp_mg < Qed.
(simpl).
reflexivity.

Qed.
no_hyp_mg is defined
```

Sometimes, we end up with statements about the equality of constructors in the inductive type. As a simple example, let us try to prove that

```
Coq < Lemma simple : ~ elliptic = hyperbolic.
1 subgoal

  ============================
  elliptic <> hyperbolic

simple < unfold not.
1 subgoal

  ============================
  elliptic = hyperbolic -> False

simple < intro H.
1 subgoal

  H : elliptic = hyperbolic
  ============================
  False
```

We need to show a contradiction here, and we can use the `discriminate` tactic, which examines the inductive type to see which equalities are possible.

```
simple < discriminate.
No more subgoals.

simple < Qed.
(unfold not).
intro H.
discriminate.

Qed.
simple is defined
```

We can explain the operation of `discriminate` by replacing our goal with a function that produces the same result using the *change* tactic. This replaces a term with another *convertible* term, meaning that the definitions are equal. This is the same process that the `simpl` tactic uses, but with `change` we can precisely control what will be substituted.

```
simple < change ((fun p:pdetype => match p with | elliptic => True | _ => False end) hyperbolic).
1 subgoal

  H : elliptic = hyperbolic
  ============================
  (fun p : pdetype =>
   match p with
   | elliptic => True
   | parabolic => False
   | hyperbolic => False
   end) hyperbolic
```

Now we can use the equality in hypothesis `H` to rewrite the goal, making the proof trivial.

```
simple < rewrite <- H.
1 subgoal

  H : elliptic = hyperbolic
  ============================
  True

simple < trivial.
No more subgoals.

simple < Qed.
(unfold not).
intro H.
(change
   ((fun p : pdetype => match p with
                     | elliptic => True
                     | _ => False
                     end) hyperbolic)).
(rewrite <- H).
trivial.
```

```
Qed.
simple is defined
```

2.3.2 Types with Recursion

The natural numbers are the archetypal example of an inductive type which uses recursion,

```
Coq < Print nat.
Inductive nat : Set := O : nat | S : nat -> nat
```

We obtain a natural number either by using 0, or by applying the successor function S to an existing natural number. Thus if we want to prove that some predicate holds for all natural numbers, we start by proving that it holds for 0, and then we prove that if it holds for some natural number n, it holds for its successor $n+1$. This inductive argument is encapsulated in the `nat_ind` function,

```
Coq < Print nat_ind.
nat_ind =
fun P : nat -> Prop => nat_rect P
     : forall P : nat -> Prop,
       P 0 -> (forall n : nat, P n -> P (S n)) -> forall n : nat, P n
```

As a simple example, let us prove that addition is associative. After introducing hypotheses, we use induction on x with the `elim` tactic, plus_assoc

```
Coq < Theorem plus_assoc : forall x y z:nat, (x+y)+z = x+(y+z).
1 subgoal

  ============================
  forall x y z : nat, x + y + z = x + (y + z)

plus_assoc < intros x y z.
1 subgoal

  x, y, z : nat
  ============================
  x + y + z = x + (y + z)

plus_assoc < elim x.
2 subgoals

  x, y, z : nat
  ============================
  0 + y + z = 0 + (y + z)

subgoal 2 is:
 forall n : nat, n + y + z = n + (y + z) -> S n + y + z = S n + (y + z)
```

The first goal produced just replaces x with 0, and we can simplify this expression easily (you could also use the `plus_O_n` lemma),

```
plus_assoc < simpl.
2 subgoals

  x, y, z : nat
  ============================
  y + z = y + z

subgoal 2 is:
 forall n : nat, n + y + z = n + (y + z) -> S n + y + z = S n + (y + z)
```

```
plus_assoc < reflexivity.
1 subgoal

  x, y, z : nat
  ============================
  forall n : nat, n + y + z = n + (y + z) -> S n + y + z = S n + (y + z)
```

Next, we introduce the induction hypothesis (which would have been done automatically if we used `induction`),

```
plus_assoc < intros n IHn.
1 subgoal

  x, y, z, n : nat
  IHn : n + y + z = n + (y + z)
  ============================
  S n + y + z = S n + (y + z)
```

and rewrite the goal so that we can apply it,

```
plus_assoc < rewrite plus_Sn_m.
1 subgoal

  x, y, z, n : nat
  IHn : n + y + z = n + (y + z)
  ============================
  S (n + y) + z = S n + (y + z)

plus_assoc < rewrite plus_Sn_m.
1 subgoal

  x, y, z, n : nat
  IHn : n + y + z = n + (y + z)
  ============================
  S (n + y + z) = S n + (y + z)

plus_assoc < rewrite plus_Sn_m.
1 subgoal

  x, y, z, n : nat
  IHn : n + y + z = n + (y + z)
  ============================
  S (n + y + z) = S (n + (y + z))

plus_assoc < rewrite IHn.
1 subgoal

  x, y, z, n : nat
  IHn : n + y + z = n + (y + z)
  ============================
  S (n + (y + z)) = S (n + (y + z))
```

making the final goal trivial.

```
plus_assoc < reflexivity.
No more subgoals.

plus_assoc < Qed.
(intros x y z).
(elim x).
 (simpl).
 reflexivity.

 (intros n IHn).
 (rewrite plus_Sn_m).
 (rewrite plus_Sn_m).
 (rewrite plus_Sn_m).
 (rewrite IHn).
```

Tactic	Purpose	Example Use
`(f x)`	Apply a function f to x	(H a)
`pose`	Define a local variable	pose (Ha := H a)
`symmetry`	Reverse terms in an equality	x = y
`reflexivity`	A quantity is equal to itself	G: x = x
`rewrite`	Substitute equal quantities	H: x*x + x*2 = x * (x + 2) / G: S (x*x + x*2) = S (x * (x + 2))
`discriminate`	Determine whether two member of an inductive type are equal	H: 0 = 1 / G: F
`inversion`	Determine how an expression with inductive types could have been constructed	n, m: `nat` / H: S n = S m / G: n = m

Table 2.1: Additional Coq tactics for Induction and Inductive Types

```
 reflexivity.

Qed.
plus_assoc is defined
```

2.4 Problems

Problem II.1 Using the truth tables for three predicates P, Q, and R, over the domain $\mathcal{D} = \{a, b, c\}$, where the first argument is the row of our truth table and the second is the column. For example, $P(b, c) = F$, whereas $P(c, b) = T$.

P	a	b	c
a	T	F	T
b	F	F	F
c	T	T	T

Q	a	b	c
a	F	T	F
b	T	F	T
c	T	T	T

R	a	b	c
a	F	F	F
b	F	F	F
c	F	F	F

Evaluate the truth values of the statements of predicate logic below, and given a one line explanation for the truth value.

1. $\exists x \forall y, P(y, x)$
2. $\exists x \forall y, P(x, y)$
3. $\exists x \exists y, Q(x, y)$
4. $\forall x \forall y, \neg R(x, y)$

Problem II.2 Using the truth tables for three predicates R, S, and T, over the domain $\mathcal{D} = \{x, y, z\}$, where the first argument is the row of our truth table and the second is the column. For example, $R(x, z) = T$, whereas $R(z, x) = F$.

R	x	y	z
x	F	T	T
y	T	F	T
z	F	T	T

S	x	y	z
x	F	F	F
y	F	T	F
z	F	F	F

T	x	y	z
x	T	F	T
y	F	F	T
z	T	F	T

Evaluate the truth values of the statements of predicate logic below, and given a one line explanation for the truth value.

1. $\forall x \forall y, \neg S(x, y)$
2. $\exists x \exists y, R(x, y)$
3. $\exists x \forall y, T(y, x)$
4. $\exists x \forall y, T(x, y)$

Problem II.3 Generate **Proof:** *ex_dist_or_bac*, demonstrating that the existential predicate distributes over disjunction,

$$(\exists y : A, Py) \vee (\exists y : A, Qy) \implies \exists x : A, Px \vee Qx \tag{2.4}$$

using the Coq syntax

```
Lemma ex_dist_or_bac : forall A : Set, forall P Q : A->Prop, (ex P) \/ (ex Q) -> (exists x:A, P x \/ Q x).
```

Note that the reverse of this implication is proved in the notes.

Problem II.4 Generate **Proof:** *fa_dist_and*, demonstrating that the universal quantifier distributes over conjunction,

$$(\forall x : A, Px \wedge Qx) \iff (\forall x : A, Px) \wedge (\forall x : A, Qx) \tag{2.5}$$

using the Coq syntax

```
Lemma fa_dist_and : forall A : Set, forall P Q : A->Prop, (forall x, P x) /\ (forall x, Q x) <-> (forall x : A, P x /\ Q x).
```

Problem II.5 We can construct a weaker version of De Morgan's Law for the existential quantifier, shown below. Generate **Proof:** *demorgan_exists* for the statement,

$$\exists x : A, \neg\neg Px \implies \neg\forall y : A, \neg Py \tag{2.6}$$

using the Coq syntax

```
Lemma demorgan_exists : forall A : Set, forall P : A->Prop, (exists x : A, ~~P x) -> ~(forall y : A, ~P y).
```

Problem II.6 Prove the Frobenius Rule

$$\exists x : A, Q \wedge P(x) \iff Q \wedge \exists x : A, P(x) \tag{2.7}$$

using the Coq syntax

```
Lemma frobenius : forall A : Set, forall P : A->Prop, forall Q : Prop, (exists x : A, Q /\ P x) <-> Q /\ (exists x : A, P x).
```

Problem II.7 Prove the common inference technique

$$\forall x : A, \exists y : A, \neg P(x,y) \implies \neg\exists x : A, \forall y : A, P(x,y) \tag{2.8}$$

using the Coq syntax

```
Lemma hodges : forall A : Set, forall P : A -> A -> Prop, (forall x : A, exists y : A, ~(P x y)) ->
  (~exists x : A, forall y : A, (P x y)).
```

which can be found in the delightful An Editor Recalls Some Hopeless Papers by Wilfred Hodges.

Problem II.8 Prove that the existential quantifier distributes over conjunction

$$\exists x : A, Px \wedge Qx \implies (\exists x : A, Px) \wedge (\exists x : A, Qx) \tag{2.9}$$

using the Coq syntax

```
Lemma ex_dist_and : forall A : Set, forall P Q : A -> Prop,
  (exists x : A, P x /\ Q x) -> (exists x : A, P x) /\ (exists x : A, Q x).
```

Problem II.9 Prove that the universal quantifier distributes over disjunction

$$(\forall x : A, Px) \vee (\forall x : A, Qx) \implies \forall x : A, Px \vee Qx \tag{2.10}$$

using the Coq syntax

```
Lemma fa_dist_or : forall A : Set, forall P Q : A -> Prop,
  (forall x : A, P x) \/ (forall x : A, Q x) -> (forall x : A, P x \/ Q x).
```

Problem II.10 Prove that the universal quantifier distributes over implication

$$(\forall x : A, Px \implies Qx) \implies (\forall x : A, Px) \implies (\forall x : A, Qx) \tag{2.11}$$

using the Coq syntax

```
Lemma fa_dist_imp : forall A : Set, forall P Q : A -> Prop,
  (forall x : A, P x -> Q x) -> (forall x : A, P x) -> (forall x : A, Q x).
```

Chapter 3

Sets, Relations, and Functions

THE AXIOM OF CHOICE ALLOWS YOU TO SELECT ONE ELEMENT FROM EACH SET IN A COLLECTION
AND HAVE IT *EXECUTED* AS AN EXAMPLE TO THE OTHERS.
MY MATH TEACHER WAS A BIG BELIEVER IN PROOF BY INTIMIDATION.

https://xkcd.com/982/

— Randall Munroe

3.1 Sets

A *set* is a collection of things, called *elements*, which compose it. A set A supports one predicate, *contains*, which we could write $C(A, x)$, but instead it is typical to write in infix notation $x \in A$, or its negation $x \notin A$. We will have a special notation for the set with no elements (there is only one), which we will call the *empty set* and denote $\emptyset$. Note that since this is the only predicate, it would be impossible to know if a set contained duplicate elements, so they do not, and the elements are not in any order. We say that two sets are equal if they contain the same elements. We can ask whether all the elements in a set B are in the set A, which we denote

$$B \subseteq A = \bigwedge_{x \in B} x \in A. \tag{3.1}$$

If $B \subseteq A$ but $B \neq A$, then we say that B is a *proper subset* of A and write $B \subset A$. Note that

$$A = B \iff B \subseteq A \wedge A \subseteq B \tag{3.2}$$

which says that equality means that any element of B is in A and any element of A is in B.

Clearly, $\forall A, \emptyset \subseteq A$. We will also have special notation for some sets which are used frequently, such as the set of natural numbers ($\mathbb{N}$), integers ($\mathbb{Z}$), rational numbers ($\mathbb{Q}$), and real numbers ($\mathbb{R}$). We will often add 'plus' to restrict a set to the positive elements, such as the set of positive integers $\mathbb{Z}^+$. The elements of a set can be arbitrary objects, and in particular can be other sets.

It might seem that sets are not a very powerful concept, however they were invented to lay the foundation for all other mathematics. For example, we can represent the integers with sets, called the von Neumann ordinals, in the following way

$$\begin{aligned}
0 &= \emptyset \\
1 &= \{\emptyset\} \\
2 &= \{\emptyset, \{\emptyset\}\} \\
3 &= \{\emptyset, \{\emptyset\}, \{\emptyset, \{\emptyset\}\}\} \\
4 &= \{\emptyset, \{\emptyset\}, \{\emptyset, \{\emptyset\}\}, \{\emptyset, \{\emptyset\}, \{\emptyset, \{\emptyset\}\}\}\}
\end{aligned}$$

where each number is the set of all numbers less than itself. Thus we could, in principle, reduce all of arithmetic to the simple operations of set theory, and this was indeed attempted in Principia Mathematica by Russell and Whitehead. However, this formalist dream ran aground on the beach of the Gödel Incompleteness Theorems. Nevertheless, set theory remains a fundamental area of mathematics.

The most basic way to define a set is to list the elements in the set explicitly. Suppose that we want the set S of odd natural numbers less than ten. We could write it as

$$S = \{1, 3, 5, 7, 9\}. \tag{3.3}$$

However, if we wanted all odd numbers less than one hundred, this is less convenient. Thus we also employ a notation which specifies a set by some predicate over a domain which is true for all set elements,

$$A = \{x \in \mathbb{N} \mid O(x) \wedge x < 100\}. \tag{3.4}$$

This is often called *set-builder* notation. We will often omit the domain when it is clear from context, or non-essential.

We will call the size of a set its *cardinality* and denote it $|A|$. For example, $|S| = 5$ and $|A| = 50$. But what does it mean to "count the elements" in a set? Counting has to do with the natural numbers, and sets are supposed to be more basic than those. Let us say that a set has size n is there is a one-to-one matching of the elements of the set to the elements of the set $\{0, 1, \ldots, n-1\}$, which we see is the set representing n in the von Neumann ordinals. Thus all sets with four elements can be matched one-to-one with the set $\{0, 1, 2, 3\}$. The beautiful thing about this definition for counting is that it generalizes cleanly to the case of an infinite number of elements, as discovered by Georg Cantor. For example, suppose we ask if the set of even numbers is smaller than the set of natural numbers. This seems very reasonable since we leave out all the odd numbers, however we have the correspondence

$$\begin{aligned} 0 &\iff 0 \\ 1 &\iff 2 \\ 2 &\iff 4 \\ &\vdots \\ n &\iff 2n \end{aligned}$$

showing us that the sets have the same size, since every natural number is matched to one and only one even number and similarly for the even numbers. The same argument can be made for odd numbers by matching n to $2n+1$. In Section 3.2.2, we will see that this matching divides sets into equivalence classes, and each class represents a cardinality.

Another way to construct sets is to build them from smaller sets. The *union* is a set which contains all the elements in the argument sets

$$A \cup B = \{x \mid x \in A \vee x \in B\}, \tag{3.5}$$

and it is no accident that its symbol is reminiscent of the or symbol ($\vee$). It dual is the *intersection*,

$$A \cap B = \{x \mid x \in A \wedge x \in B\}, \tag{3.6}$$

which is a set containing the elements in common between the argument sets. It is clear how to extend these operations to any number of sets

$$\bigcup_i A_i = \{x \mid \bigvee_i x \in A_i\} \tag{3.7}$$

$$\bigcap_i A_i = \{x \mid \bigwedge_i x \in A_i\}. \tag{3.8}$$

The intersection and union operations also obey many identities obeyed by the conjuction and disjunction operators, which are easy to prove starting with the logical relations. As with many set definitions, we start with a generic element x which is a member of the set. Consider the set $A \cup (B \cap C)$ and expand the definition using our set-builder notation,

$$A \cup (B \cap C) = \{x \mid x \in A \vee x \in (B \cap C)\}, \tag{3.9}$$

$$= \{x \mid x \in A \vee (x \in B \wedge x \in C)\}. \tag{3.10}$$

Now we can use the logical distributive relation,

$$\begin{aligned} A \cup (B \cap C) &= \{x \mid (x \in A \vee x \in B) \wedge (x \in A \vee x \in C)\}, && (3.11)\\ &= \{x \mid x \in (A \cup B) \wedge x \in (A \cup C)\}, && (3.12)\\ &= (A \cup B) \cap (A \cup C). && (3.13) \end{aligned}$$

There is also the analogue of negation for sets, called the *complement*,

$$\overline{A} = \{x \mid x \notin A\}, \tag{3.14}$$

which consists of all the elements in the domain that are not in A. Note that the domain is just some larger set $\mathcal{D}$, such that $A \subseteq \mathcal{D}$. Other operators which are sometimes used are

$$\begin{aligned} A - B &= \{x \mid x \in A \wedge x \notin B\} = A \cap \overline{B}, && (3.15)\\ A \oplus B &= \{x \mid x \in A \oplus x \in B\} = (A - B) \cup (B - A) && (3.16) \end{aligned}$$

Much reasoning about set operations can be reduced to propositional reasoning about the predicates defining the sets. For example, we wish to evaluate the proposition

$$A \oplus B \oplus A = A. \tag{3.17}$$

We can put it into set builder notation

$$\begin{aligned} A \oplus B \oplus A &= \{x \mid x \in A \oplus x \in B\} \oplus A && (3.18)\\ &= \{x \mid x \in A \oplus x \in B \oplus x \in A\} && (3.19) \end{aligned}$$

which we can evaluate using a truth table

C_A	C_B	$C_A \oplus C_B$	$C_A \oplus C_B \oplus C_A$
T	T	F	T
T	F	T	F
F	T	T	T
F	F	F	F

and thus the proposition is false, but clearly $A \oplus B \oplus A = B$ and also $A \oplus B \oplus B = A$.

3.1.1 Power Set

The *power set* of a set A, denoted $\mathcal{P}(A)$, is the set consisting of all possible subsets of A. For example, if $A = \{1, 2, 3\}$ then

$$\mathcal{P}(A) = \{\emptyset, \{1\}, \{2\}, \{3\}, \{1, 2\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\}\}. \tag{3.20}$$

The cardinality of the power set $|\mathcal{P}(A)|$ is $2^{|A|}$, since we can imagine creating it by taking the empty set and producing two sets from it, one containing the

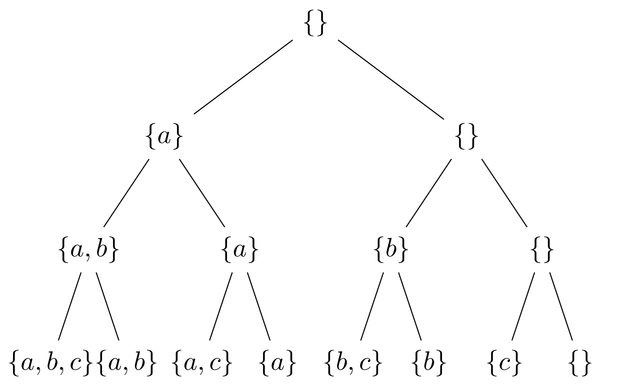


Figure 3.1: Construction of the power set for $\{a, b, c\}$.

first element and one that does not, and then repeating this procedure for each element on the new sets. Each decision, to include a given element or not, is like an edge in a binary tree and each leaf of the tree will be a subset. There are 2^l leaves in a binary tree of depth l, and we need to make a decision for each element, so our tree has $2^{|A|}$ leaves. The tree for our example set is shown in Fig. 3.1.

Clearly, the size of the power set $|\mathcal{P}(A)|$ is greater than the size of the initial set $|A|$ when A is a finite set. However, what happens when A is an infinite set? How do we compare the sizes of S and $\mathcal{P}(S)$ when $|S|$ is infinite, for instance when S is the set of natural numbers?

To attack this problem, we will start with the example of real numbers in $(0, 1)$. This may seem counterintuitive, but if we imagine that real numbers are represented by a finite or infinite series of digits, it seems clear that each real number is a sequence of natural numbers, corresponding to its digit series. Now suppose that we want to count the real numbers, so we line up each number, with the row being a real number, and the ith column being the ith digit of the number. We say that for each natural number i, $f(i)$ is a real number. Now we construct a new real number r such that it is different from the first number in the first digit, the second number in the second digit, and the ith number in the ith digit. If we indicate the ith digit of number x by x_i, then we have

$$r = \{r_i \in \mathbb{N} | r_i \neq f(i)_i\}. \tag{3.21}$$

or r is the collection of digits r_i such that its ith digit is not the same as that of $f(i)$. The number r cannot appear in our list, since it is always different in some position from our original list of numbers. We can see this by assuming that for some natural number j, $f(j) = r$. This means

$$\begin{aligned} f(j) = r &\implies \forall_k f(j)_k = r_k & (3.22)\\ &\implies f(j)_j = r_j & (3.23)\end{aligned}$$

$$
\begin{array}{c|cccccc}
1 & \mathbf{7} & 2 & 9 & 9 & 3 & \dots \\
2 & 8 & \mathbf{4} & 1 & 6 & 1 & \dots \\
3 & 1 & 5 & \mathbf{3} & 6 & 0 & \dots \\
4 & 5 & 2 & 5 & \mathbf{8} & 5 & \dots \\
\vdots & 7 & 9 & 2 & 4 & \mathbf{0} & \dots
\end{array}
$$

Figure 3.2: Construction of a number $r = 63279\dots$, guaranteed not to be in our infinite list of real numbers since it differs from each number at the diagonal digit.

but from the definition of r, we have that

$$f(j)_j \neq r_j. \tag{3.24}$$

This is a contradiction, and thus we reject the assumption that $\exists j, f(j) = r$. This is called a *diagonal argument* because we use the diagonal digit $f(i)_i$ to define our number. Note that for the definition of r, we had to choose one member each from an infinite array of sets to construct our new set. It turns out that this kind of construction employs the Axiom of Choice, which is independent of the rest of the axioms of set theory. What we have shown is that there are more real numbers than natural numbers, which might be a surprising thing, since they are both infinite sets.

A slightly more abstract diagonal argument was used by Cantor to prove that for every set S, the power set $\mathcal{P}(S)$ has a larger cardinality than S itself, which is known as Cantor's Theorem. We will prove the theorem by contradiction. First we use the definition of cardinality as a one-to-one map between sets, so consider the map f from S to $\mathcal{P}(S)$. If $\mathcal{P}(S)$ is larger than S, it means that some element T of $\mathcal{P}(S)$ is not equal to any $f(s), s \in S$. So we will assume the negation, namely that all elements of $\mathcal{P}(S)$ are mapped to by some element of S. Cantor's amazing insight was to explicitly construct a specific element which cannot be in the map. Consider the subset $T \subset S$,

$$T = \{s \in S \mid s \notin f(s)\}. \tag{3.25}$$

Our initial assumption means that there exists some $s \in S$ such that $f(s) = T$. But by construction of T, $s \in T \iff s \notin f(s) = T$. This is a contradiction, since it says that s is both in T and not in T. Thus our initial assumption, that all elements of $\mathcal{P}(S)$ are mapped to by some element of S, must be wrong. This shows that $\mathcal{P}(S)$ is not the same size as S. On the other hand, the singleton map o from S to $\mathcal{P}(S)$ defined by $s \mapsto \{s\}$ means that $|\mathcal{P}(S)| \geq S$. Consequently, we must have $|S| < |\mathcal{P}(S)|$.

Another way to explain this result is that for every $s \in S$, either s is in T or not. If s is in T, then by definition of T, s is not in $f(s)$, so T is not equal to $f(s)$. On the other hand, if s is not in T, then by definition of T, s is in $f(s)$, so again T is not equal to $f(s)$. We can imagine a table similar to our table for the real numbers. The columns correspond to set elements, and the rows

	a	b	c	d	e	...
1	**1**	0	1	1	1	...
2	0	**0**	1	0	1	...
3	0	1	**1**	1	0	...
4	1	0	0	**1**	0	...
⋮	0	1	0	1	**0**	...

Figure 3.3: Construction of a subset $T = \{b, e, \ldots\}$, guaranteed not to be in our infinite list of subsets since it differs from each subset at the diagonal member.

correspond to subsets. The table has a one if the element is contained in the subset, and a zero if not. Thus, each row of values, instead of corresponding to a number, is the indicator function for that subset. When we choose T to be different on the diagonal, we make its indicator function different from every subset in our list, exactly as we made the digit sequence for r different from every other real number in our list. In fact, if we expanded our real numbers in binary, it would be nearly identical.

3.2 Relations

A n-ary *relation* R can be defined as a predicate $R(x_1, \ldots, x_n)$ taking n arguments, although we will be most interested in binary relations taking two arguments. The graph of the relation, $G(R)$, is the set of all inputs, ordered tuples $(x_1, \ldots, x_n)$, for which the relation is true. It is called a relation because we often use this structure to relate a set of things, such as "things within one foot of this pole". However, it is easier to think about the relation between just two things.

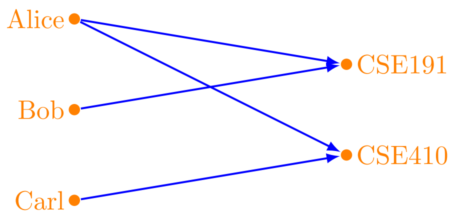
A binary relation R is usually expressed using infix notation, such as aRb. For example, the $<$ operator is a binary relation "is less than". So for our universe $\{1, 2, 3\}$, the graph of $<$ is

$$G(<) = \{(1,2), (1,3), (2,3)\}. \tag{3.26}$$

It is also possible to have different spaces for the domain and range of a relation. For example, suppose we have three students, {Alice, Bob, Carl}, and two classes, {CSE191, CSE410}. For the relation "is registered for", we could have the graph

$$\begin{aligned} G(R) = \{&(\mathrm{Alice}, \mathrm{CSE191}), (\mathrm{Bob}, \mathrm{CSE191}), \\ &(\mathrm{Alice}, \mathrm{CSE410}), (\mathrm{Carl}, \mathrm{CSE410})\}. \end{aligned} \tag{3.27}$$

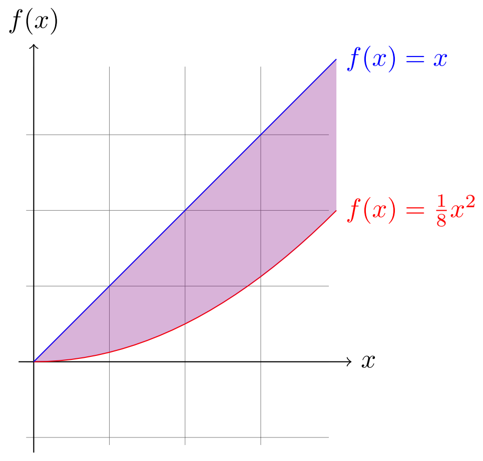
This makes it clear that the order of arguments matters, which we had seen before for multi-argument predicates. We can also express this graph as a diagram



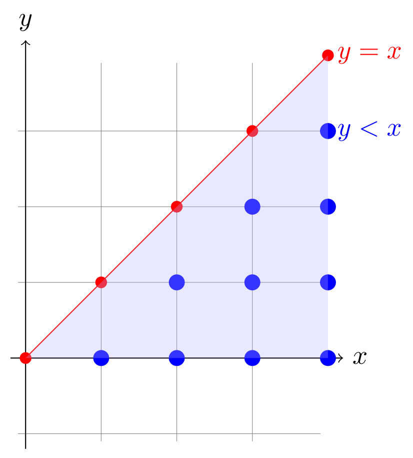
It is clear from the diagram that some inputs (Alice) can map to more than one output (CSE191, CSE410). This is what separates relations from functions, as we will see in Section 3.3.

We can think of the relation as a directed graph, if we let all the elements of the input sets be the vertices, and we insert an edge from x to y each time the proposition xRy is true. If the sets X and Y are different, as in our registration example, then this is a bipartite directed graph, meaning arrows start in one set of vertices and finish in another disjoint from the first. If instead the two sets are the same, $X = Y$, we can arrive at more general kinds of directed graphs, detailed below.

The *graph* of a relation is the set of elements from the product space $X \times Y$ which satisfy the relation. You are probably more familiar with the graph of a function, meaning the pairs (x, y) where $y = f(x)$. Scalar functions, where $Y = \mathbb{R}$, has graphs which are simple lines. For example, in the figure below we graph two functions, $y = x$ (blue) and $y = \frac{1}{8}x^2$ (red), where $X = [0, 4]$. The purple shared region is the graph of the relation $\frac{1}{8}x^2 < y < x$.



When drawing graph, we do not have to restrict ourselves to the domain of real numbers. For example, suppose that $X = Y = \mathbb{N}$, so that we are looking at relations among the natural numbers. We can make graphs in the same way, but they are not continuous. Below, we should the graph of the function $y = x$ (red) and also the relation $y < x$ (blue) for $X = \{0, 1, 2, 3, 4\}$.



When we think about the logical definition of a relation, it is a function taking in (x, y) and producing a proposition whose truth value tells us whether that point is included in the graph. We can formalize this definition (relation), in Coq

```
Definition relation (X Y : Type) := X -> Y -> Prop.
```

We will classify relations by their algebraic properties, which we will illustrate using common relations from arithmetic. A relation which is always true along the diagonal, $aRa = \mathrm{T}$, is called *reflexive*. For example, "less than or equal to" is reflexive since $x \leq x$, but "less than" is not since $x \not< x$. A relation is *symmetric* if $aRb \implies bRa$, so that "less than or equal" is not symmetric since $1 \leq 2$ but $2 \not\leq 1$, whereas "equal to" is symmetric since $x = y \implies y = x$. Another symmetric relation is "is a blood relative of". An *asymmetric* relation obeys $aRb \implies \neg bRa$, so that "less than" is asymmetric. An antisymmetric relation means that $aRb \wedge bRa \implies a = b$, so "less than or equal to" is antisymmetric since $x \leq y \wedge y \leq x \implies x = y$. Note that "less than" is also antisymmetric, but equality is always false (empty). Finally a relation is *transitive* if $aRb \wedge bRc \implies aRc$, so that "less than" is transitive since $a < b \wedge b < c \implies a < c$. Also "is an ancestor of" is transitive, but "is a parent of" is not.

Consider the following relations on our domain $\{1, 2, 3\}$,

$$\begin{aligned}
R_0 &= \{(1,1),(1,2),(2,1),(2,2),(3,1),(3,3)\} \\
R_1 &= \{(1,1),(1,3),(2,2),(3,1)\} \\
R_2 &= \{(2,3)\} \\
R_3 &= \{(1,1),(1,3)\}
\end{aligned}$$

- Which relations are reflexive?
  Only R_0 since it contains $\{(1,1),(2,2),(3,3)\}$.

- Which relations are symmetric?
  Only R_1. R_0 is missing $(1,3)$, R_2 is missing $(3,2)$, and R_3 is missing

$(3, 1)$. Notice that if we write R_1 as a Boolean matrix, then the matrix is symmetric.

- Which relations are asymmetric?
  Only R_2. All other relations have the diagonal element $(1, 1)$.
- Which relations are antisymmetric?
  R_2 and R_3 are antisymmetric. R_0 has both $(1, 2)$ and $(2, 1)$, and R_1 has both $(1, 3)$ and $(3, 1)$.
- Which relations are transitive?
  R_1, R_2 and R_3 are transitive. R_0 is missing $(3, 2)$.

These properties are manifested in the directed graphs we can draw to present the relation. For example, if the relation is symmetric, then every time we have an edge in one direction (xRy), there is a companion edge in the opposite direction (yRx), so that the graph is actually undirected. If the relation is transitive, every time we have a directed path that is made up of edges, say xRy–yRz–zRa–aRb, then we have an edge from the first to last vertex xRb. This means if any two vertices are connected by a path, they are connected by a single edge. If the relation is reflexive, then there is an edge from every vertex to itself, xRx, sometimes called a *self loop*.

We can formalize these definition using predicate calculus in Coq. For example, a reflexive relation

```
Definition reflexive {X: Type} (R: relation X X) := forall a : X, R a a.
```

and a transitive relation.

```
Definition transitive {X: Type} (R: relation X X) := forall a b c : X, (R a b) -> (R b c) -> (R a c).
```

We can define symmetry,

```
Definition symmetric {X: Type} (R: relation X X) := forall a b : X, (R a b) -> (R b a).
```

A relation R is antisymmetric if

$$Rab \implies Rba \implies a = b$$

meaning that there can be no two element *cycle*s, or loops in the graph.

```
Definition antisymmetric {X: Type} (R: relation X X) := forall a b : X, (R a b) -> (R b a) -> a = b.
```

If we suppose the relation is also transitive, then any cycle in graph of R can be reduced to a two element cycle, using transitvity to compress the path. Thus we will show in Section **??** transitive antisymmetric relations are acyclic, except for trivial cycles on a single element.

3.2.1 Partial Orders

A relation R on a set A is called a partial order if it is reflexive, transitive, and antisymmetric. The prototypical partial order is $\leq$ since we saw above that it falls into these classes, and in general we denote a partial order by $\preceq$. The set together with the relation, $(A, \preceq)$, is called a *poset*, short for partially ordered set. Why do we say "partially ordered"? In the case of $\leq$, the order is total, meaning either $x \leq y$ or $y \leq x$ or both is true. However we only require that our partial order be transitive and antisymmetric, not that all elements of A be comparable, so its possible that both $x \preceq y$ and $y \preceq x$ are false.

For example, consider $A = \mathcal{P}(\{1,2,3\})$ and $\preceq \;=\; \subseteq$. Is this a partial order? It is reflexive, since $S \subseteq S$ is true of any set. It is transitive, since $S \subseteq T \wedge T \subseteq U \implies S \subseteq U$, and it is antisymmetric since by our definition of set equality $S = T \iff T \subseteq S \wedge S \subseteq T$. However, it is not a total order because

$$\{1,2\} \nsubseteq \{2,3\} \wedge \{2,3\} \nsubseteq \{1,2\}, \tag{3.28}$$

so we say that $\{1,2\}$ and $\{2,3\}$ are incomparable.

Notice that the directed graph for a partial order cannot have a cycle. Suppose a cycle exists in the directed graph. Then there is a path from some vertex x to itself, going through another vertex y. Since there is a path from x to y, there must be an edge xRy since the relation is transitive. Since there is a cycle, there is also a path from y to x, and thus an edge yRx. Having both of these edges implies that $x = y$ because the relation is antisymmetric, but we assumed at the beginning that $x! = y$, so the original assumption of a cycle must be wrong. A directed graph with no cycles is called a directed acyclic graph or DAG, and these characterize partial orders.

The prototypical partial order (also a total order) is the "less than or equal to" relation. We can define this using an inductive predicate

```
Inductive le (n : nat) : nat -> Prop :=
  le_n : (le n n)
| le_S : forall m : nat, (le n m) -> (le n (S m)).
```

We can prove some simple things about this order, using the theorems in the inductive definition. For example, a number n is always less than or equal to its successor Sn, which we will call n_le_Sn,

```
Coq < Lemma n_le_Sn : forall n, (le n (S n)).
1 subgoal

  ============================
  forall n : nat, le n (S n)

n_le_Sn < intro.
1 subgoal

  n : nat
  ============================
  le n (S n)

n_le_Sn < apply le_S.
1 subgoal

  n : nat
  ============================
  le n n

n_le_Sn < apply le_n.
No more subgoals.
```

```
n_le_Sn < Qed.
intro.
(apply le_S).
(apply le_n).

Qed.
n_le_Sn is defined
```

We can formalize the notion of partial order

```
  Definition order {X:Type} (R: relation X X) := (reflexive R) /\ (antisymmetric R) /\ (transitive R).
```

and try to prove that `le` is one. It is straightforward to prove that `le` is le_reflexive,

```
Coq < Theorem le_reflexive : reflexive le.
1 subgoal

  ============================
  reflexive le

le_reflexive < unfold reflexive.
1 subgoal

  ============================
  forall a : nat, le a a

le_reflexive < intro a.
1 subgoal

  a : nat
  ============================
  a <= a

le_reflexive < apply le_n.
No more subgoals.

le_reflexive < Qed.
(unfold reflexive).
intro a.
(apply le_n).

Qed.
le_reflexive is defined
```

To prove that it is transitive, theorem le_trans, we first introduce the hypotheses

```
Coq < Theorem le_trans : transitive le.
1 subgoal

  ============================
  transitive le

le_trans < unfold transitive.
1 subgoal

  ============================
  forall a b c : nat, le a b -> le b c -> le a c

le_trans < intros a b c Hab Hbc.
1 subgoal

  a, b, c : nat
  Hab : le a b
  Hbc : le b c
  ============================
  le a c
```

and then carry out induction on the `Hbc` hypothesis. We can see that this makes sense by looking at the induction theorem for `le`,

```
Coq < Print le_ind.
le_ind =
fun (n : nat) (P : nat -> Prop) (f : P n)
  (f0 : forall m : nat, le n m -> P m -> P (S m)) =>
fix F (n0 : nat) (l : le n n0) {struct l} : P n0 :=
  match l in (le _ n1) return (P n1) with
  | le_n _ => f
  | le_S _ m l0 => f0 m l0 (F m l0)
  end
     : forall (n : nat) (P : nat -> Prop),
       P n ->
       (forall m : nat, le n m -> P m -> P (S m)) ->
       forall n0 : nat, le n n0 -> P n0
```

Suppose that my predicate `P(n) = le a n`, then what we want to prove is that

```
  forall c : nat, le b c -> P c
```

noting that we also have `(le a b)` as a hypothesis. The induction theorem tells us that we can prove that, using `n = b` and `n0 = c`, if we can prove `P(b) = le a b`, which we know, and

```
  forall m : nat, le n m -> P m -> P (S m)
```

which after plugging in becomes

```
  forall m : nat, le b m -> le a m -> le a (S m)
```

The proof assistant will handle all of these details smoothly. We start with our base case just `P(b) = le a b`,

```
le_trans < induction Hbc.
2 subgoals

  a, b : nat
  Hab : le a b
  ============================
  le a b

subgoal 2 is:
 le a S m

le_trans < assumption.
1 subgoal

  a, b : nat
  Hab : le a b
  m : nat
  Hbc : le b m
  IHHbc : le a m
  ============================
  le a S m
```

The induction step, `le n m -> P m -> P (S m)`, can be reduced to our induction hypothesis using the inductive definition of `le`.

```
le_trans < apply le_S.
1 subgoal

  a, b : nat
  Hab : le a b
  m : nat
  Hbc : le b m
  IHHbc : le a m
  ============================
  le a m

le_trans < assumption.
No more subgoals.

le_trans < Qed.
(unfold transitive).
(intros a b c Hab Hbc).
(induction Hbc).
 assumption.

 (apply le_S).
 assumption.

Qed.
le_trans is defined
```

Moreover, we can prove that if successors of two numbers are related, then the numbers themselves are related, which we will call Sn_le_Sm__n_le_m. We start by introducing hypotheses, and then we use the inversion tactic on the main hypothesis. This tells us that there are two ways to construct this expression inductively.

```
Coq < Lemma Sn_le_Sm__n_le_m : forall n m, le (S n) (S m) -> le n m.
1 subgoal

  ============================
  forall n m : nat, le (S n) (S m) -> le n m

Sn_le_Sm__n_le_m < intros n m HSnm.
1 subgoal

  n, m : nat
  HSnm : le (S n) (S m)
  ============================
  le n m

Sn_le_Sm__n_le_m < inversion HSnm.
2 subgoals

  n, m : nat
  HSnm : le (S n) (S m)
  H0 : n = m
  ============================
  le m m

subgoal 2 is:
 le n m
```

First, the two numbers could be equal, so that `le_n` applies

```
Sn_le_Sm__n_le_m < apply le_n.
1 subgoal

  n, m : nat
  HSnm : le (S n) (S m)
  m0 : nat
  H0 : le (S n) m
  H : m0 = m
  ============================
  le n m
```

Second, it could be the case that $Sn \leq m$ (since Sm was larger than Sn). We can apply transitivity of `le` with $a = n$, $b = Sn$ and $c = m$, which says that

```
  le n (S n) -> le (S n) m -> le n m
```

and generates two subproofs. However, we already have `le (S n) m` as a hypothesis, and we have proved `le n (S n)` above.

```
Sn_le_Sm__n_le_m < apply le_trans with (b := S n).
2 subgoals

  n, m : nat
  HSnm : le (S n) (S m)
  m0 : nat
  H0 : le (S n) m
  H : m0 = m
  ============================
  le n (S n)

subgoal 2 is:
 le (S n) m

Sn_le_Sm__n_le_m < apply n_le_Sn.
1 subgoal

  n, m : nat
  HSnm : le (S n) (S m)
  m0 : nat
  H0 : le (S n) m
  H : m0 = m
  ============================
  le (S n) m

Sn_le_Sm__n_le_m < apply H0.
No more subgoals.

Sn_le_Sm__n_le_m < Qed.
(intros n m HSnm).
(inversion HSnm).
 (apply le_n).

 (apply le_trans with (b := S n)).
  (apply n_le_Sn).

  (apply H0).

Qed.
```

We have defined antisymmetry of a relation in Eq. 3.28, so we start our proof, le_antisymmetric, with induction on a.

```
Coq < Theorem le_antisymmetric : antisymmetric le.
1 subgoal

  ============================
  antisymmetric le

le_antisymmetric < unfold antisymmetric.
1 subgoal

  ============================
  forall a b : nat, le a b -> le b a -> a = b

le_antisymmetric < induction a.
2 subgoals

  ============================
  forall b : nat, le 0 b -> le b 0 -> 0 = b

subgoal 2 is:
 forall b : nat, le (S a) b -> le b (S a) -> S a = b
```

For the base case, we introduce hypotheses, and then we analyze the hypothesis `b <= 0`. There is only one possibility for the inductive predicate, namely that *b* is zero. We use the inversion tactic to evaluate the cases, and then reflexivity to obtain the conclusion.

```
le_antisymmetric < intros b H0b Hb0.
2 subgoals

  b : nat
  H0b : le 0 b
  Hb0 : le b 0
  ============================
  0 = b

subgoal 2 is:
 forall b : nat, le (S a) b -> le b (S a) -> S a = b

le_antisymmetric < inversion Hb0.
2 subgoals

  b : nat
  H0b : le 0 b
  Hb0 : le b 0
  H : b = 0
  ============================
  0 = 0

subgoal 2 is:
 forall b : nat, le (S a) b -> le b (S a) -> S a = b

le_antisymmetric < reflexivity.
1 subgoal

  a : nat
  IHa : forall b : nat, le a b -> le b a -> a = b
  ============================
  forall b : nat, le (S a) b -> le b (S a) -> S a = b
```

For the inductive step, we begin by introducing hypotheses, and then we make use of the definition of natural numbers. We use the `destruct` tactic on the natural number *b*, which distinguishes two cases. First, *b* could be zero, and then we can use the inversion tactic on hypothesis `HSab`.

```
le_antisymmetric < intros b HSab HbSa.
1 subgoal

  a : nat
  IHa : forall b : nat, le a b -> le b a -> a = b
  b : nat
  HSab : le (S a) b
  HbSa : le b (S a)
  ============================
  S a = b

le_antisymmetric < destruct b.
2 subgoals

  a : nat
  IHa : forall b : nat, le a b -> le b a -> a = b
  HSab : le (S a) 0
  HbSa : le 0 (S a)
  ============================
  S a = 0

subgoal 2 is:
 S a = S b

le_antisymmetric < inversion HSab.
1 subgoal

  a : nat
  IHa : forall b : nat, le a b -> le b a -> a = b
  b : nat
  HSab : le (S a) (S b)
  HbSa : le (S b) (S a)
  ============================
  S a = S b
```

Second b can be the successor of some number, and this is very good for us, since now we recognize in the hypotheses the relation between successors which we addressed above. Thus, we can apply the previous theorem twice,

```
le_antisymmetric < apply Sn_le_Sm__n_le_m in HSab.
1 subgoal

  a : nat
  IHa : forall b : nat, le a b -> le b a -> a = b
  b : nat
  HSab : le a b
  HbSa : le (S b) (S a)
  ============================
  S a = S b

le_antisymmetric < apply Sn_le_Sm__n_le_m in HbSa.
1 subgoal

  a : nat
  IHa : forall b : nat, le a b -> le b a -> a = b
  b : nat
  HSab : le a b
  HbSa : le b a
  ============================
  S a = S b
```

and this allows us to apply our induction hypothesis, rewrite the conclusion using the new equality, and apply reflexivity.

```
le_antisymmetric < apply IHa in HSab.
2 subgoals

  a : nat
  IHa : forall b : nat, le a b -> le b a -> a = b
  b : nat
  HSab : a = b
  HbSa : le b a
  ============================
  S a = S b

subgoal 2 is:
 le b a

le_antisymmetric < rewrite HSab.
2 subgoals

  a : nat
  IHa : forall b : nat, le a b -> le b a -> a = b
  b : nat
  HSab : a = b
  HbSa : le b a
  ============================
  S b = S b

subgoal 2 is:
 le b a

le_antisymmetric < reflexivity.
1 subgoal

  a : nat
  IHa : forall b : nat, le a b -> le b a -> a = b
  b : nat
  HSab : le a b
  HbSa : le b a
  ============================
  le b a
```

Now all that remains to be done is note that the last assumption of the induction hypothesis was already one of our hypotheses.

```
le_antisymmetric < exact HbSa.
No more subgoals.

le_antisymmetric < Qed.
(unfold antisymmetric).
(induction a).
 (intros b H0b Hb0).
 (inversion Hb0).
 reflexivity.

 (intros b HSab HbSa).
 (destruct b).
  (inversion HSab).

  (apply Sn_le_Sm__n_le_m in HSab).
  (apply Sn_le_Sm__n_le_m in HbSa).
  (apply IHa in HSab).
   (rewrite HSab).
   reflexivity.

   (exact HbSa).

Qed.
```

Now we can use our previous results to prove the main theorem, le_order,

```
Coq < Theorem le_order : order le.
1 subgoal

  ============================
  order le

le_order < unfold order.
1 subgoal

  ============================
  reflexive le /\ antisymmetric le /\ transitive le

le_order < split.
2 subgoals

  ============================
  reflexive le

subgoal 2 is:
 antisymmetric le /\ transitive le
```

```
le_order < apply le_reflexive.
1 subgoal

  ============================
  antisymmetric le /\ transitive le

le_order < split.
2 subgoals

  ============================
  antisymmetric le

subgoal 2 is:
 transitive le

le_order < apply le_antisymmetric.
1 subgoal

  ============================
  transitive le

le_order < apply le_trans.
No more subgoals.

le_order < Qed.
(unfold order).
split.
 (apply le_reflexive).

 split.
  (apply le_antisymmetric).

  (apply le_trans).

Qed.
le_order is defined
```

3.2.2 Equivalence Relations

A relation R on a set A is called an equivalence relation if it is reflexive, transitive, and symmetric, and is denoted $a \sim b$. The prototypical equivalence relation is $=$, but consider the relation "has the same birthday as". It is reflexive since a person always has the same birthday as themselves. It is transitive, since if Alice and Bob have the same birthday, and Bob and Carl have the same birthday, then Alice and Carl have the same birthday. And its symmetric, since if Alice and Bob have the same birthday, so do Bob and Alice. Notice that underneath, we are using the properties of $=$ for the dates.

```
  Definition equivalence {X:Type} (R: relation X X) := (reflexive R) /\ (symmetric R) /\ (transitive R).
```

3.3 Functions

A *function* is a relation with a distinguished last argument and the restriction that for each combination of leading elements, there is a single last argument for which the function is true. For example, our "is registered for" relation is not a function because its graph

$$G(R) = \{(\text{Alice}, \text{CSE191}), (\text{Bob}, \text{CSE191}), (\text{Alice}, \text{CSE410}), (\text{Carl}, \text{CSE410})\}.$$

has two entries for input Alice. It is more conventional to think of a function as a map f between the domain $\mathcal{D}$ and codomain $\mathcal{R}$, denoted $f : \mathcal{D} \to \mathcal{R}$ with individual values denoted $f(x) = y$ for $x \in \mathcal{D}$ and $y \in \mathcal{R}$, instead of the infix xfy. The range is the subset of $\mathcal{R}$ covered by the function, namely the set

$$\{y \in \mathcal{R} \mid \exists x \in \mathcal{D}, f(x) = y\}. \tag{3.29}$$

We can formulate the function requirement as a statement of predicate logic. A mapping $f : \mathcal{D} \to \mathcal{R}$ is a function if and only if

$$\forall x \in \mathcal{D}, \exists y \in \mathcal{R}, f(x) = y \wedge (\forall z \in \mathcal{R}, y \neq z \implies f(x) \neq z)\,. \tag{3.30}$$

The first part of this statement guarantees that f maps every element in the domain to an element in the codomain, The second part guarantees that f does not map x to multiple elements in the codomain. Alternatively, we can say

```
Definition partial_function {X Y: Type} (R: relation X Y) :=
  forall x : X, forall y1 y2 : Y, R x y1 -> R x y2 -> y1 = y2.
```

Here we say partial_function because it might be that the relation R does not map some x to anything, which we disallowed in our function definition. This is just the negation of our previous definition. In some cases, we will need to know that our relation is defined over the entire domain, so we will also define a total_function,

```
Definition total_function {X Y: Type} (R: relation X Y) :=
  partial_function R /\ (forall x : X, exists y : Y, R x y).
```

Consider the sets $X = \mathbb{Z}$ and $Y = \mathbb{Z}$, and the mapping $f : X \to Y$ where

$$f(x) = \begin{cases} x & \text{if } x \text{ is odd} \\ x^2 & \text{if } x \geq 0 \\ |x| & \text{if } x < 0 \end{cases} \tag{3.31}$$

f is not a function, because all positive odd numbers (except 1) are mapped to multiple elements in the range, e.g. $f(3) = 3$ and $f(3) = 9$.

In the same way that we classified relations by properties, we can classify functions. The function we used for determining cardinality needed to be "one-to-one", which we term *injective*, meaning

$$\forall x_1, x_2 \in \mathcal{D}, f(x_1) = f(x_2) \implies x_1 = x_2, \tag{3.32}$$

or no two inputs have the same output (injective)

```
Definition injective {X Y : Type} (R: relation X Y) :=
  forall x1 x2: X, forall y : Y, (R x1 y) -> (R x2 y) -> x1 = x2.
```

A function is called *surjective*, or “onto”,

$$\forall y \in \mathcal{R}, \exists x \in \mathcal{D}, f(x) = y. \tag{3.33}$$

if it covers the entire codomain (surjective)

```
  Definition surjective {X Y : Type} (R: relation X Y) := forall y : Y, exists x : X, R x y.
```

A function which is both one-to-one and onto is called *bijective*

```
  Definition bijective {X Y : Type} (R: relation X Y) := total_function R /\ injective R /\ surjective R.
```

A bijective function f has an inverse f^{-1} defined by

$$\forall x \in \mathcal{D}, f^{-1}(f(x)) = x. \tag{3.34}$$

Our counting procedure basically says that if a bijection exists between two sets A and B, then $|A| = |B|$. We can define the inverse formally

```
  Definition inverse {X Y : Type} (R : relation X Y) (S : relation Y X) := forall (x : X) (y : Y), R x y = S y x.
```

and prove the existence in Coq, inverse_exists. We use injectivity of R to prove that S is a partial function, and surjectivity of R to prove that it is indeed a total function.

```
Coq < Lemma inverse_exists {X Y : Type} : forall R : relation X Y, bijective R -> exists S : relation Y X,
        inverse R S /\ total_function S.
1 subgoal

  X : Type
  Y : Type
  ============================
  forall R : relation X Y,
  bijective R -> exists S : relation Y X, inverse R S /\ total_function S
inverse_exists < Qed.
(intros R Rbij).
(pose (S := fun (y : Y) (x : X) => R x y)).
exists S.
split.
 (unfold inverse).
 (intros x y).
 reflexivity.

 (unfold total_function).
 split.
  (unfold partial_function).
  (intros x y1 y2).
  (intros H1 H2).
  (unfold bijective in Rbij).
  (destruct Rbij as [Rfunc Rinj Rsur]).
  (destruct Rinj as [Rinj Rsur]).
  (unfold injective in Rinj).
  (apply Rinj with (x1 := y1) (x2 := y2) (y := x) in H1 as H3).
   exact H3.

   exact H2.

  intro x.
  (unfold bijective in Rbij).
  (destruct Rbij as [Rfunc Rinj]).
  (destruct Rinj as [Rinj Rsur]).
  (unfold surjective in Rsur).
  (destruct (Rsur x) as [y H1]).
  exists y.
  exact H1.

Qed.
inverse_exists is defined
```

Another important function construction is *composition*, which means feeding the output of one function as the input to another. So for functions $f : X \to Y$ and $g : Y \to Z$, we have

$$f \circ g : X \to Z := g(f(x)), \tag{3.35}$$

and formally (composition)

```
  Definition composition {X Y Z : Type} (R : relation X Z) (S : relation Z Y) (T : relation X Y) :=
    forall (x : X) (y : Y), (exists z : Z, R x z /\ S z y) <-> T x y.
```

We can again prove that such a function exists and is a total function (comp_exists).

```
  Lemma comp_exists {X Y Z : Type} : forall (R : relation X Z) (S : relation Z Y),
    bijective R -> bijective S -> exists T : relation X Y, composition R S T /\ total_function T.

comp_exists < Qed.
(intros R S RBij SBij).
(pose (T := fun (x : X) (y : Y) => exists z : Z, R x z /\ S z y)).
exists T.
split.
 (unfold composition).
 (intros x y).
 split.
  intro H1.
  (destruct H1 as [z H1]).
  exists z.
  exact H1.

  intro H1.
  (destruct H1 as [z H1]).
  exists z.
  exact H1.

 (unfold total_function).
 split.
  (unfold partial_function).
  (intros x y1 y2 H1 H2).
  (destruct H1 as [z1 H1]).
  (destruct H2 as [z2 H2]).
  (destruct H1 as [H1R H1S]).
  (destruct H2 as [H2R H2S]).
  (destruct RBij as [RTot RInjSur]).
  (destruct SBij as [STot SInjSur]).
  (destruct RTot as [RPartial RTot]).
  (destruct STot as [SPartial STot]).
  (unfold partial_function in RPartial).
  (apply RPartial with (y1 := z1) (y2 := z2) in H1R as H3R).
   (rewrite <- H3R in H2S).
   (unfold partial_function in SPartial).
   (apply SPartial with (y1 := y1) (y2 := y2) in H1S as H3S).
    exact H3S.

    exact H2S.

   exact H2R.

  intro x.
  (destruct RBij as [RFunc RInjSur]).
  (destruct RFunc as [RPartial RTot]).
  (destruct (RTot x) as [z H1R]).
  (destruct SBij as [SFunc SInjSur]).
  (destruct SFunc as [SPartial STot]).
  (destruct (STot z) as [y H1S]).
  exists y.
  exists z.
  split.
   exact H1R.

   exact H1S.

Qed.
comp_exists is defined
```

3.3.1 Examples

Before we start with the examples, we should go over some proof techniques that will help us solve the problems in the end of the chapter. When considering injectivity, or the lack of it, very often we must pick out a proposition from the universal quantification that defines it. For example, suppose I would like to prove that the natural numbers are not all equal to one (notone). This is obvious, but how do I prove that it is true? The proof begins simply by introducing our main hypothesis.

```
Coq < Lemma notone : ~ forall n: nat, n = 1.
1 subgoal

  ============================
  ~ (forall n : nat, n = 1)

notone < intro H.
1 subgoal

  H : forall n : nat, n = 1
  ============================
  False
```

Now what we want to do is pick out a particular proposition from `H` which is false, generating a contradiction. Suppose we choose the proposition `0 = 1`. We can pick this out from `H` using the `pose` tactic. Then all we have to do is use `discriminate` to let Coq know that the equality is impossible.

```
notone < pose (H0 := H 0).
1 subgoal

  H : forall n : nat, n = 1
  H0 := H 0 : 0 = 1
  ============================
  False

notone < discriminate.
No more subgoals.

notone < Qed.
intro H.
(pose (H0 := H 0)).
discriminate.

Qed.
notone is defined
```

We can attack this problem in a slightly different way by instead proving an absurd implication. We begin the same way, but then we use `assert` to introduce a statement, which we then have to prove as a subgoal.

```
Coq < Lemma notone2 : ~ forall n: nat, n = 1.
1 subgoal

  ============================
  ~ (forall n : nat, n = 1)

notone2 < intro H.
1 subgoal

  H : forall n : nat, n = 1
  ============================
  False

notone2 < assert (Absurd : 0 = 1 -> False).
2 subgoals

  H : forall n : nat, n = 1
  ============================
  0 = 1 -> False

subgoal 2 is:
 False
```

After we prove the subgoal using the same `discriminate` tactic, we can apply out assertion and then apply the main hypothesis `H`.

```
notone2 < intro Absurd.
2 subgoals

  H : forall n : nat, n = 1
  Absurd : 0 = 1
  ============================
  False

subgoal 2 is:
 False

notone2 < discriminate.
1 subgoal

  H : forall n : nat, n = 1
  Absurd : 0 = 1 -> False
  ============================
  False

notone2 < apply Absurd.
1 subgoal

  H : forall n : nat, n = 1
  Absurd : 0 = 1 -> False
  ============================
  0 = 1

notone2 < apply H.
No more subgoals.

notone2 < Qed.
intro H.
(assert (Absurd : 0 = 1 -> False)).
 intro Absurd.
 discriminate.

 (apply Absurd).
 (apply H).

Qed.
notone2 is defined
```

We can see that the two proofs are slightly different, but they rely on the same ingredients, namely the induction rule `False_ind` which we use when we have a contradiction, and `eq_ind` which is used by `discriminate` to check if two terms of the same inductive type are equal.

```
Coq < Print notone.
notone =
fun H : forall n : nat, n = 1 =>
let H0 := H 0 : 0 = 1 in
let H1 :=
  eq_ind 0 (fun e : nat => match e with
                         | 0 => True
                         | S _ => False
                         end) I 1 H0
  :
  False in
False_ind False H1
     : ~ (forall n : nat, n = 1)

Coq < Print notone2.
notone2 =
fun H : forall n : nat, n = 1 =>
let Absurd :=
  (fun Absurd : 0 = 1 =>
   let H0 :=
     eq_ind 0 (fun e : nat => match e with
                            | 0 => True
                            | S _ => False
                            end) I 1 Absurd
     :
     False in
   False_ind False H0)
  :
  0 = 1 -> False in
Absurd (H 0)
     : ~ (forall n : nat, n = 1)
```

Now we would like to prove something slightly more difficult, namely that there is no natural number `c` such that all others are equal to it (notconstant). We introduce the main hypothesis and then ask for the witness from the existential quantifier.

```
Coq < Lemma notconstant : ~ exists c : nat, forall n : nat, n = c.
1 subgoal

  ============================
  ~ (exists c : nat, forall n : nat, n = c)

notconstant < intro H.
1 subgoal

  H : exists c : nat, forall n : nat, n = c
  ============================
  False

notconstant < destruct H as [c Hc].
1 subgoal

  c : nat
  Hc : forall n : nat, n = c
  ============================
  False
```

Now we again need to pick out the false proposition from the universal quantifier. What natural number is guaranteed to be different from `c`? How about the successor of `c`, which we pick out again using the `pose` tactic and reverse to make it look a little nicer.

```
notconstant < pose (HSc := Hc (S c)).
1 subgoal

  c : nat
  Hc : forall n : nat, n = c
  HSc := Hc (S c) : S c = c
  ============================
  False

notconstant < symmetry in HSc.
1 subgoal

  c : nat
  Hc : forall n : nat, n = c
  HSc : c = S c
  ============================
  False
```

It turns out that the fact that no number is equal to its successor is a theorem built-in to Coq. We can use this to finish our proof. We will see later that it can be proved by induction.

```
notconstant < Print n_Sn.
Fetching opaque proofs from disk for Coq.Init.Peano
n_Sn =
fun n : nat =>
nat_ind (fun n0 : nat => n0 <> S n0) (O_S 0)
  (fun (n0 : nat) (IHn : n0 <> S n0) => not_eq_S n0 (S n0) IHn) n
     : forall n : nat, n <> S n

Argument scope is [nat_scope]

notconstant < apply n_Sn in HSc.
1 subgoal

  c : nat
  Hc : forall n : nat, n = c
  HSc : False
  ============================
  False

notconstant < exact HSc.
No more subgoals.

notconstant < Qed.
intro H.
(destruct H as [c Hc]).
(pose (HSc := Hc (S c))).
symmetry in HSc.
(apply n_Sn in HSc).
exact HSc.

Qed.
notconstant is defined
```

If you are wondering how to prove that theorem (n_not_Sn), we can use induction and the axiom that the successor function is injective.

```
Coq < Lemma n_not_Sn : forall n : nat, n <> S n.
1 subgoal

  ============================
  forall n : nat, n <> S n

n_not_Sn < Qed.
(induction n).
 intro H.
 discriminate.

 intro H.
 (apply IHn).
 (apply PeanoNat.Nat.succ_inj in H).
 exact H.

Qed.
n_not_Sn is defined
```

For the homework, we will also need a lemma (two_not_sq) which is not simple to prove by hand. First we can use the nonlinear integer arithmetic tactic (there is a nice example here).

```
Coq < Lemma two_not_sq : forall x : nat, x*x <> 2.
1 subgoal

============================
forall x : nat, x * x <> 2

two_not_sq < Require Import Psatz.

two_not_sq < intro x.
1 subgoal

x : nat
============================
x * x <> 2

two_not_sq < destruct x.
2 subgoals

============================
0 * 0 <> 2

subgoal 2 is:
S x * S x <> 2

two_not_sq < discriminate.
1 subgoal

x : nat
============================
S x * S x <> 2

two_not_sq < destruct x.
2 subgoals

============================
1 * 1 <> 2

subgoal 2 is:
S (S x) * S (S x) <> 2

two_not_sq < discriminate.
1 subgoal

x : nat
```

```
============================
S (S x) * S (S x) <> 2

two_not_sq < destruct x.
2 subgoals

============================
2 * 2 <> 2

subgoal 2 is:
S (S (S x)) * S (S (S x)) <> 2

two_not_sq < discriminate.
1 subgoal

x : nat
============================
S (S (S x)) * S (S (S x)) <> 2

two_not_sq < nia.
No more subgoals.

two_not_sq < Qed.
Require Import Psatz.
intro x.
(destruct x).
discriminate.

(destruct x).
discriminate.

(destruct x).
discriminate.

nia.

Qed.
two_not_sq is defined
```

However, we can show this by hand by pushing a little further.

```
two_not_sq < discriminate.
1 subgoal

x : nat
============================
S (S (S x)) * S (S (S x)) <> 2

two_not_sq < intro H.
1 subgoal

x : nat
H : S (S (S x)) * S (S (S x)) = 2
============================
False

two_not_sq < simpl in H.
1 subgoal

x : nat
H : S (S (S (x + S (S (S (x + S (S (S (x + x * S (S (S x)))))))))))) = 2
============================
False

two_not_sq < inversion H.
No more subgoals.

Qed.
two_not_sq is defined
```

The `2` on the right hand side is made using two calls to `S`, whereas the left hand side has three calls to `S`, so we know that they cannot be equal.

A different approach can be taken to proving statements about functions which requires some theorems about the division on numbers based on a total order. We can define a *trichotomy*, or a division into three parts, based on the "less than" order. We start out the proof (trichotomy) by introducing variables and beginning an induction on `m`,

```
Coq < Theorem trichotomy : forall n m : nat, n < m \/ n = m \/ m < n.
1 subgoal

  ============================
  forall n m : nat, n < m \/ n = m \/ m < n

trichotomy < intros n m.
1 subgoal

  n, m : nat
  ============================
  n < m \/ n = m \/ m < n

trichotomy < induction m.
2 subgoals

  n : nat
  ============================
  n < 0 \/ n = 0 \/ 0 < n

subgoal 2 is:
 n < S m \/ n = S m \/ S m < n
```

Obviously different cases can be true for different values of `n`, so we have to divide them up. Since `n` is an inductive type, we can split the proof into cases using `destruct`, and for natural numbers there are two possible cases. The first case is trivial.

```
trichotomy < destruct n.
3 subgoals

  ============================
  0 < 0 \/ 0 = 0 \/ 0 < 0

subgoal 2 is:
 S n < 0 \/ S n = 0 \/ 0 < S n
subgoal 3 is:
 n < S m \/ n = S m \/ S m < n

trichotomy < right.
3 subgoals

  ============================
  0 = 0 \/ 0 < 0

subgoal 2 is:
 S n < 0 \/ S n = 0 \/ 0 < S n
subgoal 3 is:
 n < S m \/ n = S m \/ S m < n

trichotomy < left.
3 subgoals

  ============================
  0 = 0
```

```
subgoal 2 is:
 S n < 0 \/ S n = 0 \/ 0 < S n
subgoal 3 is:
 n < S m \/ n = S m \/ S m < n

trichotomy < reflexivity.
2 subgoals

  n : nat
  ============================
  S n < 0 \/ S n = 0 \/ 0 < S n

subgoal 2 is:
 n < S m \/ n = S m \/ S m < n
```

For the second case, we know that `Sn > 0`, but how do we show that? Luckily, Coq has this theorem built-in,

```
trichotomy < Check neq_0_lt.
neq_0_lt
     : forall n : nat, 0 <> n -> 0 < n
```

and we can use it to prove the second case.

```
trichotomy < right.
2 subgoals

  n : nat
  ============================
  S n = 0 \/ 0 < S n

subgoal 2 is:
 n < S m \/ n = S m \/ S m < n

trichotomy < right.
2 subgoals

  n : nat
  ============================
  0 < S n

subgoal 2 is:
 n < S m \/ n = S m \/ S m < n

trichotomy < apply neq_0_lt.
2 subgoals

  n : nat
  ============================
  0 <> S n

subgoal 2 is:
 n < S m \/ n = S m \/ S m < n

trichotomy < discriminate.
1 subgoal

  n, m : nat
  IHm : n < m \/ n = m \/ m < n
  ============================
  n < S m \/ n = S m \/ S m < n
```

Now we have to use the induction hypothesis to prove the new goal. We will split it into cases using `destruct`, and the first case is an easy consequence of the `le_S` theorem.

```
trichotomy < destruct IHm.
```

```
2 subgoals

  n, m : nat
  H : n < m
  ============================
  n < S m \/ n = S m \/ S m < n

subgoal 2 is:
 n < S m \/ n = S m \/ S m < n

trichotomy < left.
2 subgoals

  n, m : nat
  H : n < m
  ============================
  n < S m

subgoal 2 is:
 n < S m \/ n = S m \/ S m < n

trichotomy < apply le_S.
2 subgoals

  n, m : nat
  H : n < m
  ============================
  S n <= m

subgoal 2 is:
 n < S m \/ n = S m \/ S m < n

trichotomy < exact H.
1 subgoal

  n, m : nat
  H : n = m \/ m < n
  ============================
  n < S m \/ n = S m \/ S m < n
```

We again destruct the hypothesis, and the second case is handled in much the same way.

```
trichotomy < destruct H.
2 subgoals

  n, m : nat
  H : n = m
  ============================
  n < S m \/ n = S m \/ S m < n

subgoal 2 is:
 n < S m \/ n = S m \/ S m < n

trichotomy < left.
2 subgoals

  n, m : nat
  H : n = m
  ============================
  n < S m

subgoal 2 is:
 n < S m \/ n = S m \/ S m < n

trichotomy < rewrite H.
2 subgoals
```

```
  n, m : nat
  H : n = m
  ============================
  m < S m

subgoal 2 is:
 n < S m \/ n = S m \/ S m < n

trichotomy < apply le_n.
1 subgoal

  n, m : nat
  H : m < n
  ============================
  n < S m \/ n = S m \/ S m < n
```

The last case is tricky because it could correspond to $m+1 = n$ (the middle case of the goal), or $m+1 < n$ (the final case of the goal). Thus we would like to split up these two cases, which we again do using `destruct`.

```
trichotomy < destruct H.
2 subgoals

  m : nat
  ============================
  S m < S m \/ S m = S m \/ S m < S m

subgoal 2 is:
 S m0 < S m \/ S m0 = S m \/ S m < S m0

trichotomy < right.
2 subgoals

  m : nat
  ============================
  S m = S m \/ S m < S m

subgoal 2 is:
 S m0 < S m \/ S m0 = S m \/ S m < S m0

trichotomy < left.
2 subgoals

  m : nat
  ============================
  S m = S m

subgoal 2 is:
 S m0 < S m \/ S m0 = S m \/ S m < S m0

trichotomy < reflexivity.
1 subgoal

  m, m0 : nat
  H : S m <= m0
  ============================
  S m0 < S m \/ S m0 = S m \/ S m < S m0
```

For the final case, we use an small lemma that we have proved above for `le`,

```
trichotomy < Check lt_n_S.
lt_n_S
     : forall n m : nat, n < m -> S n < S m
```

so that

```
trichotomy < right.
```

```
1 subgoal

  m, m0 : nat
  H : S m <= m0
  ============================
  S m0 = S m \/ S m < S m0

trichotomy < right.
1 subgoal

  m, m0 : nat
  H : S m <= m0
  ============================
  S m < S m0

trichotomy < apply lt_n_S.
1 subgoal

  m, m0 : nat
  H : S m <= m0
  ============================
  m < m0

trichotomy < exact H.
No more subgoals.

trichotomy < Qed.
(intros n m).
(induction m).
 (destruct n).
  right.
  left.
  reflexivity.

  right.
  right.
  (apply neq_O_lt).
  discriminate.

 (destruct IHm).
  left.
  (apply le_S).
  exact H.

  (destruct H).
   left.
   (rewrite H).
   (apply le_n).

   (destruct H).
    right.
    left.
    reflexivity.

    right.
    right.
    (apply lt_n_S).
    exact H.

Qed.
trichotomy is defined
```

The other ingredient that we will need in our proofs is the monotonicity of the function involved, meaning the fact that if the argument increases, the result increases as well. We will show this for the square function, using the fact that multiplying an inequality on both sides by the same positive number does not change the inequality.

```
Coq < Check mult_lt_compat_l.
mult_lt_compat_l
     : forall n m p : nat, n < m -> 0 < p -> p * n < p * m
```

We begin by introducing variables, and then particular instances of this proof, as well as a helpful version of a lemma (sq_monotonic) we previously used.

```
Coq < Theorem sq_monotonic : forall n m : nat, n < m -> n * n < m * m.
1 subgoal

  ============================
  forall n m : nat, n < m -> n * n < m * m

sq_monotonic < intros n m H.
1 subgoal

  n, m : nat
  H : n < m
  ============================
  n * n < m * m

sq_monotonic < pose proof mult_lt_compat_l n m n H.
1 subgoal

  n, m : nat
  H : n < m
  H0 : 0 < n -> n * n < n * m
  ============================
  n * n < m * m

sq_monotonic < pose proof mult_lt_compat_l n m m H.
1 subgoal

  n, m : nat
  H : n < m
  H0 : 0 < n -> n * n < n * m
  H1 : 0 < m -> m * n < m * m
  ============================
  n * n < m * m

sq_monotonic < assert (forall n : nat, 0 < S n).
2 subgoals

  n, m : nat
  H : n < m
  H0 : 0 < n -> n * n < n * m
  H1 : 0 < m -> m * n < m * m
  ============================
  forall n0 : nat, 0 < S n0

subgoal 2 is:
 n * n < m * m

sq_monotonic < intro n0.
2 subgoals

  n, m : nat
  H : n < m
  H0 : 0 < n -> n * n < n * m
  H1 : 0 < m -> m * n < m * m
  n0 : nat
  ============================
  0 < S n0

subgoal 2 is:
 n * n < m * m
```

```
sq_monotonic < apply (neq_O_lt (S n0)).
2 subgoals

  n, m : nat
  H : n < m
  H0 : 0 < n -> n * n < n * m
  H1 : 0 < m -> m * n < m * m
  n0 : nat
  ============================
  0 <> S n0

subgoal 2 is:
 n * n < m * m

sq_monotonic < discriminate.
1 subgoal

  n, m : nat
  H : n < m
  H0 : 0 < n -> n * n < n * m
  H1 : 0 < m -> m * n < m * m
  H2 : forall n : nat, 0 < S n
  ============================
  n * n < m * m
```

We must now handle the case that $m = 0$, which we can do using inversion on our original hypothesis,

```
sq_monotonic < destruct m.
2 subgoals

  n : nat
  H : n < 0
  H0 : 0 < n -> n * n < n * 0
  H1 : 0 < 0 -> 0 * n < 0 * 0
  H2 : forall n : nat, 0 < S n
  ============================
  n * n < 0 * 0

subgoal 2 is:
 n * n < S m * S m

sq_monotonic < inversion H.
1 subgoal

  n, m : nat
  H : n < S m
  H0 : 0 < n -> n * n < n * S m
  H1 : 0 < S m -> S m * n < S m * S m
  H2 : forall n : nat, 0 < S n
  ============================
  n * n < S m * S m
```

Similarly, we need to handle the $n = 0$ case, which we can do using our convenience lemma.

```
sq_monotonic < destruct n.
2 subgoals

  m : nat
  H : 0 < S m
  H0 : 0 < 0 -> 0 * 0 < 0 * S m
  H1 : 0 < S m -> S m * 0 < S m * S m
  H2 : forall n : nat, 0 < S n
  ============================
  0 * 0 < S m * S m
```

```
subgoal 2 is:
 S n * S n < S m * S m

sq_monotonic < simpl.
2 subgoals

  m : nat
  H : 0 < S m
  H0 : 0 < 0 -> 0 * 0 < 0 * S m
  H1 : 0 < S m -> S m * 0 < S m * S m
  H2 : forall n : nat, 0 < S n
  ============================
  0 < S (m + m * S m)

subgoal 2 is:
 S n * S n < S m * S m

sq_monotonic < exact (H2 (m + m * (S m))).
1 subgoal

  n, m : nat
  H : S n < S m
  H0 : 0 < S n -> S n * S n < S n * S m
  H1 : 0 < S m -> S m * S n < S m * S m
  H2 : forall n : nat, 0 < S n
  ============================
  S n * S n < S m * S m
```

Next, we use the same lemma again, to simplify our hypotheses,

```
sq_monotonic < pose proof H2 n.
1 subgoal

  n, m : nat
  H : S n < S m
  H0 : 0 < S n -> S n * S n < S n * S m
  H1 : 0 < S m -> S m * S n < S m * S m
  H2 : forall n : nat, 0 < S n
  H3 : 0 < S n
  ============================
  S n * S n < S m * S m

sq_monotonic < pose proof H2 m.
1 subgoal

  n, m : nat
  H : S n < S m
  H0 : 0 < S n -> S n * S n < S n * S m
  H1 : 0 < S m -> S m * S n < S m * S m
  H2 : forall n : nat, 0 < S n
  H3 : 0 < S n
  H4 : 0 < S m
  ============================
  S n * S n < S m * S m

sq_monotonic < apply H0 in H3.
1 subgoal

  n, m : nat
  H : S n < S m
  H0 : 0 < S n -> S n * S n < S n * S m
  H1 : 0 < S m -> S m * S n < S m * S m
  H2 : forall n : nat, 0 < S n
  H3 : S n * S n < S n * S m
  H4 : 0 < S m
  ============================
  S n * S n < S m * S m
```

```
sq_monotonic < apply H1 in H4.
1 subgoal

  n, m : nat
  H : S n < S m
  H0 : 0 < S n -> S n * S n < S n * S m
  H1 : 0 < S m -> S m * S n < S m * S m
  H2 : forall n : nat, 0 < S n
  H3 : S n * S n < S n * S m
  H4 : S m * S n < S m * S m
  ============================
  S n * S n < S m * S m
```

Finally, we can use the transitivity of the relation to establish our theorem.

```
sq_monotonic < rewrite (mult_comm (S n) (S m)) in H3.
1 subgoal

  n, m : nat
  H : S n < S m
  H0 : 0 < S n -> S n * S n < S n * S m
  H1 : 0 < S m -> S m * S n < S m * S m
  H2 : forall n : nat, 0 < S n
  H3 : S n * S n < S m * S n
  H4 : S m * S n < S m * S m
  ============================
  S n * S n < S m * S m

sq_monotonic < apply (lt_trans (S n * S n) (S m * S n) (S m * S m)).
2 subgoals

  n, m : nat
  H : S n < S m
  H0 : 0 < S n -> S n * S n < S n * S m
  H1 : 0 < S m -> S m * S n < S m * S m
  H2 : forall n : nat, 0 < S n
  H3 : S n * S n < S m * S n
  H4 : S m * S n < S m * S m
  ============================
  S n * S n < S m * S n

subgoal 2 is:
 S m * S n < S m * S m

sq_monotonic < exact H3.
1 subgoal

  n, m : nat
  H : S n < S m
  H0 : 0 < S n -> S n * S n < S n * S m
  H1 : 0 < S m -> S m * S n < S m * S m
  H2 : forall n : nat, 0 < S n
  H3 : S n * S n < S m * S n
  H4 : S m * S n < S m * S m
  ============================
  S m * S n < S m * S m

sq_monotonic < exact H4.
No more subgoals.

sq_monotonic < Qed.
(intros n m H).
(pose proof (mult_lt_compat_l n m n H)).
(pose proof (mult_lt_compat_l n m m H)).
(assert (forall n : nat, 0 < S n)).
 intro n0.
 (apply (neq_O_lt (S n0))).
 discriminate.

 (destruct m).
  (inversion H).

  (destruct n).
   (simpl).
   exact (H2 (m + m * S m)).

   (pose proof (H2 n)).
   (pose proof (H2 m)).
   (apply H0 in H3).
   (apply H1 in H4).
   (rewrite (mult_comm (S n) (S m)) in H3).
   (apply (lt_trans (S n * S n) (S m * S n) (S m * S m))).
    exact H3.

    exact H4.

Qed.
sq_monotonic is defined
```

3.4 Functions Redux

It is possible to treat functions from a slightly higher level of abstraction, so they look more similar to the familiar presentation in elementary mathematics. In order to do this, we will need some more sophisticated machinery from Coq, but it will allow us to shorten our proofs from Section 3.3. We would like a way to use the `exists` tactic with type `Type` rather than `Prop`, without using the machinery of Σ-types. We will see how this is used below.

```
From Coq Require Import Bool.Bool.
From Coq Require Import Classes.RelationClasses.

Variable (choice : forall {A P} (prf : exists a : A, P a), A).
Hypothesis (choice_ok : forall {A P} (prf : exists a : A, P a), P (choice prf)).
```

We will again define a function as a special kind of binary relation mapping some domain to a codomain. Now we will incorporate the idea of totality directly into the definition, so that all functions are total, and refer to the partial function property as *functionality*.

```
Record Func (dom cod : Type) : Type :=
  mkFunc {
     rel : dom -> cod -> Prop
    ; total : forall x : dom, exists y : cod, rel x y
    ; functional : forall x : dom, forall y z : cod, rel x y -> rel x z -> y = z
    }.
```

The `Record` construct allows us to make a data structure with constructor `mkFunc`, for which we need to specify a domain, a codomain, a relation, a proof of totality, and a proof of functionality.

Most of the simplification in our proofs will come from defining a notion a function application, meaning applying the function to an element of the domain to generate and element of the codomain. We need to prove that such as map exists for a given function f. However, totality of f tells us that all elements of the domain are mapped to some element of the codomain. We can access parts of a record using the dot operator, just as in Python and C, so that `f.(total dom cod)` is our proof of totality. Given an element x of the domain, it returns an existence statement for the corresponding y in the codomain.

```
Definition app : forall {dom cod}, Func dom cod -> dom -> cod.
Proof.
  intros dom cod f x.
  exact (choice (f.(total dom cod) x)).
Defined.
```

We would normally destruct this existence statement to get the witness y. However, this is not possible for `Type`. Thus, we feed this existence statement to the `choice` operator above to extract the type `cod` itself. Notice also that we end with `Defined`. This makes a *transparent* definition that can be unfolded by tactics like `simpl`, rather than an *opaque* defintion that you get from `Qed`.

We can prove that this notion of function application is equivalent to the relational notion in Section 3.3. We define our totality proof up front in order to simplify the derivation. Notice that we can unfold the `app` because it is a transparent definition.

```
Coq < Lemma app_iff_rel : forall {dom cod} (f : Func dom cod) (x : dom) (y : cod),
  app f x = y <-> (f.(rel dom cod) x y).
1 subgoal

  ============================
  forall (dom cod : Type) (f : Func dom cod) (x : dom) (y : cod),
  app f x = y <-> rel dom cod f x y

app_iff_rel < intros dom cod f x y.
1 subgoal

  dom : Type
  cod : Type
  f : Func dom cod
  x : dom
  y : cod
  ============================
  app f x = y <-> rel dom cod f x y

app_iff_rel < pose (s := total dom cod f x).
1 subgoal

  dom : Type
  cod : Type
  f : Func dom cod
  x : dom
  y : cod
  s := total dom cod f x : exists y : cod, rel dom cod f x y
  ============================
  app f x = y <-> rel dom cod f x y

app_iff_rel < unfold app.
1 subgoal

  dom : Type
  cod : Type
  f : Func dom cod
  x : dom
  y : cod
  s := total dom cod f x : exists y : cod, rel dom cod f x y
  ============================
  choice (total dom cod f x) = y <-> rel dom cod f x y

app_iff_rel < fold s.
```

```
1 subgoal

  dom : Type
  cod : Type
  f : Func dom cod
  x : dom
  y : cod
  s := total dom cod f x : exists y : cod, rel dom cod f x y
  ============================
  choice s = y <-> rel dom cod f x y

app_iff_rel < split.
2 subgoals

  dom : Type
  cod : Type
  f : Func dom cod
  x : dom
  y : cod
  s := total dom cod f x : exists y : cod, rel dom cod f x y
  ============================
  choice s = y -> rel dom cod f x y

subgoal 2 is:
 rel dom cod f x y -> choice s = y
```

We can prove the first direction by replacing y with our choice operator applied to the existence statement, and then using the `choice_ok` theorem.

```
app_iff_rel < intro fapp.
2 subgoals

  dom : Type
  cod : Type
  f : Func dom cod
  x : dom
  y : cod
  s := total dom cod f x : exists y : cod, rel dom cod f x y
  fapp : choice s = y
  ============================
  rel dom cod f x y

subgoal 2 is:
 rel dom cod f x y -> choice s = y

app_iff_rel < rewrite <- fapp.
2 subgoals

  dom : Type
  cod : Type
  f : Func dom cod
  x : dom
  y : cod
  s := total dom cod f x : exists y : cod, rel dom cod f x y
  fapp : choice s = y
  ============================
  rel dom cod f x (choice s)

subgoal 2 is:
 rel dom cod f x y -> choice s = y

app_iff_rel < pose (H := choice_ok s).
2 subgoals

  dom : Type
  cod : Type
  f : Func dom cod
  x : dom
  y : cod
  s := total dom cod f x : exists y : cod, rel dom cod f x y
  fapp : choice s = y
  H := choice_ok s : rel dom cod f x (choice s)
  ============================
  rel dom cod f x (choice s)

subgoal 2 is:
 rel dom cod f x y -> choice s = y

app_iff_rel < exact H.
1 subgoal

  dom : Type
  cod : Type
  f : Func dom cod
  x : dom
  y : cod
  s := total dom cod f x : exists y : cod, rel dom cod f x y
  ============================
  rel dom cod f x y -> choice s = y
```

For the reverse implication, we use the functionality of f, namely that the same input x cannot generate two different outputs. We can use the same `choice_ok` hypothesis from before to prove one side, and our original hypothesis for the other.

```
app_iff_rel < intro frel.
1 subgoal

  dom : Type
  cod : Type
  f : Func dom cod
  x : dom
  y : cod
  s := total dom cod f x : exists y : cod, rel dom cod f x y
  frel : rel dom cod f x y
  ============================
  choice s = y

app_iff_rel < apply (f.(functional dom cod)) with (x := x).
2 subgoals

  dom : Type
  cod : Type
  f : Func dom cod
  x : dom
  y : cod
  s := total dom cod f x : exists y : cod, rel dom cod f x y
  frel : rel dom cod f x y
  ============================
  rel dom cod f x (choice s)

subgoal 2 is:
 rel dom cod f x y

app_iff_rel < exact (choice_ok s).
1 subgoal

  dom : Type
  cod : Type
  f : Func dom cod
  x : dom
  y : cod
  s := total dom cod f x : exists y : cod, rel dom cod f x y
  frel : rel dom cod f x y
  ============================
  rel dom cod f x y

app_iff_rel < exact frel.
No more subgoals.

app_iff_rel < Qed.
(intros dom cod f x y).
(pose (s := total dom cod f x)).
(unfold app).
(fold s).
split.
 intro fapp.
 (rewrite <- fapp).
 (pose (H := choice_ok s)).
 exact H.

 intro frel.
 (apply f.(functional dom cod) with (x := x)).
  exact (choice_ok s).

  exact frel.

Qed.
app_iff_rel is defined
```

In order to simplify the creation of functions, we would like to show that normal Coq functions are equivalent, so that we can define a function "by formula". Notice that we make a transparent defintion here as well. Ideally we would just like to create a function object out of the pieces from our Coq function. However, typing the the proofs of functionality and totality directly is difficult. Therefore, instead of the `pose` tactic, we use `refine`. This allows us to leave the proofs unspecified, and it turns them into subgoals.

```
Coq < Definition by_formula : forall {dom cod : Type}, (dom -> cod) -> Func dom cod.
1 subgoal

  ============================
  forall dom cod : Type, (dom -> cod) -> Func dom cod

by_formula < intros dom cod f.
1 subgoal

  dom : Type
  cod : Type
  f : dom -> cod
  ============================
  Func dom cod

by_formula < refine (mkFunc dom cod (fun x y => f x = y) _ _).
2 subgoals

  dom : Type
  cod : Type
  f : dom -> cod
  ============================
  forall x : dom, exists y : cod, f x = y

subgoal 2 is:
 forall (x : dom) (y z : cod), f x = y -> f x = z -> y = z
```

In order to prove totality of our Coq function, we just apply it to the input x, which is guaranteed to succeed.

```
by_formula < intro x.
2 subgoals
```

```
  dom : Type
  cod : Type
  f : dom -> cod
  x : dom
  ============================
  exists y : cod, f x = y

subgoal 2 is:
 forall (x : dom) (y z : cod), f x = y -> f x = z -> y = z

by_formula < exists (f x).
2 subgoals

  dom : Type
  cod : Type
  f : dom -> cod
  x : dom
  ============================
  f x = f x

subgoal 2 is:
 forall (x : dom) (y z : cod), f x = y -> f x = z -> y = z

by_formula < reflexivity.
1 subgoal

  dom : Type
  cod : Type
  f : dom -> cod
  ============================
  forall (x : dom) (y z : cod), f x = y -> f x = z -> y = z
```

To prove functionality, we just use reflexivity of $f(x)$.

```
by_formula < intros x y z H1 H2.
1 subgoal

  dom : Type
  cod : Type
  f : dom -> cod
  x : dom
  y, z : cod
  H1 : f x = y
  H2 : f x = z
  ============================
  y = z

by_formula < rewrite <- H1.
1 subgoal

  dom : Type
  cod : Type
  f : dom -> cod
  x : dom
  y, z : cod
  H1 : f x = y
  H2 : f x = z
  ============================
  f x = z

by_formula < rewrite <- H2.
1 subgoal

  dom : Type
  cod : Type
  f : dom -> cod
  x : dom
  y, z : cod
  H1 : f x = y
  H2 : f x = z
  ============================
  f x = f x

by_formula < reflexivity.
No more subgoals.

by_formula < Defined.
(intros dom cod f).
refine (mkFunc dom cod (fun x y => f x = y) _ _).
 intro x.
 exists (f x).
 reflexivity.

 (intros x y z H1 H2).
 (rewrite <- H1).
 (rewrite <- H2).
 reflexivity.

Defined.
by_formula is defined
```

We can prove the application of the Coq function is equivalent to application of our function type. If we unfold all definitions and simplify, we are left with a choice statement.

```
Coq < Lemma by_formula_ok : forall {dom cod : Type} (formula : dom -> cod) (x : dom),
  app (by_formula formula) x = formula x.
Coq < 1 subgoal

  ============================
  forall (dom cod : Type) (formula : dom -> cod) (x : dom),
  app (by_formula formula) x = formula x

by_formula_ok < intros dom cod f x.
1 subgoal

  dom : Type
  cod : Type
  f : dom -> cod
  x : dom
  ============================
  app (by_formula f) x = f x

by_formula_ok < unfold by_formula.
1 subgoal

  dom : Type
  cod : Type
  f : dom -> cod
  x : dom
  ============================
  app
    {|
    rel := fun (x0 : dom) (y : cod) => f x0 = y;
    total := fun x0 : dom =>
            ex_intro (fun y : cod => f x0 = y) (f x0) eq_refl;
    functional := fun (x0 : dom) (y z : cod) (eq1 : f x0 = y)
                    (eq2 : f x0 = z) =>
                  eq_ind (f x0) (fun y0 : cod => y0 = z)
                    (eq_ind (f x0) (fun z0 : cod => f x0 = z0) eq_refl z eq2)
                    y eq1 |} x = f x

by_formula_ok < unfold app.
1 subgoal
```

```
  dom : Type
  cod : Type
  f : dom -> cod
  x : dom
  ============================
  choice
    (total dom cod
       {|
       rel := fun (x0 : dom) (y : cod) => f x0 = y;
       total := fun x0 : dom =>
                ex_intro (fun y : cod => f x0 = y) (f x0) eq_refl;
       functional := fun (x0 : dom) (y z : cod) (eq1 : f x0 = y)
                       (eq2 : f x0 = z) =>
                     eq_ind (f x0) (fun y0 : cod => y0 = z)
                       (eq_ind (f x0) (fun z0 : cod => f x0 = z0) eq_refl z
                          eq2) y eq1 |} x) = f x

by_formula_ok < simpl.
1 subgoal

  dom : Type
  cod : Type
  f : dom -> cod
  x : dom
  ============================
  choice (ex_intro (fun y : cod => f x = y) (f x) eq_refl) = f x
```

This goal is exactly the same as our `choice_ok` theorem, so we can just plug in.

```
by_formula_ok < pose (H := choice_ok (ex_intro (fun y : cod => f x = y) (f x) eq_refl)).
1 subgoal

  dom : Type
  cod : Type
  f : dom -> cod
  x : dom
  H := choice_ok (ex_intro (fun y : cod => f x = y) (f x) eq_refl)
    : f x = choice (ex_intro (fun y : cod => f x = y) (f x) eq_refl)
  ============================
  choice (ex_intro (fun y : cod => f x = y) (f x) eq_refl) = f x

by_formula_ok < symmetry.
1 subgoal

  dom : Type
  cod : Type
  f : dom -> cod
  x : dom
  H := choice_ok (ex_intro (fun y : cod => f x = y) (f x) eq_refl)
    : f x = choice (ex_intro (fun y : cod => f x = y) (f x) eq_refl)
  ============================
  f x = choice (ex_intro (fun y : cod => f x = y) (f x) eq_refl)

by_formula_ok < exact H.
No more subgoals.

by_formula_ok < Qed.
(intros dom cod f x).
(unfold by_formula).
(unfold app).
(simpl).
(pose (H := choice_ok (ex_intro (fun y : cod => f x = y) (f x) eq_refl))).
symmetry.
exact H.

Qed.
by_formula_ok is defined
```

In order to make it easier to compare functions, we introduce *extensionality*. This means that two functions are the same, if they always produce the same outputs. This means that the implementation is irrelevant, we just care about what we can measure. For example, the functions $f(x) = 2(x + 1)$ and $g(x) = 2x + 2$ are equal. This is the same idea as proof irrelevance.

```
Hypothesis funext : forall {dom cod} (f g : Func dom cod),
  (forall x : dom, app f x = app g x) -> f = g.
```

We can define a surjective function in exactly the same way as in Section 3.3, but we use our new function application operator. Notice that this defintion is dual to our totality definition.

```
Definition surjective {dom cod} (f : Func dom cod) :=
  forall y : cod, exists x : dom, app f x = y.
```

Similarly, an injective function can be defined as the dual to our functionality definition.

```
Definition injective {dom cod} (f : Func dom cod) :=
  forall x xp : dom, app f x = app f xp -> x = xp.
```

We can now give a concise proof that the composition of two functions exists. Notice we can use our `app` defintion to define our new function, rather than the relational definition we used in Section 3.3.

```
Coq < Definition comp : forall {A B C}, Func B C -> Func A B -> Func A C.
1 subgoal

  ============================
  forall A B C : Type, Func B C -> Func A B -> Func A C

comp < intros A B C g f.
1 subgoal

  A : Type
  B : Type
  C : Type
  g : Func B C
  f : Func A B
  ============================
  Func A C

comp < pose (h := by_formula (fun a => app g (app f a))).
1 subgoal

  A : Type
  B : Type
  C : Type
  g : Func B C
  f : Func A B
  h := by_formula (fun a : A => app g (app f a)) : Func A C
  ============================
  Func A C

comp < exact h.
No more subgoals.

comp < Defined.
(intros A B C g f).
(pose (h := by_formula (fun a => app g (app f a)))).
exact h.

Defined.
comp is defined
```

Our composite function must agree with the usual definition. We know this because we previously proved that formula definitions are valid.

```
Coq < Corollary comp_ok : forall {A B C} (g : Func B C) (f : Func A B) (a : A),
  app (comp g f) a = app g (app f a).
1 subgoal

  ============================
  forall (A B C : Type) (g : Func B C) (f : Func A B) (a : A),
  app (comp g f) a = app g (app f a)

comp_ok < intros A B C g f a.
1 subgoal

  A : Type
  B : Type
  C : Type
  g : Func B C
  f : Func A B
  a : A
  ============================
  app (comp g f) a = app g (app f a)

comp_ok < unfold comp.
1 subgoal

  A : Type
  B : Type
  C : Type
  g : Func B C
  f : Func A B
  a : A
  ============================
  app (by_formula (fun a0 : A => app g (app f a0))) a = app g (app f a)

comp_ok < rewrite by_formula_ok.
1 subgoal

  A : Type
  B : Type
  C : Type
  g : Func B C
  f : Func A B
  a : A
  ============================
  app g (app f a) = app g (app f a)

comp_ok < reflexivity.
No more subgoals.

comp_ok < Qed.
(intros A B C g f a).
(unfold comp).
(rewrite by_formula_ok).
reflexivity.

Qed.
comp_ok is defined
```

And we can easily prove that function composition is associative. The idea is to first use extensionality, so that I am only comparing outputs, not functions. Then use the fact that composition can always be turned into a repeated series of function applications.

```
Coq> Lemma comp_assoc : forall {A B C D} (f : Func C D) (g : Func B C) (h : Func A B), comp f (comp g h) = comp (comp f g) h.
comp_assoc < intros A B C D f g h.
1 subgoal
```

```
  A : Type
  B : Type
  C : Type
  D : Type
  f : Func C D
  g : Func B C
  h : Func A B
  ============================
  comp f (comp g h) = comp (comp f g) h

comp_assoc < apply funext.
1 subgoal

  A : Type
  B : Type
  C : Type
  D : Type
  f : Func C D
  g : Func B C
  h : Func A B
  ============================
  forall x : A, app (comp f (comp g h)) x = app (comp (comp f g) h) x

comp_assoc < intro a.
1 subgoal

  A : Type
  B : Type
  C : Type
  D : Type
  f : Func C D
  g : Func B C
  h : Func A B
  a : A
  ============================
  app (comp f (comp g h)) a = app (comp (comp f g) h) a

comp_assoc < repeat rewrite comp_ok.
1 subgoal

  A : Type
  B : Type
  C : Type
  D : Type
  f : Func C D
  g : Func B C
  h : Func A B
  a : A
  ============================
  app f (app g (app h a)) = app f (app g (app h a))

comp_assoc < reflexivity.
No more subgoals.

comp_assoc < Qed.
(intros A B C D f g h).
(apply funext).
intro a.
(repeat rewrite comp_ok).
reflexivity.

Qed.
comp_assoc is defined
```

This definition of function composition allows us to define an algebra of functions, and we can start to see why certain operations work. For example, suppose I have a statement such as

$$g \circ f = h \circ f. \tag{3.36}$$

By extensionality, I know that this means the two composed functions have equal outputs for every input. If the function f is surjective, then every point $b \in B$ is produced by some input $a \in A$. This means that if g and h produce the same output for every point of B, then the composition produces the same output for every point of A, and we could cancel f from both sides of the equation. This means that the function f is *right cancellable*. We first introduce hypotheses and use extensionality.

```
Coq < Theorem sur_then_rc : forall {A B} (f : Func A B),
  surjective f -> forall C (g h : Func B C), comp g f = comp h f -> g = h.
Coq < 1 subgoal

  ============================
  forall (A B : Type) (f : Func A B),
  surjective f ->
  forall (C : Type) (g h : Func B C), comp g f = comp h f -> g = h

sur_then_rc < intros A B f Surj C g h H.
1 subgoal

  A : Type
  B : Type
  f : Func A B
  Surj : surjective f
  C : Type
  g, h : Func B C
  H : comp g f = comp h f
  ============================
  g = h

sur_then_rc < apply funext.
1 subgoal

  A : Type
  B : Type
  f : Func A B
  Surj : surjective f
  C : Type
  g, h : Func B C
  H : comp g f = comp h f
  ============================
  forall x : B, app g x = app h x

sur_then_rc < intro b.
1 subgoal

  A : Type
  B : Type
  f : Func A B
  Surj : surjective f
  C : Type
  g, h : Func B C
  H : comp g f = comp h f
  b : B
  ============================
  app g b = app h b
```

Now we can extract the witness from the sujectivity statement, and use it to rewrite the goal.

```
sur_then_rc < destruct (Surj b) as [a H2].
1 subgoal

  A : Type
  B : Type
  f : Func A B
  Surj : surjective f
  C : Type
  g, h : Func B C
  H : comp g f = comp h f
  b : B
  a : A
  H2 : app f a = b
  ============================
  app g b = app h b

sur_then_rc < rewrite <- H2.
1 subgoal

  A : Type
  B : Type
  f : Func A B
  Surj : surjective f
  C : Type
  g, h : Func B C
  H : comp g f = comp h f
  b : B
  a : A
  H2 : app f a = b
  ============================
  app g (app f a) = app h (app f a)
```

Now we can replace repeated application with composition, and use our original assumption to complete the proof.

```
sur_then_rc < rewrite <- comp_ok.
1 subgoal

  A : Type
  B : Type
  f : Func A B
  Surj : surjective f
  C : Type
  g, h : Func B C
  H : comp g f = comp h f
  b : B
  a : A
  H2 : app f a = b
  ============================
  app (comp g f) a = app h (app f a)

sur_then_rc < rewrite <- comp_ok.
1 subgoal

  A : Type
  B : Type
  f : Func A B
  Surj : surjective f
  C : Type
  g, h : Func B C
  H : comp g f = comp h f
  b : B
  a : A
  H2 : app f a = b
  ============================
  app (comp g f) a = app (comp h f) a

sur_then_rc < rewrite H.
```

```
1 subgoal

  A : Type
  B : Type
  f : Func A B
  Surj : surjective f
  C : Type
  g, h : Func B C
  H : comp g f = comp h f
  b : B
  a : A
  H2 : app f a = b
  ============================
  app (comp h f) a = app (comp h f) a

sur_then_rc < reflexivity.
No more subgoals.

sur_then_rc < Qed.
(intros A B f Surj C g h H).
(apply funext).
intro b.
(destruct (Surj b) as [a H2]).
(rewrite <- H2).
(rewrite <- comp_ok).
(rewrite <- comp_ok).
(rewrite H).
reflexivity.

Qed.
sur_then_rc is defined
```

Similarly, every injective function is *left cancellable*. We can see this by reasoning in the same way. Appealing to extensionality, the two composed functions are equal if the outputs are equal. Since f is injective, the inputs are equal if the outputs are equal. Thus we can reduce the problem to establishing the equailty of g and h. We start by introducing hypotheses and using extensionality.

```
Coq < Theorem inj_then_lc : forall {B C} (f : Func B C),
  injective f -> forall A (g h : Func A B), comp f g = comp f h -> g = h.
Coq < 1 subgoal

  ============================
  forall (B C : Type) (f : Func B C),
  injective f ->
  forall (A : Type) (g h : Func A B), comp f g = comp f h -> g = h

inj_then_lc < intros B C f Inj A g h H.
1 subgoal

  B : Type
  C : Type
  f : Func B C
  Inj : injective f
  A : Type
  g, h : Func A B
  H : comp f g = comp f h
  ============================
  g = h

inj_then_lc < apply funext.
1 subgoal

  B : Type
  C : Type
  f : Func B C
  Inj : injective f
  A : Type
  g, h : Func A B
  H : comp f g = comp f h
  ============================
  forall x : A, app g x = app h x

inj_then_lc < intro a.
1 subgoal

  B : Type
  C : Type
  f : Func B C
  Inj : injective f
  A : Type
  g, h : Func A B
  H : comp f g = comp f h
  a : A
  ============================
  app g a = app h a
```

We apply injectivity, which says that the inputs to f in the goal will be equal if the outputs from f are equal. Then we replace repeated application with composition to complete the proof.

```
inj_then_lc < apply Inj.
1 subgoal

  B : Type
  C : Type
  f : Func B C
  Inj : injective f
  A : Type
  g, h : Func A B
  H : comp f g = comp f h
  a : A
  ============================
  app f (app g a) = app f (app h a)

inj_then_lc < rewrite <- comp_ok.
1 subgoal

  B : Type
  C : Type
  f : Func B C
  Inj : injective f
  A : Type
  g, h : Func A B
  H : comp f g = comp f h
  a : A
  ============================
  app (comp f g) a = app f (app h a)

inj_then_lc < rewrite <- comp_ok.
1 subgoal

  B : Type
  C : Type
  f : Func B C
  Inj : injective f
  A : Type
  g, h : Func A B
  H : comp f g = comp f h
  a : A
  ============================
  app (comp f g) a = app (comp f h) a
```

```
inj_then_lc < rewrite H.
1 subgoal

  B : Type
  C : Type
  f : Func B C
  Inj : injective f
  A : Type
  g, h : Func A B
  H : comp f g = comp f h
  a : A
  ============================
  app (comp f h) a = app (comp f h) a

inj_then_lc < reflexivity.
No more subgoals.

inj_then_lc < Qed.
(intros B C f Inj A g h H).
(apply funext).
intro a.
(apply Inj).
(rewrite <- comp_ok).
(rewrite <- comp_ok).
(rewrite H).
reflexivity.

Qed.
inj_then_lc is defined
```

In order to define the inverse function, we will first define the *identity function*. This is the simplest possible function, which just returns its input, $f(x) = x$. Notice we use a transparent definition so that we can unfold this during proofs.

```
Definition id : forall {A}, Func A A.
Proof.
  intros A.
  exact (by_formula (fun x => x)).
Defined.
```

We can prove that this function has the action we expect. The strange syntax is needed because the type A is an implicit argument to our function.

```
Coq < Corollary id_ok : forall {A : Type} x, app (@id A) x = x.
1 subgoal

  ============================
  forall (A : Type) (x : A), app id x = x

id_ok < intros A x.
1 subgoal

  A : Type
  x : A
  ============================
  app id x = x

id_ok < unfold id.
1 subgoal

  A : Type
  x : A
  ============================
  app (by_formula (fun x0 : A => x0)) x = x
```

```
id_ok < rewrite by_formula_ok.
1 subgoal

  A : Type
  x : A
  ============================
  x = x

id_ok < reflexivity.
No more subgoals.

id_ok < Qed.
(intros A x).
(unfold id).
(rewrite by_formula_ok).
reflexivity.

Qed.
id_ok is defined
```

Now we can define invertibility. A function f is invertible if there exists a function g such that there composition, in either order, gives the identity function. Notice that the inverse g maps in the opposite direction from f, from codomain to domain.

```
Definition invertible {A B} (f : Func A B) :=
  exists g : Func B A, comp f g = id /\ comp g f = id.
```

We can now rewrite our proof from Section 3.3 that every bijective function is invertible. We begin by introducing hypotheses and unfolding the definitions of invertibility and surjectivity.

```
Coq < Theorem bij_then_inv : forall {A B} (f : Func A B), injective f -> surjective f -> invertible f.
1 subgoal

  ============================
  forall (A B : Type) (f : Func A B),
  injective f -> surjective f -> invertible f

bij_then_inv < intros A B f Inj Sur.
1 subgoal

  A : Type
  B : Type
  f : Func A B
  Inj : injective f
  Sur : surjective f
  ============================
  invertible f

bij_then_inv < unfold invertible.
1 subgoal

  A : Type
  B : Type
  f : Func A B
  Inj : injective f
  Sur : surjective f
  ============================
  exists g : Func B A, comp f g = id /\ comp g f = id

bij_then_inv < unfold surjective in Sur.
1 subgoal

  A : Type
  B : Type
  f : Func A B
  Inj : injective f
  Sur : forall y : B, exists x : A, app f x = y
  ============================
  exists g : Func B A, comp f g = id /\ comp g f = id
```

Now we need to define the inverse function g. What we want is a function that takes in an element $b \in B$ and returns the $x \in A$ from the statement of surjectivity. We use the `choice` operator to extract this witness. After providing the function, we split the proof of the conjunction.

```
bij_then_inv < exists (by_formula (fun b => choice (Sur b))).
1 subgoal

  A : Type
  B : Type
  f : Func A B
  Inj : injective f
  Sur : forall y : B, exists x : A, app f x = y
  ============================
  comp f (by_formula (fun b : B => choice (Sur b))) = id /\
  comp (by_formula (fun b : B => choice (Sur b))) f = id

bij_then_inv < split.
2 subgoals

  A : Type
  B : Type
  f : Func A B
  Inj : injective f
  Sur : forall y : B, exists x : A, app f x = y
  ============================
  comp f (by_formula (fun b : B => choice (Sur b))) = id

subgoal 2 is:
 comp (by_formula (fun b : B => choice (Sur b))) f = id
```

To prove equality of functions, we make use of extensionality, and introduce the element of B.

```
bij_then_inv < apply funext.
2 subgoals

  A : Type
  B : Type
  f : Func A B
  Inj : injective f
  Sur : forall y : B, exists x : A, app f x = y
  ============================
  forall x : B,
  app (comp f (by_formula (fun b : B => choice (Sur b)))) x = app id x

subgoal 2 is:
 comp (by_formula (fun b : B => choice (Sur b))) f = id

bij_then_inv < intro b.
2 subgoals

  A : Type
  B : Type
  f : Func A B
  Inj : injective f
  Sur : forall y : B, exists x : A, app f x = y
  b : B
  ============================
  app (comp f (by_formula (fun b0 : B => choice (Sur b0)))) b = app id b
```

```
subgoal 2 is:
 comp (by_formula (fun b : B => choice (Sur b))) f = id
```

We now turn composition into repeated application, and evaluate `app` on our formula.

```
bij_then_inv < rewrite comp_ok.
2 subgoals

  A : Type
  B : Type
  f : Func A B
  Inj : injective f
  Sur : forall y : B, exists x : A, app f x = y
  b : B
  ============================
  app f (app (by_formula (fun b0 : B => choice (Sur b0))) b) = app id b

subgoal 2 is:
 comp (by_formula (fun b : B => choice (Sur b))) f = id

bij_then_inv < rewrite by_formula_ok.
2 subgoals

  A : Type
  B : Type
  f : Func A B
  Inj : injective f
  Sur : forall y : B, exists x : A, app f x = y
  b : B
  ============================
  app f (choice (Sur b)) = app id b

subgoal 2 is:
 comp (by_formula (fun b : B => choice (Sur b))) f = id
```

We use the `choice_ok` theorem to extract the value from our choice statement, or more precisely define an equality for that value.

```
bij_then_inv < pose (H := choice_ok (Sur b)).
2 subgoals

  A : Type
  B : Type
  f : Func A B
  Inj : injective f
  Sur : forall y : B, exists x : A, app f x = y
  b : B
  H := choice_ok (Sur b) : (fun x : A => app f x = b) (choice (Sur b))
  ============================
  app f (choice (Sur b)) = app id b

subgoal 2 is:
 comp (by_formula (fun b : B => choice (Sur b))) f = id

bij_then_inv < simpl in H.
2 subgoals

  A : Type
  B : Type
  f : Func A B
  Inj : injective f
  Sur : forall y : B, exists x : A, app f x = y
  b : B
  H := choice_ok (Sur b) : app f (choice (Sur b)) = b
  ============================
  app f (choice (Sur b)) = app id b

subgoal 2 is:
 comp (by_formula (fun b : B => choice (Sur b))) f = id
```

Lastly, we eliminate the identity function and finish the proof with our choice equality.

```
bij_then_inv < rewrite id_ok.
2 subgoals

  A : Type
  B : Type
  f : Func A B
  Inj : injective f
  Sur : forall y : B, exists x : A, app f x = y
  b : B
  H := choice_ok (Sur b) : app f (choice (Sur b)) = b
  ============================
  app f (choice (Sur b)) = b

subgoal 2 is:
 comp (by_formula (fun b : B => choice (Sur b))) f = id

bij_then_inv < exact H.
1 subgoal

  A : Type
  B : Type
  f : Func A B
  Inj : injective f
  Sur : forall y : B, exists x : A, app f x = y
  ============================
  comp (by_formula (fun b : B => choice (Sur b))) f = id
```

For the other direction, we also use extensionality, rewrite composition as repeated application, and evaluate our formula.

```
bij_then_inv < apply funext.
1 subgoal

  A : Type
  B : Type
  f : Func A B
  Inj : injective f
  Sur : forall y : B, exists x : A, app f x = y
  ============================
  forall x : A,
  app (comp (by_formula (fun b : B => choice (Sur b))) f) x = app id x

bij_then_inv < intro a.
1 subgoal

  A : Type
  B : Type
  f : Func A B
  Inj : injective f
  Sur : forall y : B, exists x : A, app f x = y
  a : A
  ============================
  app (comp (by_formula (fun b : B => choice (Sur b))) f) a = app id a

bij_then_inv < rewrite comp_ok.
1 subgoal

  A : Type
  B : Type
  f : Func A B
  Inj : injective f
  Sur : forall y : B, exists x : A, app f x = y
  a : A
  ============================
  app (by_formula (fun b : B => choice (Sur b))) (app f a) = app id a

bij_then_inv < rewrite by_formula_ok.
1 subgoal

  A : Type
  B : Type
  f : Func A B
  Inj : injective f
  Sur : forall y : B, exists x : A, app f x = y
  a : A
  ============================
  choice (Sur (app f a)) = app id a
```

We again use `choice_ok` to extract the value from the surjectivity definition, and evaluate the identity function.

```
bij_then_inv < pose (H := choice_ok (Sur (app f a))).
1 subgoal

  A : Type
  B : Type
  f : Func A B
  Inj : injective f
  Sur : forall y : B, exists x : A, app f x = y
  a : A
  H := choice_ok (Sur (app f a))
    : (fun x : A => app f x = app f a) (choice (Sur (app f a)))
  ============================
  choice (Sur (app f a)) = app id a

bij_then_inv < simpl in H.
1 subgoal

  A : Type
  B : Type
  f : Func A B
  Inj : injective f
  Sur : forall y : B, exists x : A, app f x = y
  a : A
  H := choice_ok (Sur (app f a)) : app f (choice (Sur (app f a))) = app f a
  ============================
  choice (Sur (app f a)) = app id a

bij_then_inv < rewrite id_ok.
1 subgoal

  A : Type
  B : Type
  f : Func A B
  Inj : injective f
  Sur : forall y : B, exists x : A, app f x = y
  a : A
  H := choice_ok (Sur (app f a)) : app f (choice (Sur (app f a))) = app f a
  ============================
  choice (Sur (app f a)) = a
```

As the last step, we use injectivity, meaning that the two inputs in the goal will be equal if the outputs when f is applied are equal, and finish by using our choice equality.

```
bij_then_inv < apply Inj.
```

```
1 subgoal

  A : Type
  B : Type
  f : Func A B
  Inj : injective f
  Sur : forall y : B, exists x : A, app f x = y
  a : A
  H := choice_ok (Sur (app f a)) : app f (choice (Sur (app f a))) = app f a
  ============================
  app f (choice (Sur (app f a))) = app f a

bij_then_inv < exact H.
No more subgoals.

bij_then_inv < Qed.
(intros A B f Inj Sur).
(unfold invertible).
(unfold surjective in Sur).
exists (by_formula (fun b => choice (Sur b))).
split.
 (apply funext).
 intro b.
 (rewrite comp_ok).
 (rewrite by_formula_ok).
 (pose (H := choice_ok (Sur b))).
 (simpl in H).
 (rewrite id_ok).
 exact H.

 (apply funext).
 intro a.
 (rewrite comp_ok).
 (rewrite by_formula_ok).
 (pose (H := choice_ok (Sur (app f a)))).
 (simpl in H).
 (rewrite id_ok).
 (apply Inj).
 exact H.

Qed.
bij_then_inv is defined
```

We can also reproduce the proof that the composition of invertible functions is invertible, but first we will need to simple results. We need to establish that the identity function is an identity element for the composition operation. All we need to do is use extensionality, and the definitions of composition and the identity function.

```
Coq < Theorem id_left : forall {dom cod} (f : Func dom cod), comp f id = f.
1 subgoal

  ============================
  forall (dom cod : Type) (f : Func dom cod), comp f id = f

id_left < intros dom cod f.
1 subgoal

  dom : Type
  cod : Type
  f : Func dom cod
  ============================
  comp f id = f

id_left < apply funext.
1 subgoal
```

```
  dom : Type
  cod : Type
  f : Func dom cod
  ============================
  forall x : dom, app (comp f id) x = app f x

id_left < intro x.
1 subgoal

  dom : Type
  cod : Type
  f : Func dom cod
  x : dom
  ============================
  app (comp f id) x = app f x

id_left < rewrite comp_ok.
1 subgoal

  dom : Type
  cod : Type
  f : Func dom cod
  x : dom
  ============================
  app f (app id x) = app f x

id_left < rewrite id_ok.
1 subgoal

  dom : Type
  cod : Type
  f : Func dom cod
  x : dom
  ============================
  app f x = app f x

id_left < reflexivity.
No more subgoals.

id_left < Qed.
(intros dom cod f).
(apply funext).
intro x.
(rewrite comp_ok).
(rewrite id_ok).
reflexivity.

Qed.
id_left is defined
```

The reverse order is proved in exactly the same way.

```
Coq < Theorem id_right : forall {dom cod} (f : Func dom cod), comp id f = f.
1 subgoal

  ============================
  forall (dom cod : Type) (f : Func dom cod), comp id f = f

id_right < intros dom cod f.
1 subgoal

  dom : Type
  cod : Type
  f : Func dom cod
  ============================
  comp id f = f
```

```
id_right < apply funext.
1 subgoal

  dom : Type
  cod : Type
  f : Func dom cod
  ============================
  forall x : dom, app (comp id f) x = app f x

id_right < intro x.
1 subgoal

  dom : Type
  cod : Type
  f : Func dom cod
  x : dom
  ============================
  app (comp id f) x = app f x

id_right < rewrite comp_ok.
1 subgoal

  dom : Type
  cod : Type
  f : Func dom cod
  x : dom
  ============================
  app id (app f x) = app f x

id_right < rewrite id_ok.
1 subgoal

  dom : Type
  cod : Type
  f : Func dom cod
  x : dom
  ============================
  app f x = app f x

id_right < reflexivity.
No more subgoals.

id_right < Qed.
(intros dom cod f).
(apply funext).
intro x.
(rewrite comp_ok).
(rewrite id_ok).
reflexivity.

Qed.
id_right is defined
```

We start our proof for the composition of invertible functions by unfolding the defintion of invertibility, extracting witnesses from the existence statements, and constructing the inverse function $(g \circ f)^{-1}$, which is just the composition of the individual inverses $f^{-1} \circ g^{-1}$.

```
Coq < Theorem inv_closed_under_comp : forall {A B C} (g : Func B C) (f : Func A B),
  invertible g -> invertible f -> invertible (comp g f).
1 subgoal

  ============================
  forall (A B C : Type) (g : Func B C) (f : Func A B),
  invertible g -> invertible f -> invertible (comp g f)

inv_closed_under_comp < intros A B C g f Invg Invf.
```

```
1 subgoal

  A : Type
  B : Type
  C : Type
  g : Func B C
  f : Func A B
  Invg : invertible g
  Invf : invertible f
  ============================
  invertible (comp g f)

inv_closed_under_comp < unfold invertible.
1 subgoal

  A : Type
  B : Type
  C : Type
  g : Func B C
  f : Func A B
  Invg : invertible g
  Invf : invertible f
  ============================
  exists g0 : Func C A, comp (comp g f) g0 = id /\ comp g0 (comp g f) = id

inv_closed_under_comp < unfold invertible in Invg.
1 subgoal

  A : Type
  B : Type
  C : Type
  g : Func B C
  f : Func A B
  Invg : exists g0 : Func C B, comp g g0 = id /\ comp g0 g = id
  Invf : invertible f
  ============================
  exists g0 : Func C A, comp (comp g f) g0 = id /\ comp g0 (comp g f) = id

inv_closed_under_comp < unfold invertible in Invf.
1 subgoal

  A : Type
  B : Type
  C : Type
  g : Func B C
  f : Func A B
  Invg : exists g0 : Func C B, comp g g0 = id /\ comp g0 g = id
  Invf : exists g : Func B A, comp f g = id /\ comp g f = id
  ============================
  exists g0 : Func C A, comp (comp g f) g0 = id /\ comp g0 (comp g f) = id

inv_closed_under_comp < destruct Invf as [fi [Hf1 Hf2]].
1 subgoal

  A : Type
  B : Type
  C : Type
  g : Func B C
  f : Func A B
  Invg : exists g0 : Func C B, comp g g0 = id /\ comp g0 g = id
  fi : Func B A
  Hf1 : comp f fi = id
  Hf2 : comp fi f = id
  ============================
  exists g0 : Func C A, comp (comp g f) g0 = id /\ comp g0 (comp g f) = id

inv_closed_under_comp < destruct Invg as [gi [Hg1 Hg2]].
1 subgoal
```

```
  A : Type
  B : Type
  C : Type
  g : Func B C
  f : Func A B
  gi : Func C B
  Hg1 : comp g gi = id
  Hg2 : comp gi g = id
  fi : Func B A
  Hf1 : comp f fi = id
  Hf2 : comp fi f = id
  ============================
  exists g0 : Func C A, comp (comp g f) g0 = id /\ comp g0 (comp g f) = id

inv_closed_under_comp < exists (comp fi gi).
1 subgoal

  A : Type
  B : Type
  C : Type
  g : Func B C
  f : Func A B
  gi : Func C B
  Hg1 : comp g gi = id
  Hg2 : comp gi g = id
  fi : Func B A
  Hf1 : comp f fi = id
  Hf2 : comp fi f = id
  ============================
  comp (comp g f) (comp fi gi) = id /\ comp (comp fi gi) (comp g f) = id

inv_closed_under_comp < split.
2 subgoals

  A : Type
  B : Type
  C : Type
  g : Func B C
  f : Func A B
  gi : Func C B
  Hg1 : comp g gi = id
  Hg2 : comp gi g = id
  fi : Func B A
  Hf1 : comp f fi = id
  Hf2 : comp fi f = id
  ============================
  comp (comp g f) (comp fi gi) = id

subgoal 2 is:
 comp (comp fi gi) (comp g f) = id
```

For the first proof, since composition is associative, we can bring together each function and its inverse, producing the identity. We can also use our simple theorem to eliminate composition with the identity.

```
inv_closed_under_comp < rewrite comp_assoc.
2 subgoals

  A : Type
  B : Type
  C : Type
  g : Func B C
  f : Func A B
  gi : Func C B
  Hg1 : comp g gi = id
  Hg2 : comp gi g = id
  fi : Func B A
  Hf1 : comp f fi = id
  Hf2 : comp fi f = id
  ============================
  comp (comp (comp g f) fi) gi = id

subgoal 2 is:
 comp (comp fi gi) (comp g f) = id

inv_closed_under_comp < rewrite <- comp_assoc with (h := fi).
2 subgoals

  A : Type
  B : Type
  C : Type
  g : Func B C
  f : Func A B
  gi : Func C B
  Hg1 : comp g gi = id
  Hg2 : comp gi g = id
  fi : Func B A
  Hf1 : comp f fi = id
  Hf2 : comp fi f = id
  ============================
  comp (comp g (comp f fi)) gi = id

subgoal 2 is:
 comp (comp fi gi) (comp g f) = id

inv_closed_under_comp < rewrite Hf1.
2 subgoals

  A : Type
  B : Type
  C : Type
  g : Func B C
  f : Func A B
  gi : Func C B
  Hg1 : comp g gi = id
  Hg2 : comp gi g = id
  fi : Func B A
  Hf1 : comp f fi = id
  Hf2 : comp fi f = id
  ============================
  comp (comp g id) gi = id

subgoal 2 is:
 comp (comp fi gi) (comp g f) = id

inv_closed_under_comp < rewrite id_left.
2 subgoals

  A : Type
  B : Type
  C : Type
  g : Func B C
  f : Func A B
  gi : Func C B
  Hg1 : comp g gi = id
  Hg2 : comp gi g = id
  fi : Func B A
  Hf1 : comp f fi = id
  Hf2 : comp fi f = id
  ============================
  comp g gi = id

subgoal 2 is:
 comp (comp fi gi) (comp g f) = id
```

```
inv_closed_under_comp < rewrite Hg1.
2 subgoals

  A : Type
  B : Type
  C : Type
  g : Func B C
  f : Func A B
  gi : Func C B
  Hg1 : comp g gi = id
  Hg2 : comp gi g = id
  fi : Func B A
  Hf1 : comp f fi = id
  Hf2 : comp fi f = id
  ============================
  id = id

subgoal 2 is:
 comp (comp fi gi) (comp g f) = id

inv_closed_under_comp < reflexivity.
1 subgoal

  A : Type
  B : Type
  C : Type
  g : Func B C
  f : Func A B
  gi : Func C B
  Hg1 : comp g gi = id
  Hg2 : comp gi g = id
  fi : Func B A
  Hf1 : comp f fi = id
  Hf2 : comp fi f = id
  ============================
  comp (comp fi gi) (comp g f) = id
```

The second half is proved in exactly the same way, and the proof is complete.

```
inv_closed_under_comp < rewrite comp_assoc.
1 subgoal

  A : Type
  B : Type
  C : Type
  g : Func B C
  f : Func A B
  gi : Func C B
  Hg1 : comp g gi = id
  Hg2 : comp gi g = id
  fi : Func B A
  Hf1 : comp f fi = id
  Hf2 : comp fi f = id
  ============================
  comp (comp (comp fi gi) g) f = id

inv_closed_under_comp < rewrite <- comp_assoc with (h := g).
1 subgoal

  A : Type
  B : Type
  C : Type
  g : Func B C
  f : Func A B
  gi : Func C B
  Hg1 : comp g gi = id
  Hg2 : comp gi g = id
  fi : Func B A
  Hf1 : comp f fi = id
  Hf2 : comp fi f = id
  ============================
  comp (comp fi (comp gi g)) f = id

inv_closed_under_comp < rewrite Hg2.
1 subgoal

  A : Type
  B : Type
  C : Type
  g : Func B C
  f : Func A B
  gi : Func C B
  Hg1 : comp g gi = id
  Hg2 : comp gi g = id
  fi : Func B A
  Hf1 : comp f fi = id
  Hf2 : comp fi f = id
  ============================
  comp (comp fi id) f = id

inv_closed_under_comp < rewrite id_left.
1 subgoal

  A : Type
  B : Type
  C : Type
  g : Func B C
  f : Func A B
  gi : Func C B
  Hg1 : comp g gi = id
  Hg2 : comp gi g = id
  fi : Func B A
  Hf1 : comp f fi = id
  Hf2 : comp fi f = id
  ============================
  comp fi f = id

inv_closed_under_comp < rewrite Hf2.
1 subgoal

  A : Type
  B : Type
  C : Type
  g : Func B C
  f : Func A B
  gi : Func C B
  Hg1 : comp g gi = id
  Hg2 : comp gi g = id
  fi : Func B A
  Hf1 : comp f fi = id
  Hf2 : comp fi f = id
  ============================
  id = id

inv_closed_under_comp < reflexivity.
No more subgoals.

inv_closed_under_comp < Qed.
(intros A B C g f Invg Invf).
(unfold invertible).
(unfold invertible in Invg).
(unfold invertible in Invf).
(destruct Invf as [fi [Hf1 Hf2]]).
(destruct Invg as [gi [Hg1 Hg2]]).
exists (comp fi gi).
```

```
split.
 (rewrite comp_assoc).
 (rewrite <- comp_assoc with (h := fi)).
 (rewrite Hf1).
 (rewrite id_left).
 (rewrite Hg1).
 reflexivity.

 (rewrite comp_assoc).
 (rewrite <- comp_assoc with (h := g)).
 (rewrite Hg2).
 (rewrite id_left).
 (rewrite Hf2).
 reflexivity.

Qed.
inv_closed_under_comp is defined
```

3.4.1 Particular Functions

The Identity Function

We can use our methodology to prove things about particular functions. For instance, we can prove that the identity function is both injective and surjective. It has to be injective because it returns the input, so if the outputs are equal, the inputs must be equal.

```
Coq < Lemma id_inj : forall {A}, injective (@id A).
1 subgoal

  ============================
  forall A : Type, injective id

id_inj < intro A.
1 subgoal

  A : Type
  ============================
  injective id

id_inj < unfold injective.
1 subgoal

  A : Type
  ============================
  forall x y : A, app id x = app id y -> x = y

id_inj < intros x y H.
1 subgoal

  A : Type
  x, y : A
  H : app id x = app id y
  ============================
  x = y

id_inj < rewrite id_ok in H.
1 subgoal

  A : Type
  x, y : A
  H : x = app id y
  ============================
  x = y
```

```
id_inj < rewrite id_ok in H.
1 subgoal

  A : Type
  x, y : A
  H : x = y
  ============================
  x = y

id_inj < exact H.
No more subgoals.

id_inj < Qed.
intro A.
(unfold injective).
(intros x y H).
(rewrite id_ok in H).
(rewrite id_ok in H).
exact H.

Qed.
id_inj is defined
```

It is also surjective, because each element of the output type is mapped to by itself.

```
Coq < Lemma id_sur : forall {A}, surjective (@id A).
1 subgoal

  ============================
  forall A : Type, surjective id

id_sur < intro A.
1 subgoal

  A : Type
  ============================
  surjective id

id_sur < unfold surjective.
1 subgoal

  A : Type
  ============================
  forall y : A, exists x : A, app id x = y

id_sur < intro y.
1 subgoal

  A : Type
  y : A
  ============================
  exists x : A, app id x = y

id_sur < exists y.
1 subgoal

  A : Type
  y : A
  ============================
  app id y = y

id_sur < rewrite id_ok.
1 subgoal

  A : Type
  y : A
  ============================
  y = y

id_sur < reflexivity.
No more subgoals.

id_sur < Qed.
intro A.
(unfold surjective).
intro y.
exists y.
(rewrite id_ok).
reflexivity.

Qed.
id_sur is defined
```

The Shift Function

Let's consider the shift function $Sh : \mathbb{N} \to \mathbb{N}$ defined by $Sh(x) = x + 1$, which can also be written as the relation $Sh : \mathbb{N} \to \mathbb{N} \to \text{Prop}$ defined by $Sh(x, y) \Rightarrow x + 1 = y$. In Coq, we will write it as

```
  Definition plus_1 := fun x : nat => x + 1.
```

The shift function is injective, but not surjective. It is injective because $x_1 + 1 = x_2 + 1$ implies that $x_1 = x_2$, so no two natural numbers map to the same number. It is not surjective, however, because nothing maps to zero.

We start by stating the problem (plus_1_inj), unfolding our definitions, and introducing hypotheses. We use by_formula_ok to turn `Func` objects back into standard functions, and we will need standard arithmetic for the natural numbers.

```
Coq < Require Import Arith.
[Loading ML file z_syntax_plugin.cmxs ... done]
[Loading ML file quote_plugin.cmxs ... done]
[Loading ML file newring_plugin.cmxs ... done]

Coq < Theorem plus_1_inj : injective (by_formula plus_1).
1 subgoal

  ============================
  injective (by_formula plus_1)

  plus_1_inj < unfold injective.
  1 subgoal

    ============================
    forall x xp : nat,
    app (by_formula plus_1) x = app (by_formula plus_1) xp -> x = xp

  plus_1_inj < intros x xp.
  1 subgoal

    x, xp : nat
    ============================
    app (by_formula plus_1) x = app (by_formula plus_1) xp -> x = xp

  plus_1_inj < rewrite by_formula_ok.
  1 subgoal

    x, xp : nat
    ============================
    plus_1 x = app (by_formula plus_1) xp -> x = xp

  plus_1_inj < rewrite by_formula_ok.
  1 subgoal

    x, xp : nat
    ============================
    plus_1 x = plus_1 xp -> x = xp

  plus_1_inj < unfold plus_1.
  1 subgoal

    x, xp : nat
    ============================
    x + 1 = xp + 1 -> x = xp

  plus_1_inj < intro H.
  1 subgoal

    x, xp : nat
    H : x + 1 = xp + 1
    ============================
    x = xp
```

Now it turns out that one of the Peano Axioms, which define arithmetic with natural numbers, is that for all natural numbers m and n, $m = n$ if and only if $S(m) = S(n)$. That is, the successor function S is an injection. We will use this fact, and also rewrite our hypotheses in terms of S.

```
plus_1_inj < apply Nat.succ_inj.
1 subgoal

  x, xp : nat
  H : x + 1 = xp + 1
  ============================
  S x = S xp

plus_1_inj < repeat rewrite Nat.add_1_r in H.
1 subgoal

  x, xp : nat
  H : S x = S xp
  ============================
  S x = S xp

plus_1_inj < exact H.
No more subgoals.

plus_1_inj < Qed.
(unfold injective).
(intros x xp).
(rewrite by_formula_ok).
(rewrite by_formula_ok).
(unfold plus_1).
intro H.
Require Import Arith.
(apply Nat.succ_inj).
(repeat rewrite Nat.add_1_r in H).
exact H.

Qed.
plus_1_inj is defined
```

Proving that Sh is not surjective is simpler, but conceptually a little harder. We begin again by stating our problem (plus_1_not_sur), unfolding definitions, and introducing hypotheses.

```
Coq < Theorem plus_1_not_sur : ~surjective (by_formula plus_1).
1 subgoal

  ============================
  ~ surjective (by_formula plus_1)

plus_1_not_sur < unfold surjective.
1 subgoal

  ============================
  ~ (forall y : nat, exists x : nat, app (by_formula plus_1) x = y)

plus_1_not_sur < intro H.
1 subgoal

  H : forall y : nat, exists x : nat, app (by_formula plus_1) x = y
  ============================
  False
```

Since our goal is `False`, this will be a proof by contradiction. Our initial reasoning was that zero could never be the output of our function. Thus, we would like to consider our hypothesis H in the case that $y = 0$. We can get that statement by applying H to 0, which will leave us with an existence statement. We can use `destruct` on the existence statement to produce a witness, and these two operations can be combined in one Coq statement,

```
plus_1_not_sur < destruct (H 0) as [x Hx].
1 subgoal

  H : forall y : nat, exists x : nat, app (by_formula plus_1) x = y
  x : nat
  Hx : app (by_formula plus_1) x = 0
  ============================
  False

plus_1_not_sur < rewrite by_formula_ok in Hx.
1 subgoal

  H : forall y : nat, exists x : nat, app (by_formula plus_1) x = y
  x : nat
  Hx : plus_1 x = 0
  ============================
  False

plus_1_not_sur < unfold plus_1 in Hx.
1 subgoal

  H : forall y : nat, exists x : nat, app (by_formula plus_1) x = y
  x : nat
  Hx : x + 1 = 0
  ============================
  False

plus_1_not_sur < rewrite Nat.add_1_r in Hx.
1 subgoal

  H : forall y : nat, exists x : nat, app (by_formula plus_1) x = y
  x : nat
  Hx : S x = 0
  ============================
  False
```

which produces a witness x satisfying Hx, which is H for $y = 0$. We rewrite this new hypothesis in terms of S in order for Coq to apply the natural number axioms correctly. Now Coq can infer that no number x has 0 as a successor, by looking at the recursive definition of the natural numbers. This kind of introspection can be carried out by the `inversion` or `discriminate` tactics. Inversion tells us that `S x = 0` is false, and thus implies anything, so our proof by contradiction is complete.

```
plus_1_not_sur < inversion Hx.
No more subgoals.

plus_1_not_sur < Qed.
(unfold surjective).
intro H.
(destruct (H 0) as [x Hx]).
(rewrite by_formula_ok in Hx).
(unfold plus_1 in Hx).
(rewrite Nat.add_1_r in Hx).
(inversion Hx).

Qed.
plus_1_not_sur is defined
```

The Double Function

As a second example, we can look at the function that doubles its input

```
  Definition double := fun x : nat => 2 * x.
```

It is injective because we can always divide by two to get the original input (double_inj). To prove this, we first unfold the definitions and introduce hypotheses, and rewrite our hypothesis to get the equality we expect.

```
Coq < Theorem double_inj : injective (by_formula double).
1 subgoal

  ============================
  injective (by_formula double)

double_inj < unfold injective.
1 subgoal

  ============================
  forall x xp : nat,
  app (by_formula double) x = app (by_formula double) xp -> x = xp

double_inj < intros x xp H.
1 subgoal

  x, xp : nat
  H : app (by_formula double) x = app (by_formula double) xp
  ============================
  x = xp

double_inj < repeat rewrite by_formula_ok in H.
1 subgoal

  x, xp : nat
  H : double x = double xp
  ============================
  x = xp

double_inj < unfold double in H.
1 subgoal

  x, xp : nat
  H : 2 * x = 2 * xp
  ============================
  x = xp
```

It would be nice if we had a function like `Nat.half` that we could apply to both sides of `H2` using `f_equal`. That does not exist in the standard library, so we look for another lemma,

```
double_inj < Search (_ * _).
H2: 2 * x2 = 2 * x1
H1: 2 * x1 = y
mult_assoc_reverse: forall n m p : nat, n * m * p = n * (m * p)
mult_is_O: forall n m : nat, n * m = 0 -> n = 0 \/ m = 0
mult_is_one: forall n m : nat, n * m = 1 -> n = 1 /\ m = 1
...
Nat.mul_cancel_l: forall n m p : nat, p <> 0 -> p * n = p * m <-> n = m
Nat.mul_cancel_r: forall n m p : nat, p <> 0 -> n * p = m * p <-> n = m
...
```

We can apply the cancellation lemma, and use symmetry of equality in the goal to prove our statement,

```
double_inj < apply Nat.mul_cancel_l in H.
2 subgoals

  x, xp : nat
  H : x = xp
  ============================
  x = xp

subgoal 2 is:
 2 <> 0

double_inj < exact H.
1 subgoal

  x, xp : nat
  H : 2 * x = 2 * xp
  H0 : forall n m p : nat, p <> 0 -> p * n = p * m -> n = m
  ============================
  2 <> 0
```

but we still have to prove the assumption from the cancellation lemma.

```
double_inj < intro H1.
1 subgoal

  x, xp : nat
  H : 2 * x = 2 * xp
  H0 : forall n m p : nat, p <> 0 -> p * n = p * m -> n = m
  H1 : 2 = 0
  ============================
  False

double_inj < discriminate.
No more subgoals.

double_inj < Qed.
(unfold injective).
(intros x xp H).
(repeat rewrite by_formula_ok in H).
(unfold double in H).
(apply Nat.mul_cancel_l in H).
 exact H.

 intro H1.
 discriminate.
```

```
Qed.
double_inj is defined
```

The double function is not surjective (double_not_sur) because nothing can be doubled to give one, since one half is not a natural number. To prove this, we will need an auxiliary library of tactics

```
  Require Import Lia.
```

and then we can unfold the definitions and introduce hypotheses. We then apply our hypothesis to the case $y = 1$ and extract the witness.

```
Coq < Theorem double_not_sur: ~surjective (by_formula double).
1 subgoal

  ============================
  ~ surjective (by_formula double)

double_not_sur < unfold surjective.
1 subgoal

  ============================
  ~ (forall y : nat, exists x : nat, app (by_formula double) x = y)

double_not_sur < intro H.
1 subgoal

  H : forall y : nat, exists x : nat, app (by_formula double) x = y
  ============================
  False

double_not_sur < destruct (H 1) as [x Hx].
1 subgoal

  H : forall y : nat, exists x : nat, app (by_formula double) x = y
  x : nat
  Hx : app (by_formula double) x = 1
  ============================
  False

double_not_sur < rewrite by_formula_ok in Hx.
1 subgoal

  H : forall y : nat, exists x : nat, app (by_formula double) x = y
  x : nat
  Hx : double x = 1
  ============================
  False

double_not_sur < unfold double in Hx.
1 subgoal

  H : forall y : nat, exists x : nat, app (by_formula double) x = y
  x : nat
  Hx : 2 * x = 1
  ============================
  False
```

At this point we would like Coq to recognize that `H1` is impossible. It is possible to prove this by hand, but it is quite laborious. Instead, we will use the `lia` tactic which can solve linear equations and inequalities over the natural numbers.

```
double_not_sur < lia.
No more subgoals.
```

```
double_not_sur < Qed.
(unfold surjective).
intro H.
(destruct (H 1) as [x Hx]).
(rewrite by_formula_ok in Hx).
(unfold double in Hx).
lia.

Qed.
double_not_sur is defined
```

3.5 Automating Cantor's Proof

In order to automate the proof of Cantor's Theorem from Section 3.1.1 (Cantor), we will show that no function f mapping a set to its power set is surjective. Remember that the power set $\mathcal{P}(X)$ of a set X is a collection of all the subsets of X. One way of thinking of a subset is as a function mapping each member of the set to a proposition, true for inclusion and false for exclusion. Thus members of the power set can be thought of as functions from X to Prop. Cantor's Theorem thus shows that no function f from X to $X \to$ Prop can be surjective.

We will first prove this from the relational point of view. In order to reuse our previous definition 3.33, we write surjective as a predicate on relations.

```
Coq < Theorem Cantor X : ~exists f : X -> X -> Prop, surjective (fun (x : X) (y : X -> Prop) => (f x = y)).
1 subgoal

  X : Type
  ============================
  ~(exists f : X -> X -> Prop, surjective (fun (x : X) (y : X -> Prop) => f x = y))
```

We begin the proof by eliminating the negation, extracting the witness from the existence hypothesis, and unfolding the definition of surjectivity.

```
Cantor < intro.
1 subgoal

  X : Type
  H : exists f : X -> X -> Prop,
        surjective (fun (x : X) (y : X -> Prop) => f x = y)
  ============================
  False

Cantor < destruct H as [f A].
1 subgoal

  X : Type
  f : X -> X -> Prop
  A : surjective (fun (x : X) (y : X -> Prop) => f x = y)
  ============================
  False

Cantor < unfold surjective in A.
1 subgoal

  X : Type
  f : X -> X -> Prop
  A : forall y : X -> Prop, exists x : X, f x = y
  ============================
  False
```

We now insert the special function that Cantor proposed to define our set T,

```
Cantor < pose (g := fun x : X => ~ f x x).
1 subgoal

  X : Type
  f : X -> X -> Prop
  A : forall y : X -> Prop, exists x : X, f x = y
  g := fun x : X => ~ f x x : X -> Prop
  ============================
  False
```

We can apply our hypothesis A to it, which generates an existence statement from which we extract a witness x. What this means is that there has to be some element x of X such that our function f maps it to the subset defined g. This is the meaning of surjectivity.

```
Cantor < destruct (A g) as [x B].
1 subgoal

  X : Type
  f : X -> X -> Prop
  A : forall y : X -> Prop, exists x : X, f x = y
  g := fun x : X => ~ f x x : X -> Prop
  x : X
  B : f x = g
  ============================
  False
```

Using B, we can prove a simple auxiliary hypothesis C using that witness, which we assert,

```
Cantor < assert (C : g x <-> f x x).
2 subgoals

  X : Type
  f : X -> X -> Prop
  A : forall y : X -> Prop, exists x : X, f x = y
  g := fun x : X => ~ f x x : X -> Prop
  x : X
  B : f x = g
  ============================
  g x <-> f x x

subgoal 2 is:
 False

Cantor < rewrite B.
2 subgoals

  X : Type
  f : X -> X -> Prop
  A : forall y : X -> Prop, exists x : X, f x = y
  g := fun x : X => ~ f x x : X -> Prop
  x : X
  B : f x = g
  ============================
  g x <-> g x

subgoal 2 is:
 False

Cantor < reflexivity.
1 subgoal

  X : Type
  f : X -> X -> Prop
  A : forall y : X -> Prop, exists x : X, f x = y
  g := fun x : X => ~ f x x : X -> Prop
  x : X
  B : f x = g
  C : g x <-> f x x
  ============================
  False
```

Now we unfold the definition of g in our hypothesis C to expose a contradiction, and our theorem is proved.

```
Cantor < unfold g in C.
1 subgoal

  X : Type
  f : X -> X -> Prop
  A : forall y : X -> Prop, exists x : X, f x = y
  g := fun x : X => ~ f x x : X -> Prop
  x : X
  B : f x = g
  C : ~ f x x <-> f x x
  ============================
  False

Cantor < tauto.
No more subgoals.

Cantor < Qed.
intro.
(destruct H as [f A]).
(unfold surjective in A).
(pose (g := fun x : X => ~ f x x)).
(destruct (A g) as [x B]).
(assert (C : g x <-> f x x)).
 (rewrite B).
 reflexivity.

 (unfold g in C).
 tauto.

Qed.
Cantor is defined
```

We used `tauto` to prove the contradiction, but we could have done this by hand,

```
Coq < Lemma test (S : Prop) : (S <-> ~S) -> False.
1 subgoal

  S : Prop
  ============================
  S <-> ~ S -> False

test < Qed.
intro.
(destruct H).
(unfold not in H).
(apply H).
 (apply H0).
 intro.
 (apply H).
  assumption.

  assumption.

 (apply H0).
 intro.
 (apply H).
  assumption.
  assumption.

Qed.
```

We can now rewrite our proof of Cantor's Theorem using the new definition of functions. We create the same "diagonal" function, but this time we can use a formula. It proceeds along the same line as our early proof, but with easier to read function representations.

```
Coq < Theorem Cantor : forall {A} (f : Func A (Func A Prop)), ~surjective f.
1 subgoal

  ============================
  forall (A : Type) (f : Func A (Func A bool)), ~ surjective f

Cantor < intros A f Sur.
1 subgoal

  A : Type
  f : Func A (Func A Prop)
  Sur : surjective f
  ============================
  False

Cantor < pose (g := by_formula (fun a => ~(app (app f a) a))).
1 subgoal

  A : Type
  f : Func A (Func A Prop)
  Sur : surjective f
  g := by_formula (fun a : A => ~ app (app f a) a) : Func A Prop
  ============================
  False

Cantor < destruct (Sur g) as [a H].
1 subgoal

  A : Type
  f : Func A (Func A Prop)
  Sur : surjective f
  g := by_formula (fun a : A => ~ app (app f a) a) : Func A Prop
  a : A
  H : app f a = g
  ============================
  False

Cantor < assert (H2 : app (app f a) a <-> app g a).
2 subgoals

  A : Type
  f : Func A (Func A Prop)
  Sur : surjective f
  g := by_formula (fun a : A => ~ app (app f a) a) : Func A Prop
  a : A
  H : app f a = g
  ============================
  app (app f a) a <-> app g a

subgoal 2 is:
 False

Cantor < split.
3 subgoals

  A : Type
  f : Func A (Func A Prop)
  Sur : surjective f
  g := by_formula (fun a : A => ~ app (app f a) a) : Func A Prop
  a : A
  H : app f a = g
  ============================
  app (app f a) a -> app g a

subgoal 2 is:
 app g a -> app (app f a) a
subgoal 3 is:
 False

Cantor < rewrite H.
3 subgoals

  A : Type
  f : Func A (Func A Prop)
  Sur : surjective f
  g := by_formula (fun a : A => ~ app (app f a) a) : Func A Prop
  a : A
  H : app f a = g
  ============================
  app g a -> app g a

subgoal 2 is:
 app g a -> app (app f a) a
subgoal 3 is:
 False

Cantor < intro H2.
3 subgoals

  A : Type
  f : Func A (Func A Prop)
  Sur : surjective f
  g := by_formula (fun a : A => ~ app (app f a) a) : Func A Prop
  a : A
  H : app f a = g
  H2 : app g a
  ============================
  app g a

subgoal 2 is:
 app g a -> app (app f a) a
subgoal 3 is:
 False

Cantor < exact H2.
2 subgoals

  A : Type
  f : Func A (Func A Prop)
  Sur : surjective f
  g := by_formula (fun a : A => ~ app (app f a) a) : Func A Prop
  a : A
  H : app f a = g
  ============================
  app g a -> app (app f a) a

subgoal 2 is:
 False

Cantor < rewrite H.
2 subgoals

  A : Type
  f : Func A (Func A Prop)
  Sur : surjective f
  g := by_formula (fun a : A => ~ app (app f a) a) : Func A Prop
  a : A
  H : app f a = g
  ============================
  app g a -> app g a

subgoal 2 is:
 False

Cantor < intro H2.
2 subgoals

  A : Type
  f : Func A (Func A Prop)
  Sur : surjective f
  g := by_formula (fun a : A => ~ app (app f a) a) : Func A Prop
  a : A
  H : app f a = g
  H2 : app g a
  ============================
  app g a

subgoal 2 is:
 False

Cantor < exact H2.
1 subgoal

  A : Type
  f : Func A (Func A Prop)
  Sur : surjective f
  g := by_formula (fun a : A => ~ app (app f a) a) : Func A Prop
  a : A
  H : app f a = g
  H2 : app (app f a) a <-> app g a
  ============================
  False

Cantor < unfold g in H2.
1 subgoal

  A : Type
  f : Func A (Func A Prop)
  Sur : surjective f
  g := by_formula (fun a : A => ~ app (app f a) a) : Func A Prop
  a : A
  H : app f a = g
  H2 : app (app f a) a <->
       app (by_formula (fun a : A => ~ app (app f a) a)) a
  ============================
  False

Cantor < rewrite by_formula_ok in H2.
1 subgoal

  A : Type
  f : Func A (Func A Prop)
  Sur : surjective f
  g := by_formula (fun a : A => ~ app (app f a) a) : Func A Prop
  a : A
  H : app f a = g
  H2 : app (app f a) a <-> ~ app (app f a) a
  ============================
  False
```

We can prove this last step with `tauto`, but here we will do it by hand.

```
Cantor < assert (H3 : forall P : Prop, ~(P <-> ~P)).
2 subgoals
```

```
  A : Type
  f : Func A (Func A Prop)
  Sur : surjective f
  g := by_formula (fun a : A => ~ app (app f a) a) : Func A Prop
  a : A
  H : app f a = g
  H2 : app (app f a) a <-> ~ app (app f a) a
  ============================
  forall P : Prop, ~ (P <-> ~ P)

subgoal 2 is:
 False

Cantor < intros P H3.
2 subgoals

  A : Type
  f : Func A (Func A Prop)
  Sur : surjective f
  g := by_formula (fun a : A => ~ app (app f a) a) : Func A Prop
  a : A
  H : app f a = g
  H2 : app (app f a) a <-> ~ app (app f a) a
  P : Prop
  H3 : P <-> ~ P
  ============================
  False

subgoal 2 is:
 False

Cantor < destruct H3.
2 subgoals

  A : Type
  f : Func A (Func A Prop)
  Sur : surjective f
  g := by_formula (fun a : A => ~ app (app f a) a) : Func A Prop
  a : A
  H : app f a = g
  H2 : app (app f a) a <-> ~ app (app f a) a
  P : Prop
  H0 : P -> ~ P
  H1 : ~ P -> P
  ============================
  False

subgoal 2 is:
 False

Cantor < apply H0.
3 subgoals

  A : Type
  f : Func A (Func A Prop)
  Sur : surjective f
  g := by_formula (fun a : A => ~ app (app f a) a) : Func A Prop
  a : A
  H : app f a = g
  H2 : app (app f a) a <-> ~ app (app f a) a
  P : Prop
  H0 : P -> ~ P
  H1 : ~ P -> P
  ============================
  P

subgoal 2 is:
 P
subgoal 3 is:
 False

Cantor < apply H1.
3 subgoals

  A : Type
  f : Func A (Func A Prop)
  Sur : surjective f
  g := by_formula (fun a : A => ~ app (app f a) a) : Func A Prop
  a : A
  H : app f a = g
  H2 : app (app f a) a <-> ~ app (app f a) a
  P : Prop
  H0 : P -> ~ P
  H1 : ~ P -> P
  ============================
  ~ P

subgoal 2 is:
 P
subgoal 3 is:
 False

Cantor < intro p.
3 subgoals

  A : Type
  f : Func A (Func A Prop)
  Sur : surjective f
  g := by_formula (fun a : A => ~ app (app f a) a) : Func A Prop
  a : A
  H : app f a = g
  H2 : app (app f a) a <-> ~ app (app f a) a
  P : Prop
  H0 : P -> ~ P
  H1 : ~ P -> P
  p : P
  ============================
  False

subgoal 2 is:
 P
subgoal 3 is:
 False

Cantor < apply H0.
4 subgoals

  A : Type
  f : Func A (Func A Prop)
  Sur : surjective f
  g := by_formula (fun a : A => ~ app (app f a) a) : Func A Prop
  a : A
  H : app f a = g
  H2 : app (app f a) a <-> ~ app (app f a) a
  P : Prop
  H0 : P -> ~ P
  H1 : ~ P -> P
  p : P
  ============================
  P

subgoal 2 is:
 P
subgoal 3 is:
 P
```

```
subgoal 4 is:
 False

Cantor < exact p.
3 subgoals

  A : Type
  f : Func A (Func A Prop)
  Sur : surjective f
  g := by_formula (fun a : A => ~ app (app f a) a) : Func A Prop
  a : A
  H : app f a = g
  H2 : app (app f a) a <-> ~ app (app f a) a
  P : Prop
  H0 : P -> ~ P
  H1 : ~ P -> P
  p : P
  ============================
  P

subgoal 2 is:
 P
subgoal 3 is:
 False

Cantor < exact p.
2 subgoals

  A : Type
  f : Func A (Func A Prop)
  Sur : surjective f
  g := by_formula (fun a : A => ~ app (app f a) a) : Func A Prop
  a : A
  H : app f a = g
  H2 : app (app f a) a <-> ~ app (app f a) a
  P : Prop
  H0 : P -> ~ P
  H1 : ~ P -> P
  ============================
  P

subgoal 2 is:
 False

Cantor < apply H1.
2 subgoals

  A : Type
  f : Func A (Func A Prop)
  Sur : surjective f
  g := by_formula (fun a : A => ~ app (app f a) a) : Func A Prop
  a : A
  H : app f a = g
  H2 : app (app f a) a <-> ~ app (app f a) a
  P : Prop
  H0 : P -> ~ P
  H1 : ~ P -> P
  ============================
  ~ P

subgoal 2 is:
 False

Cantor < intro p.
2 subgoals

  A : Type
  f : Func A (Func A Prop)
  Sur : surjective f
  g := by_formula (fun a : A => ~ app (app f a) a) : Func A Prop
  a : A
  H : app f a = g
  H2 : app (app f a) a <-> ~ app (app f a) a
  P : Prop
  H0 : P -> ~ P
  H1 : ~ P -> P
  p : P
  ============================
  False

subgoal 2 is:
 False

Cantor < apply H0.
3 subgoals

  A : Type
  f : Func A (Func A Prop)
  Sur : surjective f
  g := by_formula (fun a : A => ~ app (app f a) a) : Func A Prop
  a : A
  H : app f a = g
  H2 : app (app f a) a <-> ~ app (app f a) a
  P : Prop
  H0 : P -> ~ P
  H1 : ~ P -> P
  p : P
  ============================
  P

subgoal 2 is:
 P
subgoal 3 is:
 False

Cantor < exact p.
2 subgoals

  A : Type
  f : Func A (Func A Prop)
  Sur : surjective f
  g := by_formula (fun a : A => ~ app (app f a) a) : Func A Prop
  a : A
  H : app f a = g
  H2 : app (app f a) a <-> ~ app (app f a) a
  P : Prop
  H0 : P -> ~ P
  H1 : ~ P -> P
  p : P
  ============================
  P

subgoal 2 is:
 False

Cantor < exact p.
1 subgoal

  A : Type
  f : Func A (Func A Prop)
  Sur : surjective f
  g := by_formula (fun a : A => ~ app (app f a) a) : Func A Prop
  a : A
  H : app f a = g
  H2 : app (app f a) a <-> ~ app (app f a) a
  H3 : forall P : Prop, ~ (P <-> ~ P)
  ============================
  False

Cantor < pose (H4 := H3 (app (app f a) a)).
1 subgoal

  A : Type
  f : Func A (Func A Prop)
  Sur : surjective f
  g := by_formula (fun a : A => ~ app (app f a) a) : Func A Prop
  a : A
  H : app f a = g
  H2 : app (app f a) a <-> ~ app (app f a) a
  H3 : forall P : Prop, ~ (P <-> ~ P)
  H4 := H3 (app (app f a) a) : ~ (app (app f a) a <-> ~ app (app f a) a)
  ============================
  False

Cantor < apply H4.
1 subgoal

  A : Type
  f : Func A (Func A Prop)
  Sur : surjective f
  g := by_formula (fun a : A => ~ app (app f a) a) : Func A Prop
  a : A
  H : app f a = g
  H2 : app (app f a) a <-> ~ app (app f a) a
  H3 : forall P : Prop, ~ (P <-> ~ P)
  H4 := H3 (app (app f a) a) : ~ (app (app f a) a <-> ~ app (app f a) a)
  ============================
  app (app f a) a <-> ~ app (app f a) a

Cantor < exact H2.
No more subgoals.

Cantor < Qed.
(intros A f Sur).
(pose (g := by_formula (fun a => ~ app (app f a) a))).
(destruct (Sur g) as [a H]).
(assert (H2 : app (app f a) a <-> app g a)).
 split.
  (rewrite H).
  intro H2.
  exact H2.

  (rewrite H).
  intro H2.
  exact H2.

 (unfold g in H2).
 (rewrite by_formula_ok in H2).
 (assert (H3 : forall P : Prop, ~ (P <-> ~ P))).
  (intros P H3).
  (destruct H3).
  (apply H0).
   (apply H1).
   intro p.
   (apply H0).
    exact p.

    exact p.

   (apply H1).
   intro p.
   (apply H0).
    exact p.
    exact p.

  (pose (H4 := H3 (app (app f a) a))).
  (apply H4).
  exact H2.

Qed.
Cantor is defined
```

Another way of looking at this proof is to imagine programs instead of sets. We can think of a program as some function that takes binary input (which we can encode as some natural number) and produces binary output (another natural number). We imagine that we can number all possible programs (since they too are binary code), so that we could produce the function executed by that program given its number. We will prove that this function is not surjective, meaning that some function cannot be computed by any program, or they are *uncomputable*. Notice that we use the successor function instead of negation to make each diagonal element different.

```
Coq < Theorem uncomputable : forall (f : Func nat (Func nat nat)), ~surjective f.
1 subgoal

  ============================
  forall f : Func nat (Func nat nat), ~ surjective f

uncomputable < intros f Sur.
1 subgoal

  f : Func nat (Func nat nat)
  Sur : surjective f
  ============================
  False

uncomputable < pose (k := by_formula (fun n => S (app (app f n) n))).
1 subgoal

  f : Func nat (Func nat nat)
  Sur : surjective f
  k := by_formula (fun n : nat => S (app (app f n) n)) : Func nat nat
  ============================
  False

uncomputable < destruct (Sur k) as [n H].
1 subgoal

  f : Func nat (Func nat nat)
  Sur : surjective f
  k := by_formula (fun n : nat => S (app (app f n) n)) : Func nat nat
  n : nat
  H : app f n = k
  ============================
  False

uncomputable < pose (m := app (app f n) n).
1 subgoal

  f : Func nat (Func nat nat)
  Sur : surjective f
  k := by_formula (fun n : nat => S (app (app f n) n)) : Func nat nat
  n : nat
  H : app f n = k
  m := app (app f n) n : nat
  ============================
  False
```

```
uncomputable < assert (H2 : m = app k n).
2 subgoals

  f : Func nat (Func nat nat)
  Sur : surjective f
  k := by_formula (fun n : nat => S (app (app f n) n)) : Func nat nat
  n : nat
  H : app f n = k
  m := app (app f n) n : nat
  ============================
  m = app k n

subgoal 2 is:
 False

uncomputable < unfold m.
2 subgoals

  f : Func nat (Func nat nat)
  Sur : surjective f
  k := by_formula (fun n : nat => S (app (app f n) n)) : Func nat nat
  n : nat
  H : app f n = k
  m := app (app f n) n : nat
  ============================
  app (app f n) n = app k n

subgoal 2 is:
 False

uncomputable < rewrite H.
2 subgoals

  f : Func nat (Func nat nat)
  Sur : surjective f
  k := by_formula (fun n : nat => S (app (app f n) n)) : Func nat nat
  n : nat
  H : app f n = k
  m := app (app f n) n : nat
  ============================
  app k n = app k n

subgoal 2 is:
 False

uncomputable < reflexivity.
1 subgoal

  f : Func nat (Func nat nat)
  Sur : surjective f
  k := by_formula (fun n : nat => S (app (app f n) n)) : Func nat nat
  n : nat
  H : app f n = k
  m := app (app f n) n : nat
  H2 : m = app k n
  ============================
  False

uncomputable < unfold k in H2.
1 subgoal

  f : Func nat (Func nat nat)
  Sur : surjective f
  k := by_formula (fun n : nat => S (app (app f n) n)) : Func nat nat
  n : nat
  H : app f n = k
  m := app (app f n) n : nat
  H2 : m = app (by_formula (fun n : nat => S (app (app f n) n))) n
  ============================
  False

uncomputable < rewrite by_formula_ok in H2.
1 subgoal

  f : Func nat (Func nat nat)
  Sur : surjective f
  k := by_formula (fun n : nat => S (app (app f n) n)) : Func nat nat
  n : nat
  H : app f n = k
  m := app (app f n) n : nat
  H2 : m = S (app (app f n) n)
  ============================
  False

uncomputable < fold m in H2.
1 subgoal

  f : Func nat (Func nat nat)
  Sur : surjective f
  k := by_formula (fun n : nat => S (app (app f n) n)) : Func nat nat
  n : nat
  H : app f n = k
  m := app (app f n) n : nat
  H2 : m = S m
  ============================
  False

uncomputable < pose (H3 := n_Sn m).
1 subgoal

  f : Func nat (Func nat nat)
  Sur : surjective f
  k := by_formula (fun n : nat => S (app (app f n) n)) : Func nat nat
  n : nat
  H : app f n = k
  m := app (app f n) n : nat
  H2 : m = S m
  H3 := n_Sn m : m <> S m
  ============================
  False

uncomputable < contradiction.
No more subgoals.

uncomputable < Qed.
(intros f Sur).
(pose (k := by_formula (fun n => S (app (app f n) n)))).
(destruct (Sur k) as [n H]).
(pose (m := app (app f n) n)).
(assert (H2 : m = app k n)).
 (unfold m).
 (rewrite H).
 reflexivity.

 (unfold k in H2).
 (rewrite by_formula_ok in H2).
 (fold m in H2).
 (pose (H3 := n_Sn m)).
 contradiction.

Qed.
uncomputable is defined
```

3.6 Problems

Problem III.1 What is the cardinality of a union, $|A \cup B|$? Express it in terms of $|A|$, $|B|$, and any other operators you need.

Problem III.2 Construct the following relations on our domain $\{1, 2, 3\}$,

- A relation that is both symmetric and antisymmetric.
- A relation that is transitive and symmetric, but not reflexive or antisymmetric.
- A relation that is both transitive and not symmetric.

Problem III.3 A *partition* of a set A is a list of one or more non-empty subsets of A such that each element of A appears in exactly one of the subsets. Notice that this means that these subsets are pairwise disjoint.

Suppose that we are given an equivalence relation on a set A. We will show that this relation generates a partition of the set. Each of these subsets $[a]$ is called an *equivalence class*, and is defined as

$$[a] = \{b \in A \mid b \sim a\} \tag{3.37}$$

where $a \in A$.

1. Prove that $\forall a \in A, a \in [a]$.
2. Prove that $\forall a, b \in A, [a] = [b] \vee [a] \cap [b] = \emptyset$, which means every two equivalence classes $[a]$ and $[b]$ are either equal or disjoint.
3. Prove that the equivalence classes form a partition.

Problem III.4 The function $f(x) = x^2$ for $x \in \mathbb{N}$ is injective, but not surjective. To prove injectivity in Coq, start with

```
Definition square := fun x y : nat => x*x = y.
```

and generate **Proof:** *sq_inj*

```
Theorem sq_inj : injective square.
```

You will likely need the `f_equal`, `Nat.sqrt`, and `Nat.sqrt_square` theorems from Coq.

Next, generate **Proof:** *sq_not_sur*

```
Theorem sq_not_sur : ~surjective square.
```

I recommend using the `two_not_sq` proof from the text showing that 2 cannot be the square of any number.

Problem III.5 The constant function $f(x) = c$ for $x \in \mathbb{N}$ is not injective or surjective. For Coq, again we define our function

```
Definition constant {X:Type} (c:X) := fun x y : X => y = c.
```

Generate **Proof:** *const_not_inj*,

```
Theorem const_not_inj : forall c:nat, ~injective (constant c).
```

Here, I used the trick of asserting an absurd hypothesis in the form, $A \implies \mathrm{F}$, where A is a false statement, in order to generate a proof by contradiction.

Next, generate **Proof:** *const_not_sur*,

```
Theorem const_not_sur : forall c:nat, ~surjective (constant c).
```

Here you may find it useful to use the `n_Sn` theorem from the Coq standard library.

Problem III.6 The inverse of a bijective function is bijective. In Coq, you can start with a definition of the inverse of a function from Eq. (3.34), and generate **Proof:** *inverse_bij*,

```
Theorem inverse_bij {X Y:Type} : forall (f : relation X Y) (g : relation Y X),
  bijective f -> inverse f g -> bijective g.
```

Note that the `inverse_exists` proof in the text asserts that such a map exists. Here you need only show it is bijective.

Problem III.7 The composition of two bijective functions is bijective. In Coq, you can start with a definition of the composition of two functions from Eq. (3.35), and generate **Proof:** *comp_bij*,

```
Theorem comp_bij {X Y Z : Type} : forall (f : relation X Z) (g : relation Z Y) (h : relation X Y),
  bijective f -> bijective g -> composition f g h -> bijective h.
```

Note that the `composition_exists` proof in the text asserts that such a map exists. Here you need only show it is bijective.

Problem III.8 How many distinct Boolean functions of n variables exist?

Problem III.9 Prove that a function from S to $\mathcal{P}(S)$ cannot be surjective. This is somewhat involved using constructive logic, so students may also prove it using a non-constructive argument based on the sizes of the sets.

Problem III.10 Prove that the relation "has a bijection to" is an equivalence relation on sets in the class of sets $\mathcal{S}$. We can define this property formally as

```
Definition hasBijection (X Y : Type) := exists R : relation X Y, bijective R.
```

and then our statement, **Proof:** *bij_equiv*, becomes

```
Theorem bij_equiv : equivalence hasBijection.
```

The proofs done in Problems 6 and 7 may be used here, and I also suggest using the `inverse_exists` and `comp_exists` theorems that we proved in this section.

Problem III.11 We would like to prove that the composition of surjective functions is surjective.

```
Theorem sur_closed_under_comp : forall {A B C} (g : Func B C) (f : Func A B),
  surjective g -> surjective f -> surjective (comp g f).
```

Problem III.12 We would like to prove that the composition of injective functions is injective.

```
Theorem inj_closed_under_comp : forall {A B C} (g : Func B C) (f : Func A B),
  injective g -> injective f -> injective (comp g f).
```

Problem III.13 Show that invertible functions must be injective.

```
Theorem inv_then_inj : forall {A B} (f : Func A B), invertible f -> injective f.
```

Problem III.14 Show that invertible functions must be surjective.

```
Theorem inv_then_sur : forall {A B} (f : Func A B), invertible f -> surjective f.
```

Problem III.15 Prove that the relation "has a bijection to" is an equivalence relation on sets in the class of sets $\mathcal{S}$. We can define this property formally as

```
Definition has_bijection (A B : Type) := exists (f : Func A B), injective f /\ surjective f.
```

and then our statement, **Proof:** *bij_equiv*, becomes

```
Theorem bij_equiv : equivalence has_bijection.
```

Problem III.16 Let $f : \mathcal{D} \to \mathcal{C}$ and $g, h : \mathcal{C} - > \mathcal{D}$ be functions. The function f together with g and h form a *quasi-inverse* if

$$f \circ g = \mathrm{Id}$$
$$h \circ f = \mathrm{Id}$$

Clearly, every invertible function and its inverse form a quasi-inverse. As it turns out, every quasi-inverse determines an invertible function. We will prove that in any quasi-inverse (f, g, h), we have $g = h$. Hint: Consider using the `id_left` and `id_right` theorems.

```
Theorem quasi : forall {dom cod} (f : Func dom cod) (g : Func cod dom) (h : Func cod dom),
  comp f g = id -> comp h f = id -> g = h.
```

Problem III.17 The constant function $f(x) = c$ for $x \in A$ is not injective or surjective. For Coq, again we define our function

```
Definition constant {A} (c : A) := by_formula (fun x : A => c).
```

Generate **Proof:** *const_not_inj*,

```
Theorem const_not_inj : forall c : nat, ~injective (constant c).
```

Here, we should think about what arguments to the injective hypothesis would generate a contradiction.

Next, generate **Proof:** *const_not_sur*,

```
Theorem const_not_sur : forall c : nat, ~surjective (constant c).
```

Here you may find it useful to use the `n_Sn` theorem from the Coq standard library.

Chapter 4

Mathematical Induction

I think some intuition leaks out in every step of an induction proof.

— Jim Propp

If we have no idea why a statement is true, we can still prove it by induction.

— Gian-Carlo Rota

Induction makes you feel guilty for getting something out of nothing, and it is artificial, but it is one of the greatest ideas of civilization.

— Herbert Wilf

4.1 Well Ordering

The Well Ordering Principle is

Every nonempty set of nonnegative integers has a smallest element.

It appears self-evident, but is the basis for much of our reasoning about integers and rational numbers. Notice that the empty set is not well-ordered, because having no elements, it has no smallest element. It does not apply to sets of negative integers, as they clearly have no lower bound. It also does not apply to sets of positive rational numbers, as you can always find another rational between any given rational number and zero.

An excellent template for proofs using the Well Ordering Principle is given in (Lehman, Leighton, and Meyer 2015). To prove $\forall n \in \mathbb{N}, P(n)$ using the Well Ordering Principle,

- Define the set C of counterexamples to P being true. Specifically, define

$$C := \{n \in \mathbb{N} \mid \neg P(n)\} \tag{4.1}$$

- Assume for proof by contradiction that C is nonempty.
- By the Well Ordering Principle, there will be a smallest element, n, in C.
- Reach a contradiction somehow, often by showing that $P(n)$ is actually true or by showing that there is another member of C that is smaller than n. This is the open-ended part of the proof task.
- Conclude that C must be empty, that is, no counterexamples exist.

As an example, we can use the Well Ordering Principle to prove that every non-prime natural number $n > 1$ can be factored into a product of primes. To begin, we define C as the set of non-prime natural numbers which cannot be factored into a product of primes, and assume that it is non-empty. By the Well Ordering Principle, it must have a least element n_0. Since n_0 is not prime, it can be factored into two numbers a and b, which are both less than n_0. Since $a, b < n_0$ we conclude that $a, b \notin C$ and a can be factored as a product of primes $p_{a,1} \cdots p_{a,k}$, as can b. However, this means that $n_0 = p_{a,1} \cdots p_{a,k} p_{b,1} \cdots p_{b,l}$, contradicting the claim that $n_0 \in C$, or proving that

$$(n_0 \in C) \wedge (n_0 \notin C) = \mathrm{F}. \tag{4.2}$$

4.2 Induction

Mathematical induction is based on our simple `apply`, or modus ponens, construction

$$\begin{array}{l} P \\ P \implies Q \\ \hline \therefore Q \end{array}$$

however, we would like to use many hypotheses,

$$\begin{array}{l} P_0 \\ P_0 \implies P_1 \\ P_1 \implies P_2 \\ \vdots \\ P_{k-1} \implies P_k \\ \hline \therefore P_k \end{array}$$

The same reasoning pattern still holds, k `apply` tactics and the initial assumption. In fact, we can extend this to a countable number of hypotheses,

$$\begin{array}{l} P_0 \\ \forall k \in \mathbb{N}, P_{k-1} \implies P_k \\ \hline \therefore \forall k \in \mathbb{N}, P_k \end{array}$$

Typically, we define the propositions P_k using a predicate P over the natural numbers. We can write induction over the natural numbers more formally as

$$\begin{aligned} &\forall P : \text{nat} \implies \text{Prop}, \\ &\quad P(0) \implies (\forall n : \text{nat}, P(n) \implies P(Sn)) \implies \forall n : \text{nat}, P(n). \end{aligned} \tag{4.3}$$

An induction proofs usually proceeds in several steps. First, define the predicate P which should be true for each k, which is referred to as the *inductive hypothesis*. Then prove the proposition $P(0)$, which is called the *base case*. Next, prove the implication $P(k) \implies P(k+1)$, commonly called the *inductive step*, and finally invoke induction to justify the conclusion.

As an example, consider the rule to differentiate products of function, also called the Leibnitz Rule,

$$(fg)' = f'g + fg'. \tag{4.4}$$

We can generalize this rule to the product of n functions,

$$\begin{aligned} (f_0 f_1 \cdots f_n)' &= f_0' f_1 \cdots f_n + f_0 f_1' f_2 \cdots f_n + \cdots + f_0 \cdots f_{n-1} f_n' && (4.5) \\ &= \sum_{i=0}^{n} \frac{\prod_{j=0}^{n} f_j}{f_i} f_i' && (4.6) \end{aligned}$$

and use induction to justify this. We will first do this in the old-style of induction proofs. The predicate P for the inductive hypothesis is

$$P(n) := (f_0 f_1 \cdots f_{n+1})' = \sum_{i=0}^{n+1} \frac{\prod_{j=0}^{n+1} f_j}{f_i} f_i'. \tag{4.7}$$

The trivial case $(n = -1)$ is immediate

$$P(-1) := (f_0)' = f_0', \tag{4.8}$$

but we will need to make our base case the Leibnitz Rule

$$P(0) := (f_0 f_1)' = f_0' f_1 + f_0 f_1'. \tag{4.9}$$

Then for $n \geq 0$, assuming $P(n)$, we have

$$P(n+1) \tag{4.10}$$

$$:= (f_0 f_1 \cdots f_{n+2})' = \sum_{i=0}^{n+2} \frac{\prod_{j=0}^{n+2} f_j}{f_i} f_i' \tag{4.11}$$

$$:= (\prod_{j=0}^{n+1} f_j)' f_{n+2} + (\prod_{j=0}^{n+1} f_j) f_{n+2}' = \sum_{i=0}^{n+1} \frac{\prod_{j=0}^{n+2} f_j}{f_i} f_i' + \frac{\prod_{j=0}^{n+2} f_j}{f_{n+2}} f_{n+2}' \tag{4.12}$$

$$:= (\prod_{j=0}^{n+1} f_j)' f_{n+2} + (\prod_{j=0}^{n+1} f_j) f_{n+2}' = f_{n+2} \sum_{i=0}^{n+1} \frac{\prod_{j=0}^{n+1} f_j}{f_i} f_i' + (\prod_{j=0}^{n+1} f_j) f_{n+2}' \tag{4.13}$$

$$:= (\prod_{j=0}^{n+1} f_j)' f_{n+2} = f_{n+2} \sum_{i=0}^{n+1} \frac{\prod_{j=0}^{n+1} f_j}{f_i} f_i' \tag{4.14}$$

$$:= (\prod_{j=0}^{n+1} f_j)' = \sum_{i=0}^{n+1} \frac{\prod_{j=0}^{n+1} f_j}{f_i} f_i' \tag{4.15}$$

$$:= P(n). \tag{4.16}$$

In the second step, we use the Leibnitz Rule, $P(0)$, on the left hand side, and split off the last term in the sum on the right hand side. In the third step, we pull out f_{n+2} from the sum and cancel it from the last term on the right hand side. In the fourth step, we subtract the same term from each side of the equality, and in the fifth step we cancel f_{n+2} from each side. Thus our inductive step is proved, and appealing to induction our theorem is true.

We can rewrite this proof in our informal style, meaning using the style of the proof assistant, but allowing assertions that we do not prove, such as cumbersome arithmetic expressions. We start again with our predicate on the natural numbers,

$$P(n) := (f_0 f_1 \cdots f_{n+1})' = \sum_{i=0}^{n+1} \frac{\prod_{j=0}^{n+1} f_j}{f_i} f_i', \tag{4.17}$$

and we would like to prove

$G0 : \forall n : nat, P(n)$

Step	Tactic

In order to do this, we will apply the induction theorem for natural numbers

$$\forall P : \text{nat} \to \text{Prop}, P(0) \to (\forall n : \text{nat}, P(n) \to P(n+1)) \to \forall n : \text{nat}, P(n)$$

which in Coq is

```
Coq < Print nat_ind.
nat_ind =
fun P : nat -> Prop => nat_rect P
     : forall P : nat -> Prop,
       P 0 -> (forall n : nat, P n -> P (S n)) -> forall n : nat, P n
```

We can apply this theorem, or more precisely the specialization of this theorem to P, to our goal

$$\begin{aligned} G0 :& \forall n : \text{nat}, P(n) \\ G1 :& P(0) \wedge (\forall n : \text{nat}, P(n) \implies P(n+1)) \end{aligned}$$

Step	Tactic
1	apply (nat_ind P)

We need to prove both preconditions for the theorem, so we split the proof into two pieces.

$$\begin{aligned} G0 :& \forall n : \text{nat}, P(n) \\ G1 :& P(0) \wedge (\forall n : \text{nat}, P(n) \implies P(n+1)) \\ G2a :& P(0) \\ G2b :& \forall n : \text{nat}, P(n) \implies P(n+1) \end{aligned}$$

Step	Tactic
1	apply (nat_ind P)
2	split

The first goal can be proved using the Leibniz Rule. This is what we mean by an informal proof, namely that we can use the Leibniz Rule without proof. The second goal can be simplified by introducing the antecedent, which is the inductive hypothesis.

$$\begin{aligned} n :& \text{nat} \\ IHn :& P(n) \end{aligned}$$

$$\begin{aligned} G0 :& \forall n : \text{nat}, P(n) \\ G1 :& P(0) \wedge (\forall n : \text{nat}, P(n) \implies P(n+1)) \\ G2a :& P(0) \\ G2b :& \forall n : \text{nat}, P(n) \implies P(n+1) \\ G3b :& P(n+1) \end{aligned}$$

Step	Tactic
1	apply (nat_ind P)
2	split
3a	Leibniz Rule
3b	intros n IHn

Finally, since this is an informal proof, we can use our derivation about to prove that after some algebra, $P(n+1) = P(n)$, and our proof is done.

n :nat
IHn :$P(n)$

$G0$:$\forall n : \text{nat}, P(n)$
$G1$:$P(0) \wedge (\forall n : \text{nat}, P(n) \implies P(n+1))$
$G2a$:$P(0)$
$G2b$:$\forall n : \text{nat}, P(n) \implies P(n+1)$
$G3b$:$P(n+1)$
$G4b$:$P(n)$

Step	Tactic
1	apply (nat_ind P)
2	split
3a	Leibnitz Rule
3b	intros n IHn
4b	Eq. (4.16)
5b	exact IHn

Induction is naturally related to recursion, in the sense that recursive algorithms can often be proved correct using induction. For example, we can construct a recursive algorithm for determining if two natural numbers are the same. As we learned in Section 2.3.2, zero is a designated natural number, and all others are built from zero with the successor function. We can therefore distinguish three cases:

1. both numbers are zero,
2. one number is zero, and the other is not,
3. both numbers are successor of some number.

We can encode this logic in a binary predicate using the `Fixpoint` construct,

```
Fixpoint nat_eq (n : nat) (m : nat) : Prop :=
  match n, m with
  | 0, 0 => True
  | 0, S m => False
  | S n, 0 => False
  | S n, S m => nat_eq n m
  end.
```

.

We can prove that our predicate is implied by equailty using induction. First, we state the lemma, introduce variables, and begin the induction on m. Note that we clear our initial hypthesis. We need to do this so that we do not have have extra assumptions in the induction.

```
Coq < Lemma eq_imp_nat_eq : forall n m : nat, n = m -> nat_eq n m.
1 subgoal

  ============================
  forall n m : nat, n = m -> nat_eq n m

eq_imp_nat_eq < intros n m H.
1 subgoal

  n, m : nat
  H : n = m
  ============================
  nat_eq n m
```

```
eq_imp_nat_eq < rewrite H.
1 subgoal

  n, m : nat
  H : n = m
  ============================
  nat_eq m m

eq_imp_nat_eq < clear H.
1 subgoal

  n, m : nat
  ============================
  nat_eq m m

eq_imp_nat_eq < induction m.
2 subgoals

  n : nat
  ============================
  nat_eq 0 0

subgoal 2 is:
 nat_eq (S m) (S m)
```

The base case $m = 0$ can be handled by simplifcation.

```
eq_imp_nat_eq < simpl.
2 subgoals

  n : nat
  ============================
  True

subgoal 2 is:
 nat_eq (S m) (S m)

eq_imp_nat_eq < trivial.
1 subgoal

  n, m : nat
  IHm : nat_eq m m
  ============================
  nat_eq (S m) (S m)
```

The induction step is proved using simplification and the induction hypothesis.

```
eq_imp_nat_eq < simpl.
1 subgoal

  n, m : nat
  IHm : nat_eq m m
  ============================
  nat_eq m m

eq_imp_nat_eq < exact IHm.
No more subgoals.

eq_imp_nat_eq < Qed.
(intros n m H).
(rewrite H).
clear H.
(induction m).
 (simpl).
 trivial.

 (simpl).
 exact IHm.
```

```
Qed.
eq_imp_nat_eq is defined
```

We can prove the other direction, but we will need induction on both variables. We begin induction on n, but to prove the base case we will need induction on m.

```
Coq < Lemma nat_eq_imp_eq : forall n m, nat_eq n m -> n = m.
1 subgoal

  ============================
  forall n m : nat, nat_eq n m -> n = m

nat_eq_imp_eq < induction n.
2 subgoals

  ============================
  forall m : nat, nat_eq 0 m -> 0 = m

subgoal 2 is:
 forall m : nat, nat_eq (S n) m -> S n = m

nat_eq_imp_eq < induction m.
3 subgoals

  ============================
  nat_eq 0 0 -> 0 = 0

subgoal 2 is:
 nat_eq 0 (S m) -> 0 = S m
subgoal 3 is:
 forall m : nat, nat_eq (S n) m -> S n = m
```

The base case $m = 0$ is trivial.

```
nat_eq_imp_eq < intro H.
3 subgoals

  H : nat_eq 0 0
  ============================
  0 = 0

subgoal 2 is:
 nat_eq 0 (S m) -> 0 = S m
subgoal 3 is:
 forall m : nat, nat_eq (S n) m -> S n = m

nat_eq_imp_eq < reflexivity.
2 subgoals

  m : nat
  IHm : nat_eq 0 m -> 0 = m
  ============================
  nat_eq 0 (S m) -> 0 = S m

subgoal 2 is:
 forall m : nat, nat_eq (S n) m -> S n = m
```

The induction step is true because the premise is false, namely that `S m` can be zero.

```
nat_eq_imp_eq < intro H.
2 subgoals

  m : nat
  IHm : nat_eq 0 m -> 0 = m
  H : nat_eq 0 (S m)
  ============================
  0 = S m

subgoal 2 is:
 forall m : nat, nat_eq (S n) m -> S n = m

nat_eq_imp_eq < simpl in H.
2 subgoals

  m : nat
  IHm : nat_eq 0 m -> 0 = m
  H : False
  ============================
  0 = S m

subgoal 2 is:
 forall m : nat, nat_eq (S n) m -> S n = m

nat_eq_imp_eq < contradiction.
1 subgoal

  n : nat
  IHn : forall m : nat, nat_eq n m -> n = m
  ============================
  forall m : nat, nat_eq (S n) m -> S n = m
```

For the induction step for n, we again use induction on m, and the base case is true by contradiction.

```
nat_eq_imp_eq < induction m.
2 subgoals

  n : nat
  IHn : forall m : nat, nat_eq n m -> n = m
  ============================
  nat_eq (S n) 0 -> S n = 0

subgoal 2 is:
 nat_eq (S n) (S m) -> S n = S m

nat_eq_imp_eq < intro H.
2 subgoals

  n : nat
  IHn : forall m : nat, nat_eq n m -> n = m
  H : nat_eq (S n) 0
  ============================
  S n = 0

subgoal 2 is:
 nat_eq (S n) (S m) -> S n = S m

nat_eq_imp_eq < simpl in H.
2 subgoals

  n : nat
  IHn : forall m : nat, nat_eq n m -> n = m
  H : False
  ============================
  S n = 0

subgoal 2 is:
 nat_eq (S n) (S m) -> S n = S m

nat_eq_imp_eq < contradiction.
1 subgoal
```

```
  n : nat
  IHn : forall m : nat, nat_eq n m -> n = m
  m : nat
  IHm : nat_eq (S n) m -> S n = m
  ============================
  nat_eq (S n) (S m) -> S n = S m
```

For the remaining induction step, we use simplification and then the induction hypothesis for n.

```
nat_eq_imp_eq < intro H.
1 subgoal

  n : nat
  IHn : forall m : nat, nat_eq n m -> n = m
  m : nat
  IHm : nat_eq (S n) m -> S n = m
  H : nat_eq (S n) (S m)
  ============================
  S n = S m

nat_eq_imp_eq < simpl in H.
1 subgoal

  n : nat
  IHn : forall m : nat, nat_eq n m -> n = m
  m : nat
  IHm : nat_eq (S n) m -> S n = m
  H : nat_eq n m
  ============================
  S n = S m

nat_eq_imp_eq < apply IHn in H.
1 subgoal

  n : nat
  IHn : forall m : nat, nat_eq n m -> n = m
  m : nat
  IHm : nat_eq (S n) m -> S n = m
  H : n = m
  ============================
  S n = S m

nat_eq_imp_eq < rewrite H.
1 subgoal

  n : nat
  IHn : forall m : nat, nat_eq n m -> n = m
  m : nat
  IHm : nat_eq (S n) m -> S n = m
  H : n = m
  ============================
  S m = S m

nat_eq_imp_eq < reflexivity.
No more subgoals.

nat_eq_imp_eq < Qed.
(induction n).
 (induction m).
  intro H.
  reflexivity.

  intro H.
  (simpl in H).
  contradiction.

 (induction m).
```

```
  intro H.
  (simpl in H).
  contradiction.

  intro H.
  (simpl in H).
  (apply IHn in H).
  (rewrite H).
  reflexivity.

Qed.
nat_eq_imp_eq is defined
```

We can use our predicate to prove fundamental properties of the natural number system. For example, we can prove that the successor function is injective, meaning that if the successors of two numbers are equal, then those numbers themselves are equal. We do this by converting from equality to our predicate, we explicitly encodes this behavior.

```
Coq < Lemma suc_inj : forall n m, S n = S m -> n = m.
1 subgoal

  ============================
  forall n m : nat, S n = S m -> n = m

suc_inj < intros n m H.
1 subgoal

  n, m : nat
  H : S n = S m
  ============================
  n = m

suc_inj < apply nat_eq_imp_eq.
1 subgoal

  n, m : nat
  H : S n = S m
  ============================
  nat_eq n m

suc_inj < apply eq_imp_nat_eq in H.
1 subgoal

  n, m : nat
  H : nat_eq (S n) (S m)
  ============================
  nat_eq n m

suc_inj < simpl in H.
1 subgoal

  n, m : nat
  H : nat_eq n m
  ============================
  nat_eq n m

suc_inj < exact H.
No more subgoals.

suc_inj < Qed.
(intros n m H).
(apply nat_eq_imp_eq).
(apply eq_imp_nat_eq in H).
(simpl in H).
exact H.
```

```
Qed.
suc_inj is defined
```

We can also prove that zero is not the successor of any number, again because we explicitly encoded this in our equality predicate.

```
Coq < Lemma O_neq_suc : forall n, 0 <> S n.
1 subgoal

  ============================
  forall n : nat, 0 <> S n

O_neq_suc < intros n H.
1 subgoal

  n : nat
  H : 0 = S n
  ============================
  False

O_neq_suc < apply eq_imp_nat_eq in H.
1 subgoal

  n : nat
  H : nat_eq 0 (S n)
  ============================
  False

O_neq_suc < simpl in H.
1 subgoal

  n : nat
  H : False
  ============================
  False

O_neq_suc < exact H.
No more subgoals.

O_neq_suc < Qed.
(intros n H).
(apply eq_imp_nat_eq in H).
(simpl in H).
exact H.

Qed.
O_neq_suc is defined
```

4.3 Strong Induction

When we introduced the induction derivation, we left some information on the table. Instead of just using $P(k-1)$ to derive $P(k)$, we could use all the prior propositions. Formally, *strong induction*, is given by

$$\begin{array}{l} P_0 \\ \forall k \in \mathbb{N}, \bigwedge_{j=0}^{k-1} P_j \implies P_k \\ \hline \therefore \forall k \in \mathbb{N}, P_k \end{array}$$

This is very useful when addressing inductive hypotheses that depend on more than one prior result. However, all strong inductive proofs can be mechanically rewritten as simple inductive proofs by altering the inductive hypothesis.

As an example of strong induction, let us show that any natural number $m \geq 8$ can be written as a sum of a multiple of 3 and 5. The induction hypothesis will be

$$P(n) := \exists a, b \in \mathbb{N}, n + 8 = 3a + 5b \tag{4.18}$$

where we use $m = n + 8$ so that $m \geq 8$ for all natural numbers n. Our base case for $n = 0$ is

$$\begin{aligned} P(0) &:= \exists a, b \in \mathbb{N}, 8 = 3a + 5b, && (4.19)\\ &:= 8 = 3(1) + 5(1). && (4.20) \end{aligned}$$

However, since this is strong induction, we will need additional base cases

$$\begin{aligned} P(1) &:= \exists a, b \in \mathbb{N}, 9 = 3a + 5b, && (4.21)\\ &:= 9 = 3(3) + 5(0). && (4.22) \end{aligned}$$

and

$$\begin{aligned} P(2) &:= \exists a, b \in \mathbb{N}, 10 = 3a + 5b, && (4.23)\\ &:= 10 = 3(0) + 5(2). && (4.24) \end{aligned}$$

For the inductive step $n \geq 2$, we assume $P(k)$ holds for all $k \leq n$, and prove that $P(n+1)$ holds,

$$\begin{aligned} P(n+1) &:= \exists a, b \in \mathbb{N}, n + 1 + 8 = 3a + 5b, && (4.25)\\ &:= \exists a, b \in \mathbb{N}, n - 2 + 8 + 3 = 3a + 5b, && (4.26)\\ &:= \exists a, b \in \mathbb{N}, 3r + 5s + 3 = 3a + 5b, && (4.27)\\ &:= \exists a, b \in \mathbb{N}, 3(r+1) + 5s = 3a + 5b. && (4.28) \end{aligned}$$

In the second step, we use the identity $1 = 3 - 2$. In the third step, since $n \geq 2$, we have that $n - 2$ is a natural number and that $n + 8$ is expressable as $3r + 5s$ for some $r, s \in \mathbb{N}$. In the fourth step, we show that $a = r + 1$ and $b = s$. Thus, by strong induction, our theorem is proved.

We will now do the same thing in the Coq, but we will change it slightly so as to have fewer base cases. Suppose I would like to prove that all natural numbers greater than one can be decomposed into a sum of multiples of two and three,

```
Theorem two_and_three : forall n : nat, 2 <= n -> exists (a b : nat), n = 2 * a + 3 * b.
```

I would like to retain all my previous hypotheses, but the `induction` tactic does not support this. Instead, we will prove the result in steps, where we bound the size of n and gradually increase this using induction on the bound.

```
Coq < Lemma two_and_three_strong : forall (m n : nat), n <= m -> 2 <= n -> exists (a b : nat), n = 2 * a + 3 * b.
1 subgoal

  ============================
  forall m n : nat, n <= m -> 2 <= n -> exists a b : nat, n = 2 * a + 3 * b

two_and_three_strong < induction m ; intros.
2 subgoals

  n : nat
  H0 : n <= 0
  H1 : 2 <= n
  ============================
  exists a b : nat, n = 2 * a + 3 * b

subgoal 2 is:
 exists a b : nat, n = 2 * a + 3 * b
```

The first thing we see is that n can only be zero, but that violates our requirement that n be greater than two, so the base case is proved.

```
two_and_three_strong < inversion H0.
2 subgoals

  n : nat
  H0 : n <= 0
  H1 : 2 <= n
  H2 : n = 0
  ============================
  exists a b : nat, 0 = 2 * a + 3 * b

subgoal 2 is:
 exists a b : nat, n = 2 * a + 3 * b

two_and_three_strong < rewrite H2 in H1.
2 subgoals

  n : nat
  H0 : n <= 0
  H1 : 2 <= 0
  H2 : n = 0
  ============================
  exists a b : nat, 0 = 2 * a + 3 * b

subgoal 2 is:
 exists a b : nat, n = 2 * a + 3 * b

two_and_three_strong < inversion H1.
1 subgoal

  m : nat
  IHm : forall n : nat,
        n <= m -> 2 <= n -> exists a b : nat, n = 2 * a + 3 * b
  n : nat
  H0 : n <= S m
  H1 : 2 <= n
  ============================
  exists a b : nat, n = 2 * a + 3 * b
```

Now we must prove all the base cases for n, which we do by destructing n several times. The first two values of n are illegal,

```
two_and_three_strong < destruct n.
2 subgoals

  m : nat
  IHm : forall n : nat,
      n <= m -> 2 <= n -> exists a b : nat, n = 2 * a + 3 * b
  H0 : 0 <= S m
  H1 : 2 <= 0
  ============================
  exists a b : nat, 0 = 2 * a + 3 * b

subgoal 2 is:
 exists a b : nat, S n = 2 * a + 3 * b

two_and_three_strong < inversion H1.
1 subgoal

  m : nat
  IHm : forall n : nat,
      n <= m -> 2 <= n -> exists a b : nat, n = 2 * a + 3 * b
  n : nat
  H0 : S n <= S m
  H1 : 2 <= S n
  ============================
  exists a b : nat, S n = 2 * a + 3 * b

two_and_three_strong < destruct n.
2 subgoals

  m : nat
  IHm : forall n : nat,
      n <= m -> 2 <= n -> exists a b : nat, n = 2 * a + 3 * b
  H0 : 1 <= S m
  H1 : 2 <= 1
  ============================
  exists a b : nat, 1 = 2 * a + 3 * b

subgoal 2 is:
 exists a b : nat, S (S n) = 2 * a + 3 * b

two_and_three_strong < inversion H1.
2 subgoals

  m : nat
  IHm : forall n : nat,
      n <= m -> 2 <= n -> exists a b : nat, n = 2 * a + 3 * b
  H0 : 1 <= S m
  H1 : 2 <= 1
  m0 : nat
  H3 : 2 <= 0
  H2 : m0 = 0
  ============================
  exists a b : nat, 1 = 2 * a + 3 * b

subgoal 2 is:
 exists a b : nat, S (S n) = 2 * a + 3 * b

two_and_three_strong < inversion H3.
1 subgoal

  m : nat
  IHm : forall n : nat,
      n <= m -> 2 <= n -> exists a b : nat, n = 2 * a + 3 * b
  n : nat
  H0 : S (S n) <= S m
  H1 : 2 <= S (S n)
  ============================
  exists a b : nat, S (S n) = 2 * a + 3 * b
```

The next two values of n are the base cases two and three,

```
two_and_three_strong < destruct n.
2 subgoals
```

```
  m : nat
  IHm : forall n : nat,
        n <= m -> 2 <= n -> exists a b : nat, n = 2 * a + 3 * b
  H0 : 2 <= S m
  H1 : 2 <= 2
  ============================
  exists a b : nat, 2 = 2 * a + 3 * b

subgoal 2 is:
 exists a b : nat, S (S (S n)) = 2 * a + 3 * b

two_and_three_strong < exists 1. exists 0.
2 subgoals

  m : nat
  IHm : forall n : nat,
        n <= m -> 2 <= n -> exists a b : nat, n = 2 * a + 3 * b
  H0 : 2 <= S m
  H1 : 2 <= 2
  ============================
  exists b : nat, 2 = 2 * 1 + 3 * b

subgoal 2 is:
 exists a b : nat, S (S (S n)) = 2 * a + 3 * b

2 subgoals

  m : nat
  IHm : forall n : nat,
        n <= m -> 2 <= n -> exists a b : nat, n = 2 * a + 3 * b
  H0 : 2 <= S m
  H1 : 2 <= 2
  ============================
  2 = 2 * 1 + 3 * 0

subgoal 2 is:
 exists a b : nat, S (S (S n)) = 2 * a + 3 * b

two_and_three_strong < reflexivity.
1 subgoal

  m : nat
  IHm : forall n : nat,
        n <= m -> 2 <= n -> exists a b : nat, n = 2 * a + 3 * b
  n : nat
  H0 : S (S (S n)) <= S m
  H1 : 2 <= S (S (S n))
  ============================
  exists a b : nat, S (S (S n)) = 2 * a + 3 * b

two_and_three_strong < destruct n.
2 subgoals

  m : nat
  IHm : forall n : nat,
        n <= m -> 2 <= n -> exists a b : nat, n = 2 * a + 3 * b
  H0 : 3 <= S m
  H1 : 2 <= 3
  ============================
  exists a b : nat, 3 = 2 * a + 3 * b

subgoal 2 is:
 exists a b : nat, S (S (S (S n))) = 2 * a + 3 * b

two_and_three_strong < exists 0. exists 1.
2 subgoals
```

```
  m : nat
  IHm : forall n : nat,
        n <= m -> 2 <= n -> exists a b : nat, n = 2 * a + 3 * b
  H0 : 3 <= S m
  H1 : 2 <= 3
  ============================
  exists b : nat, 3 = 2 * 0 + 3 * b

subgoal 2 is:
 exists a b : nat, S (S (S (S n))) = 2 * a + 3 * b

2 subgoals

  m : nat
  IHm : forall n : nat,
        n <= m -> 2 <= n -> exists a b : nat, n = 2 * a + 3 * b
  H0 : 3 <= S m
  H1 : 2 <= 3
  ============================
  3 = 2 * 0 + 3 * 1

subgoal 2 is:
 exists a b : nat, S (S (S (S n))) = 2 * a + 3 * b

two_and_three_strong < reflexivity.
1 subgoal

  m : nat
  IHm : forall n : nat,
        n <= m -> 2 <= n -> exists a b : nat, n = 2 * a + 3 * b
  n : nat
  H0 : S (S (S (S n))) <= S m
  H1 : 2 <= S (S (S (S n)))
  ============================
  exists a b : nat, S (S (S (S n))) = 2 * a + 3 * b
```

Now we can finally use our induction hypothesis. Just as we did in our proof by hand, we will use this hypothesis when the argument is $n + 2$,

```
two_and_three_strong < pose (IHn := IHm (S (S n))).
1 subgoal

  m : nat
  IHm : forall n : nat,
        n <= m -> 2 <= n -> exists a b : nat, n = 2 * a + 3 * b
  n : nat
  IHn := IHm (S (S n))
      : S (S n) <= m ->
        2 <= S (S n) -> exists a b : nat, S (S n) = 2 * a + 3 * b
  ============================
  S (S (S (S n))) <= S m ->
  2 <= S (S (S (S n))) -> exists a b : nat, S (S (S (S n))) = 2 * a + 3 * b
```

Calling destruct on `IHn` lets us prove the preconditions in order to get the existence statement we want.

```
two_and_three_strong < destruct IHn.
3 subgoals

  m : nat
  IHm : forall n : nat,
        n <= m -> 2 <= n -> exists a b : nat, n = 2 * a + 3 * b
  n : nat
  H0 : S (S (S (S n))) <= S m
  H1 : 2 <= S (S (S (S n)))
  ============================
  S (S n) <= m
```

```
subgoal 2 is:
 2 <= S (S n)
subgoal 3 is:
 exists a b : nat, S (S (S (S n))) = 2 * a + 3 * b
```

The first inequality can be proved using the transitivity of the `le` relation,

```
two_and_three_strong < apply le_S_n in H0.
3 subgoals

  m : nat
  IHm : forall n : nat,
        n <= m -> 2 <= n -> exists a b : nat, n = 2 * a + 3 * b
  n : nat
  H0 : S (S (S n)) <= m
  H1 : 2 <= S (S (S (S n)))
  ============================
  S (S n) <= m

subgoal 2 is:
 2 <= S (S n)
subgoal 3 is:
 exists a b : nat, S (S (S (S n))) = 2 * a + 3 * b

two_and_three_strong < apply (le_trans (S (S n)) (S (S (S n))) m).
4 subgoals

  m : nat
  IHm : forall n : nat,
        n <= m -> 2 <= n -> exists a b : nat, n = 2 * a + 3 * b
  n : nat
  H0 : S (S (S n)) <= m
  H1 : 2 <= S (S (S (S n)))
  ============================
  S (S n) <= S (S (S n))

subgoal 2 is:
 S (S (S n)) <= m
subgoal 3 is:
 2 <= S (S n)
subgoal 4 is:
 exists a b : nat, S (S (S (S n))) = 2 * a + 3 * b
```

We can apply the `le` theorems here, but we can also use the `constructor` tactic, which applies the correct theorem at each step that would have generated the current goal.

```
two_and_three_strong < repeat constructor.
3 subgoals

  m : nat
  IHm : forall n : nat,
        n <= m -> 2 <= n -> exists a b : nat, n = 2 * a + 3 * b
  n : nat
  H0 : S (S (S n)) <= m
  H1 : 2 <= S (S (S (S n)))
  ============================
  S (S (S n)) <= m

subgoal 2 is:
 2 <= S (S n)
subgoal 3 is:
 exists a b : nat, S (S (S (S n))) = 2 * a + 3 * b

two_and_three_strong < exact H0.
2 subgoals
```

```
  m : nat
  IHm : forall n : nat,
        n <= m -> 2 <= n -> exists a b : nat, n = 2 * a + 3 * b
  n : nat
  H0 : S (S (S (S n))) <= S m
  H1 : 2 <= S (S (S (S n)))
  ============================
  2 <= S (S n)

subgoal 2 is:
 exists a b : nat, S (S (S (S n))) = 2 * a + 3 * b
```

The next inequality is a consequence of the fact that all natural numbers are non-negative

```
two_and_three_strong < repeat apply le_n_S. apply Nat.le_0_l.
2 subgoals

  m : nat
  IHm : forall n : nat,
        n <= m -> 2 <= n -> exists a b : nat, n = 2 * a + 3 * b
  n : nat
  H0 : S (S (S (S n))) <= S m
  H1 : 2 <= S (S (S (S n)))
  ============================
  0 <= n

subgoal 2 is:
 exists a b : nat, S (S (S (S n))) = 2 * a + 3 * b

1 subgoal

  m : nat
  IHm : forall n : nat,
        n <= m -> 2 <= n -> exists a b : nat, n = 2 * a + 3 * b
  n : nat
  H0 : S (S (S (S n))) <= S m
  H1 : 2 <= S (S (S (S n)))
  x : nat
  H2 : exists b : nat, S (S n) = 2 * x + 3 * b
  ============================
  exists a b : nat, S (S (S (S n))) = 2 * a + 3 * b
```

Finally we can get both witnesses from our induction hypothesis, and construct the new decomposition

```
two_and_three_strong < destruct H2.
1 subgoal

  m : nat
  IHm : forall n : nat,
        n <= m -> 2 <= n -> exists a b : nat, n = 2 * a + 3 * b
  n : nat
  H0 : S (S (S (S n))) <= S m
  H1 : 2 <= S (S (S (S n)))
  x, x0 : nat
  H2 : S (S n) = 2 * x + 3 * x0
  ============================
  exists a b : nat, S (S (S (S n))) = 2 * a + 3 * b

two_and_three_strong < exists (S x). exists x0.
1 subgoal

  m : nat
  IHm : forall n : nat,
        n <= m -> 2 <= n -> exists a b : nat, n = 2 * a + 3 * b
  n : nat
  H0 : S (S (S (S n))) <= S m
  H1 : 2 <= S (S (S (S n)))
  x, x0 : nat
  H2 : S (S n) = 2 * x + 3 * x0
  ============================
  exists b : nat, S (S (S (S n))) = 2 * S x + 3 * b

1 subgoal

  m : nat
  IHm : forall n : nat,
        n <= m -> 2 <= n -> exists a b : nat, n = 2 * a + 3 * b
  n : nat
  H0 : S (S (S (S n))) <= S m
  H1 : 2 <= S (S (S (S n)))
  x, x0 : nat
  H2 : S (S n) = 2 * x + 3 * x0
  ============================
  S (S (S (S n))) = 2 * S x + 3 * x0

two_and_three_strong < rewrite H2.
1 subgoal

  m : nat
  IHm : forall n : nat,
        n <= m -> 2 <= n -> exists a b : nat, n = 2 * a + 3 * b
  n : nat
  H0 : S (S (S (S n))) <= S m
  H1 : 2 <= S (S (S (S n)))
  x, x0 : nat
  H2 : S (S n) = 2 * x + 3 * x0
  ============================
  S (S (2 * x + 3 * x0)) = 2 * S x + 3 * x0

two_and_three_strong < ring.
No more subgoals.

two_and_three_strong < Qed.
(induction m; intros **).
 (inversion H0).
 (rewrite H2 in H1).
 (inversion H1).

 (destruct n).
  (inversion H1).

  (destruct n).
   (inversion H1).
   (inversion H3).

   (destruct n).
    exists 1.
    exists 0.
    reflexivity.

    (destruct n).
     exists 0.
     exists 1.
     reflexivity.

     (pose (IHn := IHm (S (S n)))).
     (destruct IHn).
      (apply le_S_n in H0).
      (apply (le_trans (S (S n)) (S (S (S n))) m)).
       (repeat constructor).

       exact H0.
      (repeat apply le_n_S).
      (apply Nat.le_0_l).

      (destruct H2).
      exists (S x).
      exists x0.
      (rewrite H2).
      ring.

Qed.
two_and_three_strong is defined
```

Now we can prove our original result by just applying are lemma with $m = n$,

```
Coq < Theorem two_and_three : forall n : nat, 2 <= n -> exists (a b : nat), n = 2 * a + 3 * b.
1 subgoal

  ============================
  forall n : nat, 2 <= n -> exists a b : nat, n = 2 * a + 3 * b

two_and_three < intros n H.
1 subgoal

  n : nat
  H : 2 <= n
  ============================
  exists a b : nat, n = 2 * a + 3 * b

two_and_three < apply (two_and_three_strong n).
2 subgoals

  n : nat
  H : 2 <= n
  ============================
  n <= n

subgoal 2 is:
 2 <= n

two_and_three < apply le_n.
1 subgoal

  n : nat
  H : 2 <= n
  ============================
  2 <= n

two_and_three < exact H.
No more subgoals.

two_and_three < Qed.
(intros n H).
(apply (two_and_three_strong n)).
 (apply le_n).

 exact H.

Qed.
two_and_three is defined
```

We can assemble this kind of argument into a new induction principle called *strong induction*, but first we need a few definitions. A natural number k is said to be *accessible* with respect to the $<$ relation if every natural number less than k is accessible.

```
  Inductive acc : nat -> Prop := acc_k : forall k, (forall y, y < k -> acc y) -> acc k.
```

In fact, every natural number is accessible with respect to the < relation, meaning that < is a well founded relation. We can prove this by induction. The base case can be proved by inversion.

```
Coq < Theorem lt_wf : forall n, acc n.
1 subgoal

  ============================
  forall n : nat, acc n

lt_wf < induction n.
2 subgoals

  ============================
  acc 0

subgoal 2 is:
 acc (S n)

lt_wf < apply acc_k.
2 subgoals

  ============================
  forall y : nat, y < 0 -> acc y

subgoal 2 is:
 acc (S n)

lt_wf < intros y H.
2 subgoals

  y : nat
  H : y < 0
  ============================
  acc y

subgoal 2 is:
 acc (S n)

lt_wf < inversion H.
1 subgoal

  n : nat
  IHn : acc n
  ============================
  acc (S n)
```

In order to address the remaining goal, we need to induct on the `acc` relation itself, which we can do with the induction tactic. We get rid of the second hypothesis, which will not help us.

```
lt_wf < induction IHn.
1 subgoal

  k : nat
  H : forall y : nat, y < k -> acc y
  H0 : forall y : nat, y < k -> acc (S y)
  ============================
  acc (S k)

lt_wf < clear H0.
1 subgoal

  k : nat
  H : forall y : nat, y < k -> acc y
  ============================
  acc (S k)
```

Now we repeatedly apply the `acc_k` constructor, to generate a tower of relations that we can prove.

```
lt_wf < apply acc_k.
1 subgoal

  k : nat
  H : forall y : nat, y < k -> acc y
  ============================
  forall y : nat, y < S k -> acc y

lt_wf < intros y1 H1.
1 subgoal

  k : nat
  H : forall y : nat, y < k -> acc y
  y1 : nat
  H1 : y1 < S k
  ============================
  acc y1

lt_wf < apply acc_k.
1 subgoal

  k : nat
  H : forall y : nat, y < k -> acc y
  y1 : nat
  H1 : y1 < S k
  ============================
  forall y : nat, y < y1 -> acc y

lt_wf < intros y2 H2.
1 subgoal

  k : nat
  H : forall y : nat, y < k -> acc y
  y1 : nat
  H1 : y1 < S k
  y2 : nat
  H2 : y2 < y1
  ============================
  acc y2

lt_wf < apply H.
1 subgoal

  k : nat
  H : forall y : nat, y < k -> acc y
  y1 : nat
  H1 : y1 < S k
  y2 : nat
  H2 : y2 < y1
  ============================
  y2 < k
```

Now we apply the transitivity of the $<$ relation, which allows us to eventually prove the theorem.

```
lt_wf < apply le_trans with (m:=y1).
2 subgoals

  k : nat
  H : forall y : nat, y < k -> acc y
  y1 : nat
  H1 : y1 < S k
  y2 : nat
  H2 : y2 < y1
  ============================
  S y2 <= y1

subgoal 2 is:
 y1 <= k

lt_wf < exact H2.
1 subgoal

  k : nat
  H : forall y : nat, y < k -> acc y
  y1 : nat
  H1 : y1 < S k
  y2 : nat
  H2 : y2 < y1
  ============================
  y1 <= k
```

Now we need to transform this expression into the form of $H1$, so we look up the theorem,

```
lt_wf < Search (_ < S _ -> _ <= _).
lt_n_Sm_le: forall n m : nat, n < S m -> n <= m
```

and we can finish the proof.

```
lt_wf < apply lt_n_Sm_le.
1 subgoal

  k : nat
  H : forall y : nat, y < k -> acc y
  y1 : nat
  H1 : y1 < S k
  y2 : nat
  H2 : y2 < y1
  ============================
  y1 < S k

lt_wf < exact H1.
No more subgoals.

lt_wf < Qed.
(induction n).
 (apply acc_k).
 (intros y H).
 (inversion H).

 (induction IHn).
 clear H0.
 (apply acc_k).
 (intros y1 H1).
 (apply acc_k).
 (intros y2 H2).
 (apply H).
 (apply le_trans with (m := y1)).
  exact H2.

  (simpl).
  (apply lt_n_Sm_le).
  exact H1.

Qed.
lt_wf is defined
```

We can use the accessibility relation to define the strong induction principle for the natural numbers. The accessibility relation tells us which prior hypotheses we are allowed to use for the current step.

```
Definition acc_then_Px :
  forall {P : nat -> Prop} (n : nat),
    (forall x, (forall y, y < x -> P y) -> P x)
    -> acc n
    -> P n.
Proof.
  refine (fun P n f acc => acc_ind _ _ _ _).
  - intros k _ fp.
    apply f.
    exact fp.
  - exact acc.
Defined.

Definition strong_nat_ind :
  forall P : nat -> Prop,
    (forall x, (forall y, y < x -> P y) -> P x)
    -> forall x, P x.
Proof.
  refine (fun P f x => _).
  refine (acc_then_Px _ f _).
  apply lt_wf.
Defined.
```

Now we can prove our theorem using the strong induction tactic. We will change the statement to make this slightly easier by adding two instead of conditioning with the inequality.

```
Coq < Theorem two_and_three : forall n : nat, exists (a b : nat), n + 2 = 2 * a + 3 * b.
1 subgoal

  ============================
  forall n : nat, exists a b : nat, n + 2 = 2 * a + 3 * b

two_and_three < induction n using strong_nat_ind.
1 subgoal

  n : nat
  H : forall y : nat, y < n -> exists a b : nat, y + 2 = 2 * a + 3 * b
  ============================
  exists a b : nat, n + 2 = 2 * a + 3 * b
```

Now we will use `destruct` twice to pull out the base cases, just as we did before.

```
two_and_three < destruct n.
2 subgoals

  H : forall y : nat, y < 0 -> exists a b : nat, y + 2 = 2 * a + 3 * b
  ============================
  exists a b : nat, 0 + 2 = 2 * a + 3 * b

subgoal 2 is:
 exists a b : nat, S n + 2 = 2 * a + 3 * b

two_and_three < exists 1.
2 subgoals

  H : forall y : nat, y < 0 -> exists a b : nat, y + 2 = 2 * a + 3 * b
  ============================
  exists b : nat, 0 + 2 = 2 * 1 + 3 * b

subgoal 2 is:
 exists a b : nat, S n + 2 = 2 * a + 3 * b

two_and_three < exists 0.
2 subgoals

  H : forall y : nat, y < 0 -> exists a b : nat, y + 2 = 2 * a + 3 * b
  ============================
  0 + 2 = 2 * 1 + 3 * 0

subgoal 2 is:
 exists a b : nat, S n + 2 = 2 * a + 3 * b

two_and_three < simpl.
2 subgoals

  H : forall y : nat, y < 0 -> exists a b : nat, y + 2 = 2 * a + 3 * b
  ============================
  2 = 2

subgoal 2 is:
 exists a b : nat, S n + 2 = 2 * a + 3 * b

two_and_three < reflexivity.
1 subgoal

  n : nat
  H : forall y : nat, y < S n -> exists a b : nat, y + 2 = 2 * a + 3 * b
  ============================
  exists a b : nat, S n + 2 = 2 * a + 3 * b

two_and_three < destruct n.
2 subgoals

  H : forall y : nat, y < 1 -> exists a b : nat, y + 2 = 2 * a + 3 * b
  ============================
  exists a b : nat, 1 + 2 = 2 * a + 3 * b

subgoal 2 is:
 exists a b : nat, S (S n) + 2 = 2 * a + 3 * b

two_and_three < exists 0.
2 subgoals

  H : forall y : nat, y < 1 -> exists a b : nat, y + 2 = 2 * a + 3 * b
  ============================
  exists b : nat, 1 + 2 = 2 * 0 + 3 * b

subgoal 2 is:
 exists a b : nat, S (S n) + 2 = 2 * a + 3 * b

two_and_three < exists 1.
2 subgoals

  H : forall y : nat, y < 1 -> exists a b : nat, y + 2 = 2 * a + 3 * b
  ============================
  1 + 2 = 2 * 0 + 3 * 1

subgoal 2 is:
 exists a b : nat, S (S n) + 2 = 2 * a + 3 * b

two_and_three < simpl.
2 subgoals

  H : forall y : nat, y < 1 -> exists a b : nat, y + 2 = 2 * a + 3 * b
  ============================
  3 = 3

subgoal 2 is:
 exists a b : nat, S (S n) + 2 = 2 * a + 3 * b

two_and_three < reflexivity.
1 subgoal

  n : nat
  H : forall y : nat, y < S (S n) -> exists a b : nat, y + 2 = 2 * a + 3 * b
  ============================
  exists a b : nat, S (S n) + 2 = 2 * a + 3 * b
```

Now we use our induction hypothesis, evaluated at n, where the bound is obviously true.

```
two_and_three < pose (h := H n).
1 subgoal

  n : nat
  H : forall y : nat, y < S (S n) -> exists a b : nat, y + 2 = 2 * a + 3 * b
  h := H n : n < S (S n) -> exists a b : nat, n + 2 = 2 * a + 3 * b
  ============================
  exists a b : nat, S (S n) + 2 = 2 * a + 3 * b

two_and_three < assert (Hlt : n < S (S n)).
2 subgoals

  n : nat
  H : forall y : nat, y < S (S n) -> exists a b : nat, y + 2 = 2 * a + 3 * b
  h := H n : n < S (S n) -> exists a b : nat, n + 2 = 2 * a + 3 * b
  ============================
  n < S (S n)

subgoal 2 is:
 exists a b : nat, S (S n) + 2 = 2 * a + 3 * b

two_and_three < apply le_S.
2 subgoals

  n : nat
  H : forall y : nat, y < S (S n) -> exists a b : nat, y + 2 = 2 * a + 3 * b
  h := H n : n < S (S n) -> exists a b : nat, n + 2 = 2 * a + 3 * b
  ============================
  S n <= S n

subgoal 2 is:
 exists a b : nat, S (S n) + 2 = 2 * a + 3 * b

two_and_three < apply le_n.
1 subgoal

  n : nat
  H : forall y : nat, y < S (S n) -> exists a b : nat, y + 2 = 2 * a + 3 * b
  h := H n : n < S (S n) -> exists a b : nat, n + 2 = 2 * a + 3 * b
  Hlt : n < S (S n)
  ============================
  exists a b : nat, S (S n) + 2 = 2 * a + 3 * b

two_and_three < apply h in Hlt.
1 subgoal

  n : nat
  H : forall y : nat, y < S (S n) -> exists a b : nat, y + 2 = 2 * a + 3 * b
  h := H n : n < S (S n) -> exists a b : nat, n + 2 = 2 * a + 3 * b
  Hlt : exists a b : nat, n + 2 = 2 * a + 3 * b
  ============================
  exists a b : nat, S (S n) + 2 = 2 * a + 3 * b
```

Now we can extract the witnesses, and rewrite our goal with the resulting equality.

```
two_and_three < destruct Hlt as [a [b Heq]].
1 subgoal

  n : nat
  H : forall y : nat, y < S (S n) -> exists a b : nat, y + 2 = 2 * a + 3 * b
  h := H n : n < S (S n) -> exists a b : nat, n + 2 = 2 * a + 3 * b
  a, b : nat
  Heq : n + 2 = 2 * a + 3 * b
  ============================
  exists a0 b0 : nat, S (S n) + 2 = 2 * a0 + 3 * b0

two_and_three < exists (a + 1).
1 subgoal

  n : nat
  H : forall y : nat, y < S (S n) -> exists a b : nat, y + 2 = 2 * a + 3 * b
  h := H n : n < S (S n) -> exists a b : nat, n + 2 = 2 * a + 3 * b
  a, b : nat
  Heq : n + 2 = 2 * a + 3 * b
  ============================
  exists b0 : nat, S (S n) + 2 = 2 * (a + 1) + 3 * b0

two_and_three < exists b.
1 subgoal

  n : nat
  H : forall y : nat, y < S (S n) -> exists a b : nat, y + 2 = 2 * a + 3 * b
  h := H n : n < S (S n) -> exists a b : nat, n + 2 = 2 * a + 3 * b
  a, b : nat
  Heq : n + 2 = 2 * a + 3 * b
  ============================
  S (S n) + 2 = 2 * (a + 1) + 3 * b

two_and_three < simpl.
1 subgoal

  n : nat
  H : forall y : nat, y < S (S n) -> exists a b : nat, y + 2 = 2 * a + 3 * b
  h := H n : n < S (S n) -> exists a b : nat, n + 2 = 2 * a + 3 * b
  a, b : nat
  Heq : n + 2 = 2 * a + 3 * b
  ============================
  S (S (n + 2)) = a + 1 + (a + 1 + 0) + (b + (b + (b + 0)))

two_and_three < rewrite Heq.
1 subgoal

  n : nat
  H : forall y : nat, y < S (S n) -> exists a b : nat, y + 2 = 2 * a + 3 * b
  h := H n : n < S (S n) -> exists a b : nat, n + 2 = 2 * a + 3 * b
  a, b : nat
  Heq : n + 2 = 2 * a + 3 * b
  ============================
  S (S (2 * a + 3 * b)) = a + 1 + (a + 1 + 0) + (b + (b + (b + 0)))

two_and_three < ring.
No more subgoals.

two_and_three < Qed.
(induction n using strong_nat_ind).
(destruct n).
 exists 1.
 exists 0.
 (simpl).
 reflexivity.

 (destruct n).
  exists 0.
  exists 1.
  (simpl).
  reflexivity.
  (pose (h := H n)).
  (assert (Hlt : n < S (S n))).
   (apply le_S).
   (apply le_n).

   (apply h in Hlt).
   (destruct Hlt as [a [b Heq]]).
   exists (a + 1).
   exists b.
   (simpl).
   (rewrite Heq).
   ring.

Qed.
two_and_three is defined
```

4.4 Sequences and Series

A *series* is a sum of several terms. For example, the sum of the first n natural numbers

$$\sum_{i=1}^{n} i = 1 + 2 + \cdots + n. \tag{4.29}$$

We are often looking for a close-form expression for the sum $S(n)$, in terms of its upper limit. There are many strategies for guessing forms for answer, see (Graham, Knuth, and Patashnik 1989), which we can then justify using induction.

In order to work with inductive types, Coq provides a structure for computing recursive functions. For example, we can write

$$\sum_{i=0}^{n} 1 \tag{4.30}$$

as sum_1

```
Fixpoint sum_1 (n : nat) := match n with
    0 => 1
| S p => S (sum_1 p)
end.
```

using the *Fixpoint* operator. Coq makes sure the recursive definitions are sensible by analyzing the structure of the result. In our example above, we can have p on the right hand side, but not Sp, so that we are guaranteed to reach the fixed point. There is no limitation to a single argument, as long as we indicate which argument is structural. For example, we can define addition recursively,

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

where we indicate that n is the induction variable using the `struct` keyword.

As a simple example, lets prove that if we sum one $n+1$ times, we get $n+1$. We start by using the definition of the sum from above, `sum_1`. We can state the theorem, and then initiate induction on n, sum_1_analytic

```
Coq < Theorem sum_1_analytic : forall n : nat, sum_1 n = S n.
1 subgoal

  A : Set
  ============================
  forall n : nat, sum_1 n = S n

sum_1_analytic < induction n.
2 subgoals

  A : Set
  ============================
  sum_1 0 = 1

subgoal 2 is:
 sum_1 (S n) = S (S n)
```

The first goal, our base case, can be solved simply by replacing the function `sum_1` by its value using the `simpl` tactic,

```
sum_1_analytic < simpl.
2 subgoals

  A : Set
  ============================
  1 = 1

subgoal 2 is:
 sum_1 (S n) = S (S n)

sum_1_analytic < reflexivity.
1 subgoal

  A : Set
  n : nat
  IHn : sum_1 n = S n
  ============================
  sum_1 (S n) = S (S n)
```

Now we need to prove the induction step. Notice that the proof assistant has already introduced the induction hypothesis `IHn` for us. We again use `simpl` to evaluate the inductive function,

```
sum_1_analytic < simpl.
1 subgoal

  A : Set
  n : nat
  IHn : sum_1 n = S n
  ============================
  S (sum_1 n) = S (S n)
```

and then use the induction hypothesis to rewrite the goal and we are finished.

```
sum_1_analytic < rewrite IHn.
1 subgoal

  A : Set
  n : nat
  IHn : sum_1 n = S n
  ============================
  S (S n) = S (S n)

sum_1_analytic < reflexivity.
No more subgoals.

sum_1_analytic < Qed.
```

```
(induction n).
 (simpl).
 reflexivity.

 (simpl).
 (rewrite IHn).
 reflexivity.

Qed.
sum_1_analytic is defined
```

We can also write this proof in the by-hand style used in homework,

$$\begin{aligned}
n &: \text{nat} \\
IHn &: \text{sum}_1(n) = S(n) \\
\hline
G0 &: \forall n : \text{nat}, \text{sum}_1(n) = S(n) \\
G1 &: \text{sum}_1(0) = S(0) \wedge \\
&\quad (\forall n : \text{nat}, \text{sum}_1(n) = S(n) \implies \text{sum}_1(S(n)) = S(S(n))) \\
G2a &: \text{sum}_1(0) = S(0) \\
G3a &: 1 = 1 \\
G2b &: \forall n : \text{nat}, \text{sum}_1(n) = S(n) \implies \text{sum}_1(S(n)) = S(S(n)) \\
G3b &: \text{sum}_1(S(n)) = S(S(n)) \\
G4b &: S(\text{sum}_1(n)) = S(S(n)) \\
G5b &: S(S(n)) = S(S(n))
\end{aligned}$$

Step	Tactic
1	apply nat_ind
2	split
3a	simpl
4a	reflexivity
3b	intros n IHn
4b	simpl
5b	rewrite IHn
6b	reflexivity

4.4.1 Arithmetic Series

The canonical arithmetic series, the sum of the first n natural numbers, was solved by Gauss as a young child. We will guess a closed form solution

$$\sum_{i=0}^{n-1} i = \frac{n(n-1)}{2}, \tag{4.31}$$

which gives us our induction hypothesis $P(n)$. The base case $P(0)$

$$P(0) := \sum_{i=0}^{-1} i = \frac{0(0-1)}{2}, \tag{4.32}$$

$$0 = 0, \tag{4.33}$$

$$\text{T}. \tag{4.34}$$

is true. For the induction step, we assume $P(n)$, and prove the implication $P(n) \implies P(n+1)$,

$$\begin{aligned}
P(n+1) := \sum_{i=0}^{n} i &= \frac{n(n+1)}{2} && (4.35)\\
\sum_{i=0}^{n-1} i + n &= \frac{n(n+1)}{2} && (4.36)\\
\frac{n(n-1)}{2} + n &= \frac{n(n+1)}{2} && (4.37)\\
\frac{n(n-1)+2n}{2} &= \frac{n(n+1)}{2} && (4.38)\\
\frac{n(n+1)}{2} &= \frac{n(n+1)}{2} && (4.39)\\
&\text{T.} && (4.40)
\end{aligned}$$

Thus the implication is true, and the induction is proved. We can write this proof in our informal style

$$\begin{aligned}
n :&\text{nat}\\
IHn :&2 * \text{sum}_n(n) + n = n * n
\end{aligned}$$

$$\begin{aligned}
G0 :&\forall n : \text{nat}, 2 * \text{sum}_n(n) + n = n * n\\
G1 :&2 * \text{sum}_n(0) + 0 = 0 * 0 \wedge\\
&(\forall n : \text{nat}, 2 * \text{sum}_n(n) + n = n * n \implies\\
&2 * \text{sum}_n(S(n)) + S(n) = S(n) * S(n))\\
G2a :&2 * \text{sum}_n(0) + 0 = 0 * 0\\
G3a :&0 = 0\\
G2b :&\forall n : \text{nat}, 2 * \text{sum}_n(n) + n = n * n \implies\\
&2 * \text{sum}_n(S(n)) + S(n) = S(n) * S(n)\\
G3b :&2 * \text{sum}_n(S(n)) + S(n) = S(n) * S(n)\\
G4b :&2 * \text{sum}_n(S(n)) + S(n) = n * n + 2 * n + 1\\
G5b :&2 * \text{sum}_n(S(n)) + S(n) = 2 * \text{sum}_n(n) + n + 2 * n + 1\\
G6b :&2 * \text{sum}_n(n) + 2 * n + n + 1 = 2 * \text{sum}_n(n) + n + 2 * n + 1
\end{aligned}$$

Step	Tactic
1	apply nat_ind
2	split
3a	simpl
4a	reflexivity
3b	intros n IHn
4b	algebra
5b	rewrite → IHn
6b	unfold sum_n
7b	reflexivity

We can verify our proof of a closed form for the sum of the n first natural numbers using Coq. First we define a helper function which gives us the sum, sum_n

```
Fixpoint sum_n n :=
  match n with
     0 => 0
  | S p => p + sum_n p
```

```
end.
```

Notice that a natural number n can be either 0 or the successor of another number p, and we treat those two cases. If there are more constructors for an inductive type, we would have more cases in our Fixpoint statement. We can test this definition using the *Compute* operation in the proof assistant

```
Coq < Compute (sum_n 1).
     = 0
     : nat

Coq < Compute (sum_n 2).
     = 1
     : nat

Coq < Compute (sum_n 5).
     = 10
     : nat
```

Remember that the `Fixpoint` operator allows us to define recurisve expressions as long as they are well-founded, meaning that they do not infinitely recurse. This is enforced by making sure that it is *structural recursion*, meaning that the right hand side of the match only uses part of the left hand side. Here we again match `S p`, but only use `p`.

Next, we remove the division and subtraction from the lemma we want to prove. We do this so that we can employ the *ring* tactic to solve the algebraic equations. The ring operations for the natural numbers are addition and multiplication, so the tactic cannot handle subtraciton and division. If we are careful in the definition, we can get `ring` to do most of the work for us.

First we use the `induction` tactic, sum_n_p

```
Coq < Lemma sum_n_p : forall n, 2 * sum_n n + n = n * n.
1 subgoal

  ============================
  forall n : nat, 2 * sum_n n + n = n * n

sum_n_p < induction n.
2 subgoals

  ============================
  2 * sum_n 0 + 0 = 0 * 0

subgoal 2 is:
 2 * sum_n (S n) + S n = S n * S n
```

that generates two goals. The first goal is our base case that has n replaced by 0. The second one has n replaced by Sn, our induction step, with a hidden hypothesis corresponding the statement being already true for n, the induction hypothesis. To handle the first goal, we can use the `simpl` tactic to eliminate the use of `sum_n 0` and then `reflexivity` since the expressions are equal

```
sum_n_p < simpl.
2 subgoals

  ============================
  0 = 0

subgoal 2 is:
 2 * sum_n (S n) + S n = S n * S n
```

```
sum_n_p < reflexivity.
1 subgoal

  n : nat
  IHn : 2 * sum_n n + n = n * n
  ============================
  2 * sum_n (S n) + S n = S n * S n
```

although we could have just used the reflexivity tactic. We can now see the induction hypothesis.

The **main idea** in an induction proof to verify the closed form for a series is to use the induction hypothesis either to eliminate the defintion of our sum, or to replace part of the closed form with our sum definition. Either strategy will allow the ring tactic to verify the resulting equality. In this proof, we will replace part of the closed form solution by our sum definition. Thus we need to pull out a term that looks like the closed form for our induction step. In this proof that means we need to pull out the n^2 term from our expression $(n+1)^2$. We can do this by hand and them have the proof assistant check our work.

Note that we do not need to use `assert` when proving these identities. We can use low level theorems about the associativity, commutativity, and distributivity of addition and multiplication. You can find these by searching in Coq. Below I show some output edited for length.

```
I find them by searching For instance, an edited list
Coq < Search (_ * _ = _ * _).
Nat.mul_comm: forall n m : nat, n * m = m * n
Nat.mul_assoc: forall n m p : nat, n * (m * p) = n * m * p
Nat.mul_cancel_l: forall n m p : nat, p <> 0 -> p * n = p * m <-> n = m
Nat.mul_cancel_r: forall n m p : nat, p <> 0 -> n * p = m * p <-> n = m

Coq < Search (_ + _ = _ + _).
Nat.add_comm: forall n m : nat, n + m = m + n
Nat.add_assoc: forall n m p : nat, n + (m + p) = n + m + p
Nat.add_cancel_l: forall n m p : nat, p + n = p + m <-> n = m
Nat.add_cancel_r: forall n m p : nat, n + p = m + p <-> n = m

Coq < Search (_ * (_ + _) = _ * _ + _ * _).
Nat.mul_add_distr_l: forall n m p : nat, n * (m + p) = n * m + n * p
```

We will assume the identity we derived by hand, which just treats it like a separate subproof, using the `assert` tactic. Then we can use the `ring` tactic to prove it.

```
sum_n_p < assert (SnSn : S n * S n = n * n + 2 * n + 1).
2 subgoals

  n : nat
  IHn : 2 * sum_n n + n = n * n
  ============================
  S n * S n = n * n + 2 * n + 1

subgoal 2 is:
 2 * sum_n (S n) + S n = S n * S n

sum_n_p < ring.
1 subgoal

  n : nat
  IHn : 2 * sum_n n + n = n * n
  SnSn : S n * S n = n * n + 2 * n + 1
  ============================
  2 * sum_n (S n) + S n = S n * S n
```

Now we can use the `rewrite` tactic to replace `S n * S n` with our identity that exposes the `n * n` term. This then allows us to use our induction hypothesis,

```
sum_n_p < rewrite SnSn.
1 subgoal

  n : nat
  IHn : 2 * sum_n n + n = n * n
  SnSn : S n * S n = n * n + 2 * n + 1
  ============================
  2 * sum_n (S n) + S n = n * n + 2 * n + 1

sum_n_p < rewrite <- IHn.
1 subgoal

  n : nat
  IHn : 2 * sum_n n + n = n * n
  SnSn : S n * S n = n * n + 2 * n + 1
  ============================
  2 * sum_n (S n) + S n = 2 * sum_n n + n + 2 * n + 1
```

The next step is to make `sum_n (S n)` unwind the recursive definition one step, so that its value is expressed using `sum_n n` according to the definition, which also forces the symbolic computation of multiplications. Finally, we use the `ring` tactic to recognize that this equation is a consequence of associativity and commutativity of addition.

```
sum_n_p < simpl.
1 subgoal

  n : nat
  IHn : 2 * sum_n n + n = n * n
  SnSn : S n * S n = n * n + 2 * n + 1
  ============================
  n + sum_n n + (n + sum_n n + 0) + S n =
  sum_n n + (sum_n n + 0) + n + (n + (n + 0)) + 1

sum_n_p < ring.
No more subgoals.

sum_n_p < Qed.
(induction n).
 (simpl).
 reflexivity.

 (assert (SnSn : S n * S n = n * n + 2 * n + 1)).
  ring.

  (rewrite SnSn).
  (rewrite <- IHn).
  (simpl).
  ring.

Qed.
sum_n_p is defined
```

We can also prove something more interesting about the sum of the first n

natural numbers. If we look at the sums,

$$\begin{aligned}
&(n=0) \quad 0 = \sum_{i=0}^{-1} i = 0\\
&(n=1) \quad 0 = \sum_{i=0}^{0} i = 0\\
&(n=2) \quad 1 = \sum_{i=0}^{1} i = 0+1\\
&(n=3) \quad 3 = \sum_{i=0}^{2} i = 0+1+2\\
&(n=4) \quad 6 = \sum_{i=0}^{3} i = 0+1+2+3\\
&\qquad\qquad \vdots
\end{aligned}$$

we see that the sum is always greater than or equal to $n-1$. Thus our predicate for induction will be

$$P(n) := \sum_{i=0}^{n-1} i \geq n-1.$$

The base case can be easily verified

$$\begin{aligned}
P(0) &:= \sum_{i=0}^{-1} i \geq 0-1\\
&:= 0 \geq -1\\
&:= \mathrm{T}
\end{aligned}$$

For the induction step, we assume $P(n)$ and prove $P(n+1)$,

$$\begin{aligned}
P(n+1) &:= \sum_{i=0}^{n} i \geq n\\
&:= \sum_{i=0}^{n-1} i + n \geq n\\
&:= n \geq 1
\end{aligned}$$

where we used $P(n)$ to subtract from both sides. The problem is now clear. We need another base case in order to patch up our argument,

$$\begin{aligned} P(1) &:= \sum_{i=0}^{0} i \geq 1-1 \\ &:= 0 \geq 0 \\ &:= \mathrm{T} \end{aligned}$$

and our implication is true for any $n \geq 1$. We can put this into our informal proof tableau, shown below. We are allowed to conclude some things, like $1 \geq 0$ and $n \geq 0$ without proof.

$$\begin{aligned} n &: \text{nat} \\ IHn &: \text{sum}_n(n) + 1 \geq n \\ IHnb &: \text{sum}_n(Sn) + 1 \geq Sn \end{aligned}$$

$$\begin{aligned} G0 &: \forall n : \text{nat}, \text{sum}_n(n) + 1 \geq n \\ G1a &: \text{sum}_n(0) + 1 \geq 0 \\ G2a &: 1 \geq 0 \\ G1b &: \text{sum}_n(n) + 1 \geq n \implies \text{sum}_n(S(n)) + 1 \geq S(n) \\ G2b &: \text{sum}_n(S(n)) + 1 \geq S(n) \\ G3ba &: \text{sum}_n(1) + 1 \geq 1 \\ G4ba &: 0 + 1 \geq 1 \\ G3bb &: \text{sum}_n(S(S(n))) + 1 \geq S(S(n)) \\ G4bb &: \text{sum}_n(S(S(n))) \geq S(n) \\ G5bb &: \text{sum}_n(S(n)) + S(n) \geq S(n) \\ G6bb &: n \geq 0 \end{aligned}$$

Step	Tactic
1	induction n
2a	simpl
3a	ℕ property
2b	intros n IHn
3b	destruct n
4ba	simpl
5ba	<= property
4bb	cancel 1
5bb	simpl
6bb	use IHnb
7bb	ℕ property

We can also prove this by induction using the proof assistant. bigger

```
Coq < Theorem bigger : forall n : nat, n <= sum_n n + 1.
1 subgoal

  ============================
  forall n : nat, n <= sum_n n + 1

bigger < induction n.
2 subgoals

  ============================
  0 <= sum_n 0 + 1
```

For the base case, $n = 0$, we can simplify the result and use our standard theorems about the $\leq$ operator,

```
bigger < simpl.
2 subgoals

  ============================
  0 <= 1
```

```
subgoal 2 is:
 S n <= sum_n (S n) + 1

bigger < apply le_S.
2 subgoals

  ============================
  0 <= 0

subgoal 2 is:
 S n <= sum_n (S n) + 1

bigger < apply le_n.
1 subgoal

  n : nat
  IHn : n <= sum_n n + 1
  ============================
  S n <= sum_n (S n) + 1
```

At this point, we can get into trouble. If we use our induction hypothesis `IHn` directly, it will not be strong enough to prove our goal. We have

$$n+1 \leq \sum_{i=0}^{n} i+1 \tag{4.41}$$

$$n+1 \leq n+\sum_{i=0}^{n-1} i+1 \tag{4.42}$$

$$1 \leq n \tag{4.43}$$

where we used our induction hypothesis in the last step. Clearly, we cannot prove this because n could be zero. However, we can avoid this by proving another base case. We decompose n into the two possible cases for a natural number, namely zero or the successor of some number,

```
bigger < destruct n.
2 subgoals

  IHn : 0 <= sum_n 0 + 1
  ============================
  1 <= sum_n 1 + 1

subgoal 2 is:
 S (S n) <= sum_n (S (S n)) + 1
```

and then use `simpl` to replace an inductive expression with its definition,

```
bigger < simpl.
2 subgoals

  IHn : 0 <= sum_n 0 + 1
  ============================
  1 <= 1

subgoal 2 is:
 S (S n) <= sum_n (S (S n)) + 1

bigger < apply le_n.
1 subgoal

  n : nat
  IHn : S n <= sum_n (S n) + 1
  ============================
  S (S n) <= sum_n (S (S n)) + 1
```

Now we can apply a commonsense theorem about adding to each side of the inequality,

```
  Nat.le_le_add_le : forall n m p q : nat, n <= m -> p + m <= q + n -> p <= q
```

where the first goal is just one of our assumptions.

```
bigger < rewrite <- Nat.add_1_r.
1 subgoal

  n : nat
  IHn : S n <= sum_n (S n) + 1
  ============================
  S n + 1 <= sum_n (S n + 1) + 1

bigger < apply Nat.le_le_add_le with (n := S n) (m := sum_n (S n) + 1).
2 subgoals

  n : nat
  IHn : S n <= sum_n (S n) + 1
  ============================
  S n <= sum_n (S n) + 1

subgoal 2 is:
 S n + 1 + (sum_n (S n) + 1) <= sum_n (S n + 1) + 1 + S n

bigger < assumption.
1 subgoal

  n : nat
  IHn : S n <= sum_n (S n) + 1
  ============================
  S n + 1 + (sum_n (S n) + 1) <= sum_n (S n + 1) + 1 + S n
```

Now we want to simplify the sums, so that we can cancel them out on each side.

```
bigger < rewrite Nat.add_1_r at 2.
1 subgoal

  n : nat
  IHn : S n <= sum_n (S n) + 1
  ============================
  S n + 1 + (sum_n (S n) + 1) <= sum_n (S (S n)) + 1 + S n

bigger < simpl.
1 subgoal

  n : nat
  IHn : S n <= sum_n (S n) + 1
  ============================
  S (n + 1 + (n + sum_n n + 1)) <= S (n + (n + sum_n n) + 1 + S n)
```

Now we get rid of all the uses of successor, and cancel the first term, which is a one,

```
bigger < rewrite <- Nat.add_1_r.
1 subgoal

  n : nat
  IHn : S n <= sum_n (S n) + 1
  ============================
  n + 1 + (n + sum_n n + 1) + 1 <= S (n + (n + sum_n n) + 1 + S n)
```

```
bigger < rewrite <- Nat.add_1_r with (n:=n).
1 subgoal

  n : nat
  IHn : S n <= sum_n (S n) + 1
  ============================
  n + 1 + (n + sum_n n + 1) + 1 <= S (n + (n + sum_n n) + 1 + (n + 1))

bigger < rewrite <- Nat.add_1_r with (n:=n + (n + sum_n n) + 1 + (n + 1)).
1 subgoal

  n : nat
  IHn : S n <= sum_n (S n) + 1
  ============================
  n + 1 + (n + sum_n n + 1) + 1 <= n + (n + sum_n n) + 1 + (n + 1) + 1

bigger < apply plus_le_compat_r.
1 subgoal

  n : nat
  IHn : S n <= sum_n (S n) + 1
  ============================
  n + 1 + (n + sum_n n + 1) <= n + (n + sum_n n) + 1 + (n + 1)
```

We use the same strategy, and some algebraic manipulation, to cancel everything from the left side, leaving only n on the right,

```
bigger < rewrite <- plus_assoc with (n:=n) (m:=n + sum_n n) (p:=1).
1 subgoal

  n : nat
  IHn : S n <= sum_n (S n) + 1
  ============================
  n + 1 + (n + sum_n n + 1) <= n + (n + sum_n n + 1) + (n + 1)

bigger < rewrite <- plus_assoc with (n:=n) (m:=n + sum_n n + 1) (p:=n+1).
1 subgoal

  n : nat
  IHn : S n <= sum_n (S n) + 1
  ============================
  n + 1 + (n + sum_n n + 1) <= n + (n + sum_n n + 1 + (n + 1))

bigger < rewrite plus_comm with (n:=n + sum_n n + 1) (m:=n+1).
1 subgoal

  n : nat
  IHn : S n <= sum_n (S n) + 1
  ============================
  n + 1 + (n + sum_n n + 1) <= n + (n + 1 + (n + sum_n n + 1))

bigger < rewrite plus_assoc with (n:=n) (m:=n+1).
1 subgoal

  n : nat
  IHn : S n <= sum_n (S n) + 1
  ============================
  n + 1 + (n + sum_n n + 1) <= n + (n + 1) + (n + sum_n n + 1)

bigger < apply plus_le_compat_r.
1 subgoal

  n : nat
  IHn : S n <= sum_n (S n) + 1
  ============================
  n + 1 <= n + (n + 1)
```

```
bigger < rewrite <- plus_O_n at 1.
1 subgoal

  n : nat
  IHn : S n <= sum_n (S n) + 1
  ============================
  0 + (n + 1) <= n + (n + 1)

bigger < apply plus_le_compat_r.
1 subgoal

  n : nat
  IHn : S n <= sum_n (S n) + 1
  ============================
  0 <= n
```

Finally, we use the fundamental fact from Peano arithmetic that all numbers are greater than or equal to zero.

```
bigger < apply Peano.le_0_n.
No more subgoals.

bigger < Qed.
(induction n).
 (simpl).
 (apply le_S).
 (apply le_n).

 (destruct n).
  (simpl).
  (apply le_n).

  (rewrite <- Nat.add_1_r).
  (apply Nat.le_le_add_le with (n := S n) (m := sum_n (S n) + 1)).
   assumption.

   (rewrite Nat.add_1_r at 2).
   (simpl).
   (rewrite <- Nat.add_1_r).
   (rewrite <- Nat.add_1_r with (n := n)).
   (rewrite <- Nat.add_1_r with (n := (n + (n + sum_n n) + 1 + (n + 1)))).
   (apply plus_le_compat_r).
   (rewrite <- plus_assoc with (n := n) (m := (n + sum_n n)) (p := 1)).
   (rewrite <- plus_assoc with
      (n := n) (m := (n + sum_n n + 1)) (p := (n + 1))).
   (rewrite plus_comm with (n := (n + sum_n n + 1)) (m := (n + 1))).
   (rewrite plus_assoc with (n := n) (m := (n + 1))).
   (apply plus_le_compat_r).
   (rewrite <- plus_O_n at 1).
   (apply plus_le_compat_r).
   (apply Peano.le_0_n).

Qed.
bigger is defined
```

4.4.2 Geometric Series

Another common example of induction is the geometric series,

$$\sum_{i=0}^{n} x^i = \frac{x^{n+1}-1}{x-1}, \tag{4.44}$$

for $x \in \mathbb{R}$. The induction hypothesis $P(n)$ is given by

$$P(n) := \sum_{i=0}^{n} x^i = \frac{x^{n+1}-1}{x-1} \tag{4.45}$$

and thus the base case $P(0)$,

$$\begin{align}
P(0) := \sum_{i=0}^{0} x^i &= \frac{x-1}{x-1} \tag{4.46}\\
x^0 &= 1 \tag{4.47}\\
&\text{T} \tag{4.48}
\end{align}$$

is true. Assuming $P(n)$ is true, the induction step is

$$\begin{align}
P(n+1) := \sum_{i=0}^{n+1} x^i &= \frac{x^{n+2}-1}{x-1} \tag{4.49}\\
\sum_{i=0}^{n} x^i + x^{n+1} &= \frac{x^{n+2}-1}{x-1} \tag{4.50}\\
\frac{x^{n+1}-1}{x-1} + \frac{x^{n+2}-x^{n+1}}{x-1} &= \frac{x^{n+2}-1}{x-1} \tag{4.51}\\
\frac{x^{n+2}-1}{x-1} &= \frac{x^{n+2}-1}{x-1} \tag{4.52}\\
&\text{T.} \tag{4.53}
\end{align}$$

Thus the induction step is true and the theorem is proved. We can put this in our informal tableau,

$$
\begin{aligned}
x &: \mathbb{R} \\
n &: \mathbb{N} \\
IHn &: \text{sum}_g(n,x) * x + 1 = x^{n+1} + \text{sum}_g(n,x)
\end{aligned}
$$

$$
\begin{aligned}
G0 &: \forall x : \mathbb{R}, \forall n : \mathbb{N}, \text{sum}_g(n,x) * x + 1 = x^{n+1} + \text{sum}_g(n,x) \\
G1 &: \forall n : \mathbb{N}, \text{sum}_g(n,x) * x + 1 = x^{n+1} + \text{sum}_g(n,x) \\
G2a &: \text{sum}_g(0,x) * x + 1 = x^{0+1} + \text{sum}_g(0,x) \\
G3a &: 1 * x + 1 = x^(0+1) + 1 \\
G4a &: x + 1 = x + 1 \\
G2b &: \text{sum}_g(S(n),x) * x + 1 = x^{S(n)+1} + \text{sum}_g(S(n),x) \\
G3b &: (x^{S(n)} + \text{sum}_g(n,x)) * x + 1 = x^{S(n)+1} + (x^{S(n)} + \text{sum}_g(n,x)) \\
G4b &: x^{S(n)+1} + \text{sum}_g(n,x) * x + 1 = x^{S(n)+1} + x^{S(n)} + \text{sum}_g(n,x) \\
G5b &: \text{sum}_g(n,x) * x + 1 = x^{S(n)} + \text{sum}_g(n,x)
\end{aligned}
$$

Step	Tactic
1	intro x
2	induction n
3a	simpl
4a	algebra
5a	reflexivity
3b	simpl
4b	algebra
5b	cancel x^{n+2}
6b	apply IHn

We can also use Coq to prove the closed form for the geometric sum. First we need some definitions to handle real numbers, and a recursive definition for our sum, sum_geom

```
Require Import Reals.
Require Import Rfunctions.
Open Scope R_scope.

Fixpoint sum_geom (n:nat) (x:R) : R :=
match n with
  0 => 1
| S p => x^n + sum_geom p x
end.
sum_geom is defined
sum_geom is recursively defined (decreasing on 1st argument)
```

We can check that this Now we state the lemma and prove the base case, sum_geom_p

```
Coq < Lemma sum_geom_p : forall n:nat, forall x:R, sum_geom n x * x + 1 = x^(n + 1) + sum_geom n x.
1 subgoal

  ============================
  forall (n : nat) (x : R), sum_geom n x * x + 1 = x ^ (n + 1) + sum_geom n x

sum_geom_p < induction n.
2 subgoals

  ============================
  forall x : R, sum_geom 0 x * x + 1 = x ^ (0 + 1) + sum_geom 0 x

subgoal 2 is:
 forall x : R, sum_geom (S n) x * x + 1 = x ^ (S n + 1) + sum_geom (S n) x

sum_geom_p < intros x.
2 subgoals
```

```
  x : R
  ============================
  sum_geom 0 x * x + 1 = x ^ (0 + 1) + sum_geom 0 x

subgoal 2 is:
 forall x : R, sum_geom (S n) x * x + 1 = x ^ (S n + 1) + sum_geom (S n) x

sum_geom_p < simpl.
2 subgoals

  x : R
  ============================
  1 * x + 1 = x * 1 + 1

subgoal 2 is:
 forall x : R, sum_geom (S n) x * x + 1 = x ^ (S n + 1) + sum_geom (S n) x

sum_geom_p < rewrite Rmult_1_l.
2 subgoals

  x : R
  ============================
  x + 1 = x * 1 + 1

subgoal 2 is:
 forall x : R, sum_geom (S n) x * x + 1 = x ^ (S n + 1) + sum_geom (S n) x

sum_geom_p < rewrite Rmult_1_r.
2 subgoals

  x : R
  ============================
  x + 1 = x + 1

subgoal 2 is:
 forall x : R, sum_geom (S n) x * x + 1 = x ^ (S n + 1) + sum_geom (S n) x

sum_geom_p < reflexivity.
1 subgoal

  n : nat
  IHn : forall x : R, sum_geom n x * x + 1 = x ^ (n + 1) + sum_geom n x
  ============================
  forall x : R, sum_geom (S n) x * x + 1 = x ^ (S n + 1) + sum_geom (S n) x
```

We used induction and then the introduction tactic for universal quantification, combined with some simplification of the real arithmetic. We could also have used the `ring` tactic instead of explicit simplification. Note that we avoid using subtraction because that makes the arithmetic simplification harder. Now we handle the induction step.

```
sum_geom_p < intros x.
1 subgoal

  n : nat
  IHn : forall x : R, sum_geom n x * x + 1 = x ^ (n + 1) + sum_geom n x
  x : R
  ============================
  sum_geom (S n) x * x + 1 = x ^ (S n + 1) + sum_geom (S n) x

sum_geom_p < simpl.
1 subgoal

  n : nat
  IHn : forall x : R, sum_geom n x * x + 1 = x ^ (n + 1) + sum_geom n x
  x : R
  ============================
  (x * x ^ n + sum_geom n x) * x + 1 =
  x * x ^ (n + 1) + (x * x ^ n + sum_geom n x)
```

Then we simplify the powers

```
sum_geom_p < rewrite Rmult_plus_distr_r.
1 subgoal

  n : nat
  IHn : forall x : R, sum_geom n x * x + 1 = x ^ (n + 1) + sum_geom n x
  x : R
  ============================
  x * x ^ n * x + sum_geom n x * x + 1 =
  x * x ^ (n + 1) + (x * x ^ n + sum_geom n x)

sum_geom_p < rewrite tech_pow_Rmult.
1 subgoal

  n : nat
  IHn : forall x : R, sum_geom n x * x + 1 = x ^ (n + 1) + sum_geom n x
  x : R
  ============================
  x ^ S n * x + sum_geom n x * x + 1 =
  x * x ^ (n + 1) + (x ^ S n + sum_geom n x)

sum_geom_p < rewrite tech_pow_Rmult.
1 subgoal

  n : nat
  IHn : forall x : R, sum_geom n x * x + 1 = x ^ (n + 1) + sum_geom n x
  x : R
  ============================
  x ^ S n * x + sum_geom n x * x + 1 =
  x ^ S (n + 1) + (x ^ S n + sum_geom n x)

sum_geom_p < rewrite Rmult_comm with (r1:=x ^ S n) (r2:=x).
1 subgoal

  n : nat
  IHn : forall x : R, sum_geom n x * x + 1 = x ^ (n + 1) + sum_geom n x
  x : R
  ============================
  x * x ^ S n + sum_geom n x * x + 1 =
  x ^ S (n + 1) + (x ^ S n + sum_geom n x)

sum_geom_p < rewrite tech_pow_Rmult.
1 subgoal

  n : nat
  IHn : forall x : R, sum_geom n x * x + 1 = x ^ (n + 1) + sum_geom n x
  x : R
  ============================
  x ^ S (S n) + sum_geom n x * x + 1 =
  x ^ S (n + 1) + (x ^ S n + sum_geom n x)
```

Then we apply the induction hypothesis

```
sum_geom_p < rewrite Rplus_assoc.
1 subgoal

  n : nat
  IHn : forall x : R, sum_geom n x * x + 1 = x ^ (n + 1) + sum_geom n x
  x : R
  ============================
  x ^ S (S n) + (sum_geom n x * x + 1) =
  x ^ S (n + 1) + (x ^ S n + sum_geom n x)
```

```
sum_geom_p < rewrite IHn.
1 subgoal

  n : nat
  IHn : forall x : R, sum_geom n x * x + 1 = x ^ (n + 1) + sum_geom n x
  x : R
  ============================
  x ^ S (S n) + (x ^ (n + 1) + sum_geom n x) =
  x ^ S (n + 1) + (x ^ S n + sum_geom n x)

sum_geom_p < rewrite Nat.add_1_r.
1 subgoal

  n : nat
  IHn : forall x : R, sum_geom n x * x + 1 = x ^ (n + 1) + sum_geom n x
  x : R
  ============================
  x ^ S (S n) + (x ^ S n + sum_geom n x) =
  x ^ S (S n) + (x ^ S n + sum_geom n x)

sum_geom_p < reflexivity.
No more subgoals.

sum_geom_p < Qed.
(induction n).
 (intros x).
 (simpl).
 (rewrite Rmult_1_l).
 (rewrite Rmult_1_r).
 reflexivity.

 (intros x).
 (simpl).
 (rewrite Rmult_plus_distr_r).
 (rewrite tech_pow_Rmult).
 (rewrite tech_pow_Rmult).
 (rewrite Rmult_comm with (r1 := (x ^ S n)) (r2 := x)).
 (rewrite tech_pow_Rmult).
 (destruct (IHn x) as [H _]).
 (rewrite Rplus_assoc).
 (rewrite IHn).
 (rewrite Nat.add_1_r).
 reflexivity.

Qed.
sum_geom_p is defined
```

4.4.3 Fibonacci Numbers

We can use the `Fixpoint` operator to define more general sequences, such as the Fibonacci numbers. We will use the integers here in order to be able to use this later on in the book. Thus we will start with fibonacci

```
  Require Import ZArith.
  Require Import Znumtheory.
  Open Scope Z_scope.


Fixpoint fibonacci (n:nat) : Z :=
   match n with
     | O => 1
     | S O => 1
     | S (S n as p) => fibonacci p + fibonacci n
   end.
```

We can check our code by computing some examples

```
Coq < Eval compute in (fibonacci 2).
     = 2
     : Z
Coq < Eval compute in (fibonacci 5).
     = 8
     : Z
```

As a first example, let's try to prove that Fibonacci numbers are all positive. fibonacci_pos

```
Coq < Lemma fibonacci_pos : forall n, 0 <= fibonacci n.
1 subgoal

  ============================
  forall n : nat, 0 <= fibonacci n
```

We could try induction on n, but since the Fibonacci recurrence depends on the last two numbers, it is likely that we will need strong induction. But how do we do strong induction in Coq? We want to explicitly add all the hypotheses from strong induction. We will do this using the *cut* tactic. This tactic is the inverse of *modus ponens*, since instead of proving some goal G, we prove $H \implies G$ and H for some hypothesis H. These are equivalent using the *modus ponens* inference rule. Our new hypothesis will be that the Fibonacci numbers are positive for every smaller number.

```
fibonacci_pos < cut (forall N n, (n<N)%nat -> 0<=fibonacci n).
2 subgoals

  ============================
  (forall N n : nat, (n < N)%nat -> 0 <= fibonacci n) ->
  forall n : nat, 0 <= fibonacci n

subgoal 2 is:
 forall N n : nat, (n < N)%nat -> 0 <= fibonacci n
```

This first goal can be solved using `eauto`, but lets see if we can do it by hand. First we introduce hypotheses. We see that we can use H to get a specific hypothesis which makes the goal true, namely when $N = n + 1$. We can apply that hypothesis, and then we have only to prove that $n < n + 1$.

```
fibonacci_pos < intros H n.
2 subgoals

  H : forall N n : nat, (n < N)%nat -> 0 <= fibonacci n
  n : nat
  ============================
  0 <= fibonacci n

fibonacci_pos < pose (Hn := H (S n) n).
2 subgoals

  H : forall N n : nat, (n < N)%nat -> 0 <= fibonacci n
  n : nat
  Hn := H (S n) n : (n < S n)%nat -> 0 <= fibonacci n
  ============================
  0 <= fibonacci n

subgoal 2 is:
 forall N n : nat, (n < N)%nat -> 0 <= fibonacci n

fibonacci_pos < apply Hn.
2 subgoals
```

```
  H : forall N n : nat, (n < N)%nat -> 0 <= fibonacci n
  n : nat
  Hn := H (S n) n : (n < S n)%nat -> 0 <= fibonacci n
  ============================
  (n < S n)%nat

subgoal 2 is:
 forall N n : nat, (n < N)%nat -> 0 <= fibonacci n

fibonacci_pos < apply le_lt_n_Sm.
2 subgoals

  H : forall N n : nat, (n < N)%nat -> 0 <= fibonacci n
  n : nat
  Hn := H (S n) n : (n < S n)%nat -> 0 <= fibonacci n
  ============================
  (n <= n)%nat

subgoal 2 is:
 forall N n : nat, (n < N)%nat -> 0 <= fibonacci n

fibonacci_pos < apply le_n.
1 subgoal

  ============================
  forall N n : nat, (n < N)%nat -> 0 <= fibonacci n
```

Now we have only our strong induction to prove. We start with induction on N. The base case is true because the false hypothesis $n < 0$ implies anything. We can prove it is false using the inversion tactic, telling it to look at the first expression.

```
fibonacci_pos < induction N.
2 subgoals

  ============================
  forall n : nat, (n < 0)%nat -> 0 <= fibonacci n

subgoal 2 is:
 forall n : nat, (n < S N)%nat -> 0 <= fibonacci n

fibonacci_pos < inversion 1.
1 subgoal

  N : nat
  IHN : forall n : nat, (n < N)%nat -> 0 <= fibonacci n
  ============================
  forall n : nat, (n < S N)%nat -> 0 <= fibonacci n
```

After introducing hypotheses, we want to get rid of the base cases for the Fibonacci recurrence. Therefore, we separate n into the two cases for natural numbers, and prove the zero case. To prove $0 \leq 1$, we could apply theorems by hand, as before, but we will just use the *auto* tactic since we now know how it goes.

```
fibonacci_pos < intros n H.
1 subgoal

  N : nat
  IHN : forall n : nat, (n < N)%nat -> 0 <= fibonacci n
  n : nat
  H : (n < S N)%nat
  ============================
  0 <= fibonacci n
```

```
fibonacci_pos < destruct n.
2 subgoals

  N : nat
  IHN : forall n : nat, (n < N)%nat -> 0 <= fibonacci n
  H : (0 < S N)%nat
  ============================
  0 <= fibonacci 0

subgoal 2 is:
 0 <= fibonacci (S n)

fibonacci_pos < simpl.
2 subgoals

  N : nat
  IHN : forall n : nat, (n < N)%nat -> 0 <= fibonacci n
  H : (0 < S N)%nat
  ============================
  0 <= 1

subgoal 2 is:
 0 <= fibonacci (S n)

fibonacci_pos < auto with zarith.
1 subgoal

  N : nat
  IHN : forall n : nat, (n < N)%nat -> 0 <= fibonacci n
  n : nat
  H : (S n < S N)%nat
  ============================
  0 <= fibonacci (S n)
```

We do this a second time to get rid of the second base case in the recurrence.

```
fibonacci_pos < destruct n.
2 subgoals

  N : nat
  IHN : forall n : nat, (n < N)%nat -> 0 <= fibonacci n
  H : (1 < S N)%nat
  ============================
  0 <= fibonacci 1

subgoal 2 is:
 0 <= fibonacci (S (S n))

fibonacci_pos < simpl.
2 subgoals

  N : nat
  IHN : forall n : nat, (n < N)%nat -> 0 <= fibonacci n
  H : (1 < S N)%nat
  ============================
  0 <= 1

subgoal 2 is:
 0 <= fibonacci (S (S n))

fibonacci_pos < auto with zarith.
1 subgoal

  N : nat
  IHN : forall n : nat, (n < N)%nat -> 0 <= fibonacci n
  n : nat
  H : (S (S n) < S N)%nat
  ============================
  0 <= fibonacci (S (S n))
```

Now we can use the Fibonacci recurrence. Unfortunately the `simpl` tactic will not give us what we want here, so we are forced to do the unfolding by hand using the `change` tactic. Remember that `change` allows us to replace one term with another that has the same definition.

```
fibonacci_pos < change (0 <= fibonacci (S n) + fibonacci n).
1 subgoal

  N : nat
  IHN : forall n : nat, (n < N)%nat -> 0 <= fibonacci n
  n : nat
  H : (S (S n) < S N)%nat
  ============================
  0 <= fibonacci (S n) + fibonacci n
```

Now we can use our induction hypothesis to generate the two specific hypotheses we need.

```
fibonacci_pos < pose (Hn := IHN n).
1 subgoal

  N : nat
  IHN : forall n : nat, (n < N)%nat -> 0 <= fibonacci n
  n : nat
  H : (S (S n) < S N)%nat
  Hn := IHN n : (n < N)%nat -> 0 <= fibonacci n
  ============================
  0 <= fibonacci (S n) + fibonacci n

fibonacci_pos < pose (HSn := IHN (S n)).
1 subgoal

  N : nat
  IHN : forall n : nat, (n < N)%nat -> 0 <= fibonacci n
  n : nat
  H : (S (S n) < S N)%nat
  Hn := IHN n : (n < N)%nat -> 0 <= fibonacci n
  HSn := IHN (S n) : (S n < N)%nat -> 0 <= fibonacci (S n)
  ============================
  0 <= fibonacci (S n) + fibonacci n
```

From hypothesis H, it is clear both that $n < N$ and $n + 1 < N$. We could generate those inequalities, and use them with Hn and HSn to prove that F_n and F_{n+1} are non-negative. Finally, we would use the fact that the sum of two non-negative numbers is non-negative. However, all this can be done by the `omega` tactic, which can handle inequalities and arithmetic.

```
fibonacci_pos < omega.
No more subgoals.

fibonacci_pos < Qed.
(cut (forall N n, (n < N)%nat -> 0 <= fibonacci n)).
 (intros H n).
 (pose (Hn := H (S n) n)).
 (apply Hn).
 (apply le_lt_n_Sm).
 (apply le_n).

 (induction N).
  (inversion 1).

  (intros n H).
  (destruct n).
   (simpl).
   auto with zarith.
   (destruct n).
    (simpl).
    auto with zarith.

    (change (0 <= fibonacci (S n) + fibonacci n)).
    (pose (Hn := IHN n)).
    (pose (HSn := IHN (S n))).
    omega.

Qed.
fibonacci_pos is defined
```

Next, we will prove that the Fibonacci recurrence is monotone, meaning that each successive number is larger than the last. fibonacci_monotone

```
Coq < Lemma fibonacci_monotone : forall n m, (n<=m)%nat -> fibonacci n <= fibonacci m.
1 subgoal

  ============================
  forall n m : nat, (n <= m)%nat -> fibonacci n <= fibonacci m
```

We will prove this by induction. However, we will not induct on n or m by using `nat_ind`, but rather we will induct on $n \leq m$ using `le_ind`. We cannot use `le_n` for the base case, since that only applies to natural numbers. Instead we can use the integer version, or just let the automatic tactic handle it.

```
fibonacci_monotone < induction 1.
2 subgoals

  n : nat
  ============================
  fibonacci n <= fibonacci n

subgoal 2 is:
 fibonacci n <= fibonacci (S m)

fibonacci_monotone < apply Zle_refl.
1 subgoal

  n, m : nat
  H : (n <= m)%nat
  IHle : fibonacci n <= fibonacci m
  ============================
  fibonacci n <= fibonacci (S m)
```

Now we can use the transitive property of the less-then-or-equal-to relation. The first precondition is obvious, since it is one of our hypotheses, but the second will require more work.

```
fibonacci_monotone < apply Zle_trans with (m := (fibonacci m)).
2 subgoals

  n, m : nat
  H : (n <= m)%nat
  IHle : fibonacci n <= fibonacci m
  ============================
  fibonacci n <= fibonacci m

subgoal 2 is:
 fibonacci m <= fibonacci (S m)

fibonacci_monotone < exact IHle.
1 subgoal
```

```
  n, m : nat
  H : (n <= m)%nat
  IHle : fibonacci n <= fibonacci m
  ============================
  fibonacci m <= fibonacci (S m)
```

None of the prior hypotheses will help us here, so we can remove them using `clear`. We ultimately want to use the recurrence to prove this inequality, but we must take care of the base cases first. Thus we split m into the two constructor for natural numbers, and prove the base case by reflexivity.

```
fibonacci_monotone < clear.
1 subgoal

  m : nat
  ============================
  fibonacci m <= fibonacci (S m)

fibonacci_monotone < destruct m.
2 subgoals

  ============================
  fibonacci 0 <= fibonacci 1

subgoal 2 is:
 fibonacci (S m) <= fibonacci (S (S m))

fibonacci_monotone < simpl.
2 subgoals

  ============================
  1 <= 1

subgoal 2 is:
 fibonacci (S m) <= fibonacci (S (S m))

fibonacci_monotone < apply Zle_refl.
1 subgoal

  m : nat
  ============================
  fibonacci (S m) <= fibonacci (S (S m))
```

Now we can use the recurrence, just as we did in the previous proof. We could then prove this by canceling F_{m+1} from both sides and using our previous lemma. However, we can do this in a slightly simpler way using the automatic techniques. We first assume our previous lemma applied to m using `generalize`. We could have made it a new hypothesis using `pose` in the same way. Finally, we can use the automated solver `omega` which will find the solution from these inequalities.

```
fibonacci_monotone < change (fibonacci (S m) <= fibonacci (S m)+fibonacci m).
1 subgoal

  m : nat
  ============================
  fibonacci (S m) <= fibonacci (S m) + fibonacci m

fibonacci_monotone < generalize (fibonacci_pos m).
1 subgoal

  m : nat
  ============================
  0 <= fibonacci m -> fibonacci (S m) <= fibonacci (S m) + fibonacci m
```

```
fibonacci_monotone < omega.
No more subgoals.

fibonacci_monotone < Qed.
(induction 1).
 (apply Zle_refl).

 (apply Zle_trans with (m := fibonacci m)).
  exact IHle.

  clear.
  (destruct m).
   (simpl).
   (apply Zle_refl).

   (change (fibonacci (S m) <= fibonacci (S m) + fibonacci m)).
   (generalize (fibonacci_pos m)).
   omega.

Qed.
fibonacci_monotone is defined
```

4.4.4 Even and Odd Numbers

We will call the domain of natural numbers $\mathbb{N}$, and introduce numerical predicates which will be used in many examples and problems,

- $O(x)$: True if the input x is an odd natural number,
- $E(x)$: True if the input x is an even natural number.

We also have that $\neg O(x) = E(x)$ and $\neg E(x) = O(x)$. Consider the following statement,

For every natural number x, if $x^2 + 2x + 7$ is even, then x is odd.

This can be translated into a statement of predicate logic

$$\forall x : \mathbb{N}, E(x^2 + 2x + 7) \implies O(x) \tag{4.54}$$

For any given integer x, we could determine x^2+2x+7 by arithmetic, and then check the predicates using division and remainder. However, we cannot exhaust all integers, so this proof strategy will not work. However, we can characterize even numbers as those expressible as $2k$ for some integer k, and likewise odd numbers as $2k + 1$. This will allow us to carry out the proof for arbitrary k.

A direct proof would proceed by modus ponens, namely assuming $E(x^2 - 2x + 7)$, and then demonstrating $O(x)$. We would have that $x^2 - 2x + 7 = 2k$ for some integer k, and $x = 2l + 1$ for some l. We have that

$$(2l + 1)^2 + 2(2l + 1) + 7 = 2k \tag{4.55}$$

$$4l^2 + 4l + 1 + 4l + 2 + 7 = 2k \tag{4.56}$$

$$2l^2 + 4l + 5 = k \tag{4.57}$$

but this does not show us that l is an integer. Let us try to prove this by contraposition. In this case we would assume the negation of the consequent, namely that $\neg O(x) = E(x)$, and demonstrate the negation of the antecedent. So we assume $x = 2l$,

$$\begin{aligned} x^2 + 2x + 7 &= (2l)^2 + 2(2l) + 7 \\ &= 4l^2 + 4l + 7 \\ &= 4l^2 + 4l + 6 + 1 \\ &= 2(2l^2 + 2l + 3) + 1 \\ &= 2m + 1 \end{aligned} \tag{4.58}$$

where we used the fact that $l \in \mathbb{Z}$ implies that $2l^2 + 2l + 3 = m \in \mathbb{Z}$. Thus we have

$$E(x^2 + 2x + 7) = E(2m + 1) = \mathrm{F} \tag{4.59}$$

or

$$\neg O(x) \implies \neg E(x^2 + 2x + 7) \tag{4.60}$$

so that by the implication $(\neg Q \implies \neg P) \implies (P \implies Q)$,

$$E(x^2 + 2x + 7) \implies O(x). \tag{4.61}$$

Suppose that we would like to prove that

If x is even, then $x + 10$ is even.

which is the logical statement

$$\forall x : \mathbb{Z}, E(x) \implies E(x + 10). \tag{4.62}$$

We could prove this directly, since for even x

$$x + 10 = 2k + 10 \tag{4.63}$$
$$= 2(k + 5) \tag{4.64}$$
$$= 2m \tag{4.65}$$

so that $E(x + 10)$.

On the other hand, a proof by contradiction assumes the negation of the statement to be proved and tried to derive the false statement from it. Thus we would assume

$$\neg\left(\forall x : \mathbb{Z}, E(x) \implies E(x + 10)\right) = \exists x : \mathbb{Z}, \neg\left(E(x) \implies E(x + 10)\right) \tag{4.66}$$
$$= \exists x : \mathbb{Z}, \neg\left(\neg E(x) \vee E(x + 10)\right) \tag{4.67}$$
$$= \exists x : \mathbb{Z}, E(x) \wedge \neg E(x + 10) \tag{4.68}$$
$$= \exists x : \mathbb{Z}, E(x) \wedge O(x + 10) \tag{4.69}$$

We can use our proof from above to show the $E(x)$ implies $E(x+10)$. Thus we have

$$E(x+10) \wedge O(x+10) = \mathrm{F}. \tag{4.70}$$

We have derived a contradiction, and thus our original assumption must be false, making the statement we want to prove true.

This is more work in Coq since we have assumed a lot of intermediate results. This makes a good point about automation, namely that it is easy to assume a lot of complicated things when you fail to explain it to a computer. We begin by loading the library for arithmetic with natural numbers and defining an *inductive predicate* even

```
Require Import Arith.

Inductive even : nat -> Prop :=
  even0 : even 0
| evenS : forall x:nat, even x -> even (S (S x)).
```

which produces a proposition for every even number. The `even0` and `evenS` components are theorems which prove the propositions on the right. This is somewhat counterintuitive, since we would normally also define the odd cases. We will not need them as we can make use of the *inversion* tactic. This tactic analyzes all the constructors of the inductive predicate, discards the ones that could not have been applied, and when some constructors could have been applied, it creates a new goal where the premises of this constructor are added in the context. Lets look at a simple proof not_even_1,

```
Coq < Lemma not_even_1 : ~even 1.
1 subgoal

  ============================
  ~ even 1

not_even_1 < intros even1.
1 subgoal

  even1 : even 1
  ============================
  False

not_even_1 < inversion even1.
No more subgoals.

not_even_1 < Qed.
(intros even1).
(inversion even1).

Qed.
not_even_1 is defined
```

Since no constructor can conclude to the proposition `even 1` (using `evenS` would require that `1 = S (S x)`, and using `even0` would require that `1 = 0`), the premise in never true and we are finished.

Now we can try something a little harder, which illustrates how to use the theorems which build our inductive predicate. We will prove that $2n$ is always even, or more formally even_double_p

```
Coq < Lemma even_double_p : forall n, even(2*n).
1 subgoal

  ============================
  forall n : nat, even (2 * n)
```

We begin by invoking induction, and solving the base case.

```
even_double_p < induction n.
2 subgoals

  ============================
  even (2 * 0)

subgoal 2 is:
 even (2 * S n)

even_double_p < simpl.
2 subgoals

  ============================
  even 0

subgoal 2 is:
 even (2 * S n)

even_double_p < apply even0.
1 subgoal

  n : nat
  IHn : even (2 * n)
  ============================
  even (2 * S n)
```

Notice how we used the theorem `even0` to prove `even 0`. Next we will attack the inductive step by rewriting it so that we can use our `evenS` theorem,

```
even_double_p < simpl.
1 subgoal

  n : nat
  IHn : even (2 * n)
  ============================
  even (S (n + S (n + 0)))

even_double_p < rewrite Nat.add_succ_r.
1 subgoal

  n : nat
  IHn : even (2 * n)
  ============================
  even (S (S (n + (n + 0))))

even_double_p < apply evenS.
1 subgoal

  n : nat
  IHn : even (2 * n)
  ============================
  even (n + (n + 0))

even_double_p < assumption.
No more subgoals.

even_double_p < Qed.
(induction n).
 (simpl).
 (apply even0).
 (simpl).
 (rewrite Nat.add_succ_r).
 (apply evenS).
 assumption.

Qed.
even_double_p is defined
```

and the proof is complete.

We can state the problem above in terms of our inductive predicate, and handle the base case with repeated application of `evenS`, even_plus10_p

```
Coq < Lemma even_plus10_p : forall n, even n -> even (n + 10).
1 subgoal

  ============================
  forall n : nat, even n -> even (n + 10)

even_plus10_p < induction n.
2 subgoals

  ============================
  even 0 -> even (0 + 10)

subgoal 2 is:
 even (S n) -> even (S n + 10)

even_plus10_p < intro H0.
2 subgoals

  H0 : even 0
  ============================
  even (0 + 10)

subgoal 2 is:
 even (S n) -> even (S n + 10)

even_plus10_p < apply evenS.
2 subgoals

  H0 : even 0
  ============================
  even 8

subgoal 2 is:
 even (S n) -> even (S n + 10)

even_plus10_p < apply evenS; apply evenS; apply evenS; apply evenS.
2 subgoals

  H0 : even 0
  ============================
  even 0

subgoal 2 is:
 even (S n) -> even (S n + 10)

even_plus10_p < assumption.
1 subgoal

  n : nat
  IHn : even n -> even (n + 10)
  ============================
  even (S n) -> even (S n + 10)
```

We use the introduction tactic for the implication, but it looks like we are stuck, because we do not have a case for the single successor (which would be an odd number). However, the inversion tactic comes to our rescue,

```
even_plus10_p < intro H1.
1 subgoal

  n : nat
  IHn : even n -> even (n + 10)
  H1 : even (S n)
  ============================
  even (S n + 10)

even_plus10_p < inversion H1.
1 subgoal

  n : nat
  IHn : even n -> even (n + 10)
  H1 : even (S n)
  x : nat
  H0 : even x
  H : S x = n
  ============================
  even (S (S x) + 10)
```

Now all that remains is to eliminate the 10. We start by moving the addition to successor using the repeat tactic, which applies it argument until it cannot be applied anymore. Then we eliminate the 0.

```
even_plus10_p < repeat rewrite Nat.add_succ_r.
1 subgoal

  A : Set
  P, Q : A -> Prop
  n : nat
  IHn : even n -> even (n + 10)
  H1 : even (S n)
  x : nat
  H0 : even x
  H : S x = n
  ============================
  even (S (S (S (S (S (S (S (S (S (S (S (S x) + 0)))))))))))

even_plus10_p < rewrite Nat.add_0_r.
1 subgoal

  A : Set
  P, Q : A -> Prop
  n : nat
  IHn : even n -> even (n + 10)
  H1 : even (S n)
  x : nat
  H0 : even x
  H : S x = n
  ============================
  even (S (S (S (S (S (S (S (S (S (S (S (S x))))))))))))
```

Finally we remove the successor with `evenS`,

```
even_plus10_p < repeat apply evenS.
1 subgoal

  A : Set
  P, Q : A -> Prop
  n : nat
  IHn : even n -> even (n + 10)
  H1 : even (S n)
  x : nat
  H0 : even x
  H : S x = n
  ============================
  even x

even_plus10_p < exact H0.
No more subgoals.

even_plus10_p < Qed.
(induction n).
 intro H1.
 (repeat apply evenS).
 exact H1.

 intro H1.
 (inversion H1).
 (repeat rewrite Nat.add_succ_r).
 (rewrite Nat.add_0_r).
 (repeat apply evenS).
 exact H0.

Qed.
even_plus10_p is defined
```

It is more convenient, however, to use the induction theorem for even numbers directly. Restarting the proof, we can introduce hypotheses, and then eliminate our new hypothesis `even n`. This applies the induction theorem for the even type, generating the base case and inductive step for us.

```
even_plus10_p < Restart.
1 subgoal

  ============================
  forall n : nat, even n -> even (n + 10)

even_plus10_p < intros n Heven.
1 subgoal

  n : nat
  Heven : even n
  ============================
  even (n + 10)

even_plus10_p < elim Heven.
2 subgoals

  n : nat
  Heven : even n
  ============================
  even (0 + 10)

subgoal 2 is:
 forall x : nat, even x -> even (x + 10) -> even (S (S x) + 10)
```

We can easily prove the base case as before

```
even_plus10_p < rewrite Nat.add_0_l.
2 subgoals

  n : nat
  Heven : even n
  ============================
  even 10

subgoal 2 is:
 forall x : nat, even x -> even (x + 10) -> even (S (S x) + 10)
```

```
even_plus10_p < repeat apply evenS.
2 subgoals

  n : nat
  Heven : even n
  ============================
  even 0

subgoal 2 is:
 forall x : nat, even x -> even (x + 10) -> even (S (S x) + 10)

even_plus10_p < apply even0.
1 subgoal

  n : nat
  Heven : even n
  ============================
  forall x : nat, even x -> even (x + 10) -> even (S (S x) + 10)
```

To prove the inductive step, we just need to introduce hypotheses, apply the inductive theorem from our even type, and use the inductive hypothesis.

```
even_plus10_p < intros x Hxeven Hx10even.
1 subgoal

  n : nat
  Heven : even n
  x : nat
  Hxeven : even x
  Hx10even : even (x + 10)
  ============================
  even (S (S x) + 10)

even_plus10_p < repeat rewrite Nat.add_succ_l.
1 subgoal

  n : nat
  Heven : even n
  x : nat
  Hxeven : even x
  Hx10even : even (x + 10)
  ============================
  even (S (S (x + 10)))

even_plus10_p < apply evenS.
1 subgoal

  n : nat
  Heven : even n
  x : nat
  Hxeven : even x
  Hx10even : even (x + 10)
  ============================
  even (x + 10)

even_plus10_p < exact Hx10even.
No more subgoals.

even_plus10_p < Qed.
(intros n Heven).
(elim Heven).
 (rewrite Nat.add_0_l).
 (repeat apply evenS).
 (apply even0).

 (intros x Hxeven Hx10even).
 (repeat apply Nat.add_succ_l).
 (repeat rewrite Nat.add_succ_l).
 (apply evenS).
 exact Hx10even.

Qed.
even_plus10_p is defined
```

This is simpler than navigating the natural number induction using inversion, and also more clearly shows how induction over even numbers is structured. This will prepare us for induction over more complicated types.

As a final warmup, let's prove that any number of the form $2n + 1$ is not even. We first state the lemma and prove the base case using inversion odd_notdouble_p

```
tester < Lemma odd_notdouble_p : forall n, ~even(2*n+1).
1 subgoal

  ============================
  forall n : nat, ~ even (2 * n + 1)

odd_notdouble_p < induction n.
2 subgoals

  ============================
  ~ even (2 * 0 + 1)

subgoal 2 is:
 ~ even (2 * S n + 1)

odd_notdouble_p < simpl.
2 subgoals

  ============================
  ~ even 1

subgoal 2 is:
 ~ even (2 * S n + 1)

odd_notdouble_p < intros H1even.
2 subgoals

  H1even : even 1
  ============================
  False

subgoal 2 is:
 ~ even (2 * S n + 1)

odd_notdouble_p < inversion H1even.
1 subgoal

  n : nat
  IHn : ~ even (2 * n + 1)
  ============================
  ~ even (2 * S n + 1)
```

Next we simplify the induction step so that it has the form of our recursion and then use inversion

```
odd_notdouble_p < simpl.
1 subgoal

  n : nat
  IHn : ~ even (2 * n + 1)
  ============================
  ~ even (S (n + S (n + 0) + 1))

odd_notdouble_p < rewrite Nat.add_succ_r.
1 subgoal

  n : nat
  IHn : ~ even (2 * n + 1)
  ============================
  ~ even (S (S (n + S (n + 0) + 0)))

odd_notdouble_p < rewrite plus_0_r.
1 subgoal

  n : nat
  IHn : ~ even (2 * n + 1)
  ============================
  ~ even (S (S (n + S (n + 0))))

odd_notdouble_p < rewrite plus_0_r.
1 subgoal

  n : nat
  IHn : ~ even (2 * n + 1)
  ============================
  ~ even (S (S (n + S n)))

odd_notdouble_p < intro HSneven.
1 subgoal

  n : nat
  IHn : ~ even (2 * n + 1)
  HSneven : even (S (S (n + S n)))
  ============================
  False

odd_notdouble_p < inversion HSneven.
1 subgoal

  n : nat
  IHn : ~ even (2 * n + 1)
  HSneven : even (S (S (n + S n)))
  x : nat
  H0 : even (n + S n)
  H : x = n + S n
  ============================
  False
```

Now we `destruct` the induction hypothesis `IHn`, do a small rewrite of our goal, and use the `H0` hypothesis generated from the induction,

```
odd_notdouble_p < destruct IHn.
1 subgoal

  n : nat
  HSneven : even (S (S (n + S n)))
  x : nat
  H0 : even (n + S n)
  H : x = n + S n
  ============================
  even (2 * n + 1)

odd_notdouble_p < simpl.
1 subgoal

  n : nat
  HSneven : even (S (S (n + S n)))
  x : nat
  H0 : even (n + S n)
  H : x = n + S n
  ============================
  even (n + (n + 0) + 1)

odd_notdouble_p < rewrite plus_0_r.
1 subgoal

  n : nat
  HSneven : even (S (S (n + S n)))
  x : nat
  H0 : even (n + S n)
  H : x = n + S n
  ============================
  even (n + n + 1)

odd_notdouble_p < rewrite Nat.add_1_r.
1 subgoal

  n : nat
  HSneven : even (S (S (n + S n)))
  x : nat
  H0 : even (n + S n)
  H : x = n + S n
  ============================
  even (S (n + n))

odd_notdouble_p < rewrite <- Nat.add_succ_r.
1 subgoal

  n : nat
  HSneven : even (S (S (n + S n)))
  x : nat
  H0 : even (n + S n)
  H : x = n + S n
  ============================
  even (n + S n)

odd_notdouble_p < assumption.
No more subgoals.

odd_notdouble_p < Qed.
(induction n).
 intro H1even.
 (inversion H1even).

 (simpl).
 (rewrite Nat.add_succ_r).
 (rewrite plus_0_r).
 (rewrite plus_0_r).
 intro HSneven.
 (inversion HSneven).
 (destruct IHn).
 (simpl).
 (rewrite plus_0_r).
 (rewrite Nat.add_1_r).
 (rewrite <- Nat.add_succ_r).
 assumption.

Qed.
odd_notdouble_p is defined
```

so that our theorem is proved.

We will need a subsidiary result in order to prove our main theorem, namely that evenness of n implies that half of it exists, $2m = n$. First we state the lemma, and `elim` to reason by cases, even_mult

```
Coq < Lemma even_mult : forall x, even x -> exists y, x = 2*y.
```

```
1 subgoal

  ============================
  forall x : nat, even x -> exists y : nat, x = 2 * y

even_mult < intros x H.
1 subgoal

  x : nat
  H : even x
  ============================
  exists y : nat, x = 2 * y

even_mult < elim H.
2 subgoals

  x : nat
  H : even x
  ============================
  exists y : nat, 0 = 2 * y

subgoal 2 is:
 forall x0 : nat,
 even x0 -> (exists y : nat, x0 = 2 * y) -> exists y : nat, S (S x0) = 2 * y
```

The 0 case is easy, since the answer is 0 as well.

```
even_mult < exists 0.
2 subgoals

  x : nat
  H : even x
  ============================
  0 = 2 * 0

subgoal 2 is:
 forall x0 : nat,
 even x0 -> (exists y : nat, x0 = 2 * y) -> exists y : nat, S (S x0) = 2 * y

even_mult < ring.
1 subgoal

  x : nat
  H : even x
  ============================
  forall x0 : nat,
  even x0 -> (exists y : nat, x0 = 2 * y) -> exists y : nat, S (S x0) = 2 * y
```

Now we use introduction to remove the universal quantifier, even assumption, and induction hypothesis. We destruct the existential quantifier in the induction hypothesis, and use the resulting theorem in our goal to locate a witness.

```
even_mult < intros x0 Hx0even IHx0.
1 subgoal

  x : nat
  H : even x
  x0 : nat
  Hx0even : even x0
  IHx0 : exists y : nat, x0 = 2 * y
  ============================
  exists y : nat, S (S x0) = 2 * y

even_mult < destruct IHx0 as [y Heq].
1 subgoal

  x : nat
  H : even x
  x0 : nat
  Hx0even : even x0
  y : nat
  Heq : x0 = 2 * y
  ============================
  exists y0 : nat, S (S x0) = 2 * y0

even_mult < rewrite Heq.
1 subgoal

  x : nat
  H : even x
  x0 : nat
  Hx0even : even x0
  y : nat
  Heq : x0 = 2 * y
  ============================
  exists y0 : nat, S (S (2 * y)) = 2 * y0

even_mult < exists (S y).
1 subgoal

  x : nat
  H : even x
  x0 : nat
  Hx0even : even x0
  y : nat
  Heq : x0 = 2 * y
  ============================
  S (S (2 * y)) = 2 * S y

even_mult < ring.
No more subgoals.

even_mult < Qed.
(intros x H).
(elim H).
 exists 0.
 ring.

 (intros x0 Hx0even IHx0).
 (destruct IHx0 as [y Heq]).
 (rewrite Heq).
 exists (S y).
 ring.

Qed.
even_mult is defined
```

We are finally ready to attack the problem at the top of this section. First, we define the lemma, start the induction, and simplify the base case even_cp_p

```
Coq < Lemma even_cp_p : forall n, even n -> ~even(n*n + 2*n + 7).
1 subgoal

  ============================
  forall n : nat, even n -> ~ even (n * n + 2 * n + 7)

even_cp_p < induction n.
2 subgoals

  ============================
  even 0 -> ~ even (0 * 0 + 2 * 0 + 7)

subgoal 2 is:
 even (S n) -> ~ even (S n * S n + 2 * S n + 7)
```

```
even_cp_p < intro H0even.
2 subgoals

  H0even : even 0
  ============================
  ~ even (0 * 0 + 2 * 0 + 7)

subgoal 2 is:
 even (S n) -> ~ even (S n * S n + 2 * S n + 7)

even_cp_p < simpl.
2 subgoals

  H0even : even 0
  ============================
  ~ even 7

subgoal 2 is:
 even (S n) -> ~ even (S n * S n + 2 * S n + 7)

even_cp_p < intro H7even.
2 subgoals

  H0even : even 0
  H7even : even 7
  ============================
  False

subgoal 2 is:
 even (S n) -> ~ even (S n * S n + 2 * S n + 7)
```

Now we just need to repeatedly use inversion to whittle this down to the impossible `even 1`,

```
even_cp_p < inversion H7even.
2 subgoals

  H0even : even 0
  H7even : even 7
  x : nat
  H0 : even 5
  H : x = 5
  ============================
  False

subgoal 2 is:
 even (S n) -> ~ even (S n * S n + 2 * S n + 7)

even_cp_p < inversion H0.
2 subgoals

  H0even : even 0
  H7even : even 7
  x : nat
  H0 : even 5
  H : x = 5
  x0 : nat
  H2 : even 3
  H1 : x0 = 3
  ============================
  False

subgoal 2 is:
 even (S n) -> ~ even (S n * S n + 2 * S n + 7)

even_cp_p < inversion H2.
2 subgoals
```

```
  H0even : even 0
  H7even : even 7
  x : nat
  H0 : even 5
  H : x = 5
  x0 : nat
  H2 : even 3
  H1 : x0 = 3
  x1 : nat
  H4 : even 1
  H3 : x1 = 1
  ============================
  False

subgoal 2 is:
 even (S n) -> ~ even (S n * S n + 2 * S n + 7)

even_cp_p < inversion H4.
1 subgoal

  n : nat
  IHn : even n -> ~ even (n * n + 2 * n + 7)
  ============================
  even (S n) -> ~ even (S n * S n + 2 * S n + 7)
```

Now we can attack the inductive step. First, we introduce the antecedent, apply our previous theorem, destruct the existential quantifier and use it to simplify our goal. Note here that we can apply a theorem to any hypothesis using the `apply Th in H` syntax.

```
even_cp_p < intro HSneven.
1 subgoal

  n : nat
  IHn : even n -> ~ even (n * n + 2 * n + 7)
  HSneven : even (S n)
  ============================
  ~ even (S n * S n + 2 * S n + 7)

even_cp_p < apply even_mult in HSneven.
1 subgoal

  n : nat
  IHn : even n -> ~ even (n * n + 2 * n + 7)
  HSneven : exists y : nat, S n = 2 * y
  ============================
  ~ even (S n * S n + 2 * S n + 7)

even_cp_p < destruct HSneven.
1 subgoal

  n : nat
  IHn : even n -> ~ even (n * n + 2 * n + 7)
  x : nat
  H : S n = 2 * x
  ============================
  ~ even (S n * S n + 2 * S n + 7)

even_cp_p < rewrite H.
1 subgoal

  n : nat
  IHn : even n -> ~ even (n * n + 2 * n + 7)
  x : nat
  H : S n = 2 * x
  ============================
  ~ even (2 * x * (2 * x) + 2 * (2 * x) + 7)
```

We now introduce an assertion, which mirrors our result from the proof by hand above in Eq. 4.58. It is easily proved by using the witness we found above and the `ring` tactic for simplification.

```
even_cp_p < assert (HOdd : exists m, 2 * x * (2 * x) + 2 * (2 * x) + 7 = 2*m + 1).
2 subgoals

  n : nat
  IHn : even n -> ~ even (n * n + 2 * n + 7)
  x : nat
  H : S n = 2 * x
  ============================
  exists m : nat, 2 * x * (2 * x) + 2 * (2 * x) + 7 = 2 * m + 1

subgoal 2 is:
 ~ even (2 * x * (2 * x) + 2 * (2 * x) + 7)

even_cp_p < exists (2*x*x + 2*x + 3).
2 subgoals

  n : nat
  IHn : even n -> ~ even (n * n + 2 * n + 7)
  x : nat
  H : S n = 2 * x
  ============================
  2 * x * (2 * x) + 2 * (2 * x) + 7 = 2 * (2 * x * x + 2 * x + 3) + 1

subgoal 2 is:
 ~ even (2 * x * (2 * x) + 2 * (2 * x) + 7)

even_cp_p < ring.
1 subgoal

  n : nat
  IHn : even n -> ~ even (n * n + 2 * n + 7)
  x : nat
  H : S n = 2 * x
  HOdd : exists m : nat, 2 * x * (2 * x) + 2 * (2 * x) + 7 = 2 * m + 1
  ============================
  ~ even (2 * x * (2 * x) + 2 * (2 * x) + 7)
```

All that is left is to destruct our assertion, use it to rewrite the goal, and then apply our theorem about odd numbers,

```
even_cp_p < destruct HOdd.
1 subgoal

  n : nat
  IHn : even n -> ~ even (n * n + 2 * n + 7)
  x : nat
  H : S n = 2 * x
  x0 : nat
  H0 : 2 * x * (2 * x) + 2 * (2 * x) + 7 = 2 * x0 + 1
  ============================
  ~ even (2 * x * (2 * x) + 2 * (2 * x) + 7)

even_cp_p < rewrite H0.
1 subgoal

  n : nat
  IHn : even n -> ~ even (n * n + 2 * n + 7)
  x : nat
  H : S n = 2 * x
  x0 : nat
  H0 : 2 * x * (2 * x) + 2 * (2 * x) + 7 = 2 * x0 + 1
  ============================
  ~ even (2 * x0 + 1)
```

```
even_cp_p < apply odd_notdouble_p.
No more subgoals.

even_cp_p < Qed.
(induction n).
 intro H0even.
 (simpl).
 intro H7even.
 (inversion H7even).
 (inversion H0).
 (inversion H2).
 (inversion H4).

 intro HSneven.
 (apply even_mult in HSneven).
 (destruct HSneven).
 (rewrite H).
 (assert (HOdd : exists m, 2 * x * (2 * x) + 2 * (2 * x) + 7 = 2 * m + 1)).
  exists (2 * x * x + 2 * x + 3).
  ring.

  (destruct HOdd).
  (rewrite H0).
  (apply odd_notdouble_p).

Qed.
even_cp_p is defined
```

so that the result is proved.

4.5 Lists

We can define lists of natural numbers inductively, as things to which we can add a number n at the beginning,

```
Inductive list : Type :=
  | nil : list
  | H (n : nat) (tail : list) : list.
```

We denote the empty list by `nil`, and use the function `H` to push a natural number, which we will call the *head*, onto the front of a list passed as the second argument, which we will call the *tail* of the list. We can use this pattern to build some sample lists,

```
Coq < Compute let l := nil in l.
     = nil
     : list

Coq < Compute let l := H 5 nil in l.
     = H 5 nil
     : list

Coq < Compute let l := H 4 (H 5 nil) in l.
     = H 4 (H 5 nil)
     : list

Coq < Compute let l := H 3 (H 4 (H 5 nil)) in l.
     = H 3 (H 4 (H 5 nil))
     : list

Coq < Compute let l := H 3 (H 3 (H 4 (H 5 nil))) in l.
     = H 3 (H 3 (H 4 (H 5 nil)))
     : list
```

This type also comes with an induction theorem,

```
Coq < Print list_ind.
list_ind =
fun P : list -> Prop => list_rect P
     : forall P : list -> Prop,
       P nil ->
       (forall (n : nat) (tail : list), P tail -> P (H n tail)) ->
       forall l : list, P l
```

In order to prove that a predicate is true for every list, we must prove that it is true for the empty list, and then we must prove that if the predicate is true for a list, it is also true for that list with any natural number added to the front.

4.5.1 Operations

Now let's define some operations on our list data structure. For instance, the length of a list can be calculated recursively,

```
Fixpoint length (l : list) : nat :=
  match l with
  | nil => 0
  | H _ lt => S (length lt)
  end.
```

The length of the empty list is zero, and the length is one more than the length of its tail. We can verify our implementation with a few examples.

```
Coq < Compute let l := nil in length l.
     = 0
     : nat

Coq < Compute let l := H 5 nil in length l.
     = 1
     : nat

Coq < Compute let l := H 4 (H 5 nil) in length l.
     = 2
     : nat

Coq < Compute let l := H 3 (H 4 (H 5 nil)) in length l.
     = 3
     : nat

Coq < Compute let l := H 3 (H 3 (H 4 (H 5 nil))) in length l.
     = 4
     : nat
```

Another operation would be the concatenation of lists, which works by appending the head of list1 to the concatenation of the the tail with list2.

```
Fixpoint concat (l1 : list) (l2 : list) : list :=
  match l1 with
  | nil => l2
  | H n l1t => H n (concat l1t l2)
  end.
```

and we can confirm this with some examples.

```
Coq < Compute
let l1 := H 5 nil in
let l2 := H 4 (H 5 nil) in
concat l1 l2.
Coq < = H 5 (H 4 (H 5 nil))
```

```
     : list

Coq < Compute
let l1 := H 3 (H 4 (H 5 nil)) in
let l2 := H 3 (H 3 (H 4 (H 5 nil))) in
concat l1 l2.
Coq < = H 3 (H 4 (H 5 (H 3 (H 3 (H 4 (H 5 nil))))))
     : list
```

And finally we can implement list reversal by concatenating the reverse of the tail with the head.

```
Fixpoint reverse (l : list) : list :=
  match l with
  | nil => nil
  | H n lt => concat (reverse lt) (H n nil)
  end.

Coq < Compute let l := nil in reverse l.
     = nil
     : list

Coq < Compute let l := H 5 nil in reverse l.
     = H 5 nil
     : list

Coq < Compute let l := H 4 (H 5 nil) in reverse l.
     = H 5 (H 4 nil)
     : list

Coq < Compute let l := H 3 (H 4 (H 5 nil)) in reverse l.
     = H 5 (H 4 (H 3 nil))
     : list

Coq < Compute let l := H 3 (H 3 (H 4 (H 5 nil))) in reverse l.
     = H 5 (H 4 (H 3 (H 3 nil)))
     : list
```

We can prove some elementary theorems about the length function, namely that it is a function,

```
Coq < Theorem length_equal : forall l1 l2 : list, l1 = l2 -> length l1 = length l2.
1 subgoal

  ============================
  forall l1 l2 : list, l1 = l2 -> length l1 = length l2

length_equal < intros l1 l2 H.
1 subgoal

  l1, l2 : list
  H : l1 = l2
  ============================
  length l1 = length l2

length_equal < rewrite H.
1 subgoal

  l1, l2 : list
  H : l1 = l2
  ============================
  length l2 = length l2

length_equal < reflexivity.
No more subgoals.

length_equal < Qed.
```

```
(intros l1 l2 H).
(rewrite H).
reflexivity.

Qed.
length_equal is defined
```

and the converse,

```
Coq < Theorem length_not_equal : forall l1 l2 : list, length l1 <> length l2 -> l1 <> l2.
1 subgoal

  ============================
  forall l1 l2 : list, length l1 <> length l2 -> l1 <> l2

length_not_equal < intros l1 l2 H0 H1.
1 subgoal

  l1, l2 : list
  H0 : length l1 <> length l2
  H1 : l1 = l2
  ============================
  False

length_not_equal < apply length_equal in H1.
1 subgoal

  l1, l2 : list
  H0 : length l1 <> length l2
  H1 : length l1 = length l2
  ============================
  False

length_not_equal < apply H0.
1 subgoal

  l1, l2 : list
  H0 : length l1 <> length l2
  H1 : length l1 = length l2
  ============================
  length l1 = length l2

length_not_equal < exact H1.
No more subgoals.

length_not_equal < Qed.
(intros l1 l2 H0 H1).
(apply length_equal in H1).
(apply H0).
exact H1.

Qed.
length_not_equal is defined
```

and that something of length zero has to be the empty list.

```
Coq < Theorem length_zero : forall l : list, length l = 0 -> l = nil.
1 subgoal

  ============================
  forall l : list, length l = 0 -> l = nil

length_zero < intros l H.
1 subgoal

  l : list
  H : length l = 0
  ============================
  l = nil

length_zero < destruct l.
2 subgoals

  H : length nil = 0
  ============================
  nil = nil

subgoal 2 is:
 Top.H n l = nil

length_zero < reflexivity.
1 subgoal

  n : nat
  l : list
  H : length (Top.H n l) = 0
  ============================
  Top.H n l = nil

length_zero < inversion H.
No more subgoals.

length_zero < Qed.
(intros l H).
(destruct l).
 reflexivity.

 (inversion H).

Qed.
length_zero is defined
```

We can prove similar, straightforward theorems about concatenation, namely that concatenating the empty list does not change anything

```
Coq < Theorem concat_nil : forall l : list, l = concat l nil.
1 subgoal

  ============================
  forall l : list, l = concat l nil

concat_nil < induction l.
2 subgoals

  ============================
  nil = concat nil nil

subgoal 2 is:
 H n l = concat (H n l) nil

concat_nil < simpl.
2 subgoals

  ============================
  nil = nil

subgoal 2 is:
 H n l = concat (H n l) nil

concat_nil < reflexivity.
1 subgoal

  n : nat
  l : list
  IHl : l = concat l nil
  ============================
  H n l = concat (H n l) nil

concat_nil < simpl.
1 subgoal

  n : nat
  l : list
  IHl : l = concat l nil
  ============================
  H n l = H n (concat l nil)

concat_nil < rewrite <- IHl.
1 subgoal

  n : nat
  l : list
  IHl : l = concat l nil
  ============================
  H n l = H n l

concat_nil < reflexivity.
No more subgoals.

concat_nil < Qed.
(induction l).
 (simpl).
 reflexivity.

 (simpl).
 (rewrite <- IHl).
 reflexivity.

Qed.
concat_nil is defined
```

and the symmetric counterpart,

```
Coq < Theorem concat_nil_t : forall l : list, l = concat nil l.
1 subgoal

  ============================
  forall l : list, l = concat nil l

concat_nil_t < intro l.
1 subgoal

  l : list
  ============================
  l = concat nil l

concat_nil_t < simpl.
1 subgoal

  l : list
  ============================
  l = l

concat_nil_t < reflexivity.
No more subgoals.

concat_nil_t < Qed.
intro l.
(simpl).
reflexivity.

Qed.
concat_nil_t is defined
```

and that the length of the concatenated string is the sum of the original lengths

```
Coq < Theorem concat_length : forall l1 l2 : list, length (concat l1 l2) = length l1 + length l2.
1 subgoal

  ============================
  forall l1 l2 : list, length (concat l1 l2) = length l1 + length l2

concat_length < induction l1.
2 subgoals

  ============================
  forall l2 : list, length (concat nil l2) = length nil + length l2

subgoal 2 is:
 forall l2 : list, length (concat (H n l1) l2) = length (H n l1) + length l2

concat_length < intro l2.
2 subgoals

  l2 : list
  ============================
  length (concat nil l2) = length nil + length l2

subgoal 2 is:
 forall l2 : list, length (concat (H n l1) l2) = length (H n l1) + length l2

concat_length < simpl.
2 subgoals

  l2 : list
  ============================
  length l2 = length l2

subgoal 2 is:
 forall l2 : list, length (concat (H n l1) l2) = length (H n l1) + length l2

concat_length < reflexivity.
1 subgoal

  n : nat
  l1 : list
  IHl1 : forall l2 : list, length (concat l1 l2) = length l1 + length l2
  ============================
  forall l2 : list, length (concat (H n l1) l2) = length (H n l1) + length l2

concat_length < intro l2.
1 subgoal

  n : nat
  l1 : list
  IHl1 : forall l2 : list, length (concat l1 l2) = length l1 + length l2
  l2 : list
  ============================
  length (concat (H n l1) l2) = length (H n l1) + length l2

concat_length < simpl.
1 subgoal

  n : nat
  l1 : list
  IHl1 : forall l2 : list, length (concat l1 l2) = length l1 + length l2
  l2 : list
  ============================
  S (length (concat l1 l2)) = S (length l1 + length l2)

concat_length < rewrite IHl1.
1 subgoal
```

```
  n : nat
  l1 : list
  IHl1 : forall l2 : list, length (concat l1 l2) = length l1 + length l2
  l2 : list
  ============================
  S (length l1 + length l2) = S (length l1 + length l2)

concat_length < reflexivity.
No more subgoals.

concat_length < Qed.
(induction l1).
 intro l2.
 (simpl).
 reflexivity.

 intro l2.
 (simpl).
 (rewrite IHl1).
 reflexivity.

Qed.
concat_length is defined
```

and finally that concatenation is associative.

```
Coq < Theorem concat_assoc : forall l1 l2 l3 : list, concat (concat l1 l2) l3 = concat l1 (concat l2 l3).
1 subgoal

  ============================
  forall l1 l2 l3 : list, concat (concat l1 l2) l3 = concat l1 (concat l2 l3)

concat_assoc < induction l1.
2 subgoals

  ============================
  forall l2 l3 : list, concat (concat nil l2) l3 = concat nil (concat l2 l3)

subgoal 2 is:
 forall l2 l3 : list,
 concat (concat (H n l1) l2) l3 = concat (H n l1) (concat l2 l3)

concat_assoc < intros l2 l3.
2 subgoals

  l2, l3 : list
  ============================
  concat (concat nil l2) l3 = concat nil (concat l2 l3)

subgoal 2 is:
 forall l2 l3 : list,
 concat (concat (H n l1) l2) l3 = concat (H n l1) (concat l2 l3)

concat_assoc < simpl.
2 subgoals

  l2, l3 : list
  ============================
  concat l2 l3 = concat l2 l3

subgoal 2 is:
 forall l2 l3 : list,
 concat (concat (H n l1) l2) l3 = concat (H n l1) (concat l2 l3)

concat_assoc < reflexivity.
1 subgoal

  n : nat
  l1 : list
  IHl1 : forall l2 l3 : list,
         concat (concat l1 l2) l3 = concat l1 (concat l2 l3)
  ============================
  forall l2 l3 : list,
  concat (concat (H n l1) l2) l3 = concat (H n l1) (concat l2 l3)

concat_assoc < intros l2 l3.
1 subgoal

  n : nat
  l1 : list
  IHl1 : forall l2 l3 : list,
         concat (concat l1 l2) l3 = concat l1 (concat l2 l3)
  l2, l3 : list
  ============================
  concat (concat (H n l1) l2) l3 = concat (H n l1) (concat l2 l3)

concat_assoc < simpl.
1 subgoal

  n : nat
  l1 : list
  IHl1 : forall l2 l3 : list,
         concat (concat l1 l2) l3 = concat l1 (concat l2 l3)
  l2, l3 : list
  ============================
  H n (concat (concat l1 l2) l3) = H n (concat l1 (concat l2 l3))

concat_assoc < rewrite IHl1.
1 subgoal

  n : nat
  l1 : list
  IHl1 : forall l2 l3 : list,
         concat (concat l1 l2) l3 = concat l1 (concat l2 l3)
  l2, l3 : list
  ============================
  H n (concat l1 (concat l2 l3)) = H n (concat l1 (concat l2 l3))

concat_assoc < reflexivity.
No more subgoals.

concat_assoc < Qed.
(induction l1).
 (intros l2 l3).
 (simpl).
 reflexivity.

 (intros l2 l3).
 (simpl).
 (rewrite IHl1).
 reflexivity.

Qed.
concat_assoc is defined
```

We can also show some simple theorems about list reversal. The empty list is the same reversed,

```
Coq < Theorem reverse_nil : nil = reverse nil.
1 subgoal

  ============================
  nil = reverse nil

reverse_nil < reflexivity.
No more subgoals.
```

```
reverse_nil < Qed.
reflexivity.

Qed.
reverse_nil is defined
```

as is a single element list

```
Coq < Theorem reverse_single : forall n : nat, H n nil = reverse (H n nil).
1 subgoal

  ============================
  forall n : nat, H n nil = reverse (H n nil)

reverse_single < intro n.
1 subgoal

  n : nat
  ============================
  H n nil = reverse (H n nil)

reverse_single < reflexivity.
No more subgoals.

reverse_single < Qed.
intro n.
reflexivity.

Qed.
reverse_single is defined
```

The length of a list and its reversal are the same.

```
Coq < Theorem reverse_length : forall l : list, length l = length (reverse l).
1 subgoal

  ============================
  forall l : list, length l = length (reverse l)

reverse_length < induction l.
2 subgoals

  ============================
  length nil = length (reverse nil)

subgoal 2 is:
 length (H n l) = length (reverse (H n l))

reverse_length < reflexivity.
1 subgoal

  n : nat
  l : list
  IHl : length l = length (reverse l)
  ============================
  length (H n l) = length (reverse (H n l))

reverse_length < simpl.
1 subgoal

  n : nat
  l : list
  IHl : length l = length (reverse l)
  ============================
  S (length l) = length (concat (reverse l) (H n nil))

reverse_length < rewrite concat_length.
```

```
1 subgoal

  n : nat
  l : list
  IHl : length l = length (reverse l)
  ============================
  S (length l) = length (reverse l) + length (H n nil)

reverse_length < rewrite <- IHl.
1 subgoal

  n : nat
  l : list
  IHl : length l = length (reverse l)
  ============================
  S (length l) = length l + length (H n nil)

reverse_length < simpl.
1 subgoal

  n : nat
  l : list
  IHl : length l = length (reverse l)
  ============================
  S (length l) = length l + 1

reverse_length < ring.
No more subgoals.

reverse_length < Qed.
(induction l).
 reflexivity.

 (simpl).
 (rewrite concat_length).
 (rewrite <- IHl).
 (simpl).
 ring.

Qed.
reverse_length is defined
```

The reverse operation is a function

```
Coq < Theorem reverse_equal : forall l1 l2 : list, l1 = l2 -> reverse l1 = reverse l2.
1 subgoal

  ============================
  forall l1 l2 : list, l1 = l2 -> reverse l1 = reverse l2

reverse_equal < intros l1 l2 H.
1 subgoal

  l1, l2 : list
  H : l1 = l2
  ============================
  reverse l1 = reverse l2

reverse_equal < apply (f_equal reverse).
1 subgoal

  l1, l2 : list
  H : l1 = l2
  ============================
  l1 = l2

reverse_equal < exact H.
No more subgoals.
```

```
reverse_equal < Qed.
(intros l1 l2 H).
(apply (f_equal reverse)).
exact H.

Qed.
reverse_equal is defined
```

and it commutes with concatenation,

```
Coq < Theorem reverse_concat : forall l1 l2 : list, reverse (concat l1 l2) = concat (reverse l2) (reverse l1).
1 subgoal

  ============================
  forall l1 l2 : list,
  reverse (concat l1 l2) = concat (reverse l2) (reverse l1)

reverse_concat < induction l1.
2 subgoals

  ============================
  forall l2 : list,
  reverse (concat nil l2) = concat (reverse l2) (reverse nil)

subgoal 2 is:
 forall l2 : list,
 reverse (concat (H n l1) l2) = concat (reverse l2) (reverse (H n l1))

reverse_concat < intro l2.
2 subgoals

  l2 : list
  ============================
  reverse (concat nil l2) = concat (reverse l2) (reverse nil)

subgoal 2 is:
 forall l2 : list,
 reverse (concat (H n l1) l2) = concat (reverse l2) (reverse (H n l1))

reverse_concat < simpl.
2 subgoals

  l2 : list
  ============================
  reverse l2 = concat (reverse l2) nil

subgoal 2 is:
 forall l2 : list,
 reverse (concat (H n l1) l2) = concat (reverse l2) (reverse (H n l1))

reverse_concat < rewrite <- concat_nil.
2 subgoals

  l2 : list
  ============================
  reverse l2 = reverse l2

subgoal 2 is:
 forall l2 : list,
 reverse (concat (H n l1) l2) = concat (reverse l2) (reverse (H n l1))

reverse_concat < reflexivity.
1 subgoal

  n : nat
  l1 : list
  IHl1 : forall l2 : list,
         reverse (concat l1 l2) = concat (reverse l2) (reverse l1)
  ============================
  forall l2 : list,
  reverse (concat (H n l1) l2) = concat (reverse l2) (reverse (H n l1))

reverse_concat < intro l2.
1 subgoal

  n : nat
  l1 : list
  IHl1 : forall l2 : list,
         reverse (concat l1 l2) = concat (reverse l2) (reverse l1)
  l2 : list
  ============================
  reverse (concat (H n l1) l2) = concat (reverse l2) (reverse (H n l1))

reverse_concat < simpl.
1 subgoal

  n : nat
  l1 : list
  IHl1 : forall l2 : list,
         reverse (concat l1 l2) = concat (reverse l2) (reverse l1)
  l2 : list
  ============================
  concat (reverse (concat l1 l2)) (H n nil) =
  concat (reverse l2) (concat (reverse l1) (H n nil))

reverse_concat < rewrite <- (concat_assoc (reverse l2) (reverse l1) (H n nil)).
1 subgoal

  n : nat
  l1 : list
  IHl1 : forall l2 : list,
         reverse (concat l1 l2) = concat (reverse l2) (reverse l1)
  l2 : list
  ============================
  concat (reverse (concat l1 l2)) (H n nil) =
  concat (concat (reverse l2) (reverse l1)) (H n nil)

reverse_concat < rewrite <- IHl1.
1 subgoal

  n : nat
  l1 : list
  IHl1 : forall l2 : list,
         reverse (concat l1 l2) = concat (reverse l2) (reverse l1)
  l2 : list
  ============================
  concat (reverse (concat l1 l2)) (H n nil) =
  concat (reverse (concat l1 l2)) (H n nil)

reverse_concat < reflexivity.
No more subgoals.

reverse_concat < Qed.
(induction l1).
 intro l2.
 (simpl).
 (rewrite <- concat_nil).
 reflexivity.

 intro l2.
 (simpl).
 (rewrite <- (concat_assoc (reverse l2) (reverse l1) (H n nil))).
 (rewrite <- IHl1).
 reflexivity.
```

```
Qed.
reverse_concat is defined
```

List reversal is an involution, which means that if applied twice it return the original input. We will use this later on. To prove this, we will use a fact we proved earlier, that the reversal of a concatenation is the concatenation of the reversals.

```
Coq < Theorem reverse_involution : forall l : list, l = reverse (reverse l).
1 subgoal

  ============================
  forall l : list, l = reverse (reverse l)

reverse_involution < induction l.
2 subgoals

  ============================
  nil = reverse (reverse nil)

subgoal 2 is:
 H n l = reverse (reverse (H n l))

reverse_involution < reflexivity.
1 subgoal

  n : nat
  l : list
  IHl : l = reverse (reverse l)
  ============================
  H n l = reverse (reverse (H n l))

reverse_involution < simpl.
1 subgoal

  n : nat
  l : list
  IHl : l = reverse (reverse l)
  ============================
  H n l = reverse (concat (reverse l) (H n nil))

reverse_involution < rewrite reverse_concat.
1 subgoal

  n : nat
  l : list
  IHl : l = reverse (reverse l)
  ============================
  H n l = concat (reverse (H n nil)) (reverse (reverse l))

reverse_involution < rewrite <- IHl.
1 subgoal

  n : nat
  l : list
  IHl : l = reverse (reverse l)
  ============================
  H n l = concat (reverse (H n nil)) l

reverse_involution < simpl.
1 subgoal

  n : nat
  l : list
  IHl : l = reverse (reverse l)
  ============================
  H n l = H n l
```

```
reverse_involution < reflexivity.
No more subgoals.

reverse_involution < Qed.
(induction l).
 reflexivity.

 (simpl).
 (rewrite reverse_concat).
 (rewrite <- IHl).
 (simpl).
 reflexivity.

Qed.
reverse_involution is defined
```

The reversal function is also injective, which we can prove using our involution theorem.

```
Coq < Theorem reverse_injective : forall l1 l2 : list, reverse l1 = reverse l2 -> l1 = l2.
1 subgoal

  ============================
  forall l1 l2 : list, reverse l1 = reverse l2 -> l1 = l2

reverse_injective < intros l1 l2 H.
1 subgoal

  l1, l2 : list
  H : reverse l1 = reverse l2
  ============================
  l1 = l2

reverse_injective < rewrite (reverse_involution l1).
1 subgoal

  l1, l2 : list
  H : reverse l1 = reverse l2
  ============================
  reverse (reverse l1) = l2

reverse_injective < rewrite H.
1 subgoal

  l1, l2 : list
  H : reverse l1 = reverse l2
  ============================
  reverse (reverse l2) = l2

reverse_injective < rewrite <- (reverse_involution l2).
1 subgoal

  l1, l2 : list
  H : reverse l1 = reverse l2
  ============================
  l2 = l2

reverse_injective < reflexivity.
No more subgoals.

reverse_injective < Qed.
(intros l1 l2 H).
(rewrite (reverse_involution l1)).
(rewrite H).
(rewrite <- (reverse_involution l2)).
reflexivity.
```

```
Qed.
reverse_injective is defined
```

We can also use involution to prove that reversals can be flipped from one list to another in equalities.

```
Coq < Theorem reverse_left_right : forall l1 l2 : list, l1 = reverse l2 -> reverse l1 = l2.
1 subgoal

  ============================
  forall l1 l2 : list, l1 = reverse l2 -> reverse l1 = l2

reverse_left_right < intros l1 l2 H.
1 subgoal

  l1, l2 : list
  H : l1 = reverse l2
  ============================
  reverse l1 = l2

reverse_left_right < rewrite (reverse_involution l2).
1 subgoal

  l1, l2 : list
  H : l1 = reverse l2
  ============================
  reverse l1 = reverse (reverse l2)

reverse_left_right < rewrite <- H.
1 subgoal

  l1, l2 : list
  H : l1 = reverse l2
  ============================
  reverse l1 = reverse l1

reverse_left_right < reflexivity.
No more subgoals.

reverse_left_right < Qed.
(intros l1 l2 H).
(rewrite (reverse_involution l2)).
(rewrite <- H).
reflexivity.

Qed.
reverse_left_right is defined
```

Finally we can prove that concatenation is injective for both arguments. The proof for the first argument is easier. The key step is to use the fact that the `H` function that builds lists is injective, with the `injection` tactic.

```
Coq < Theorem concat_injective_right : forall l1 l2 l3 : list, concat l1 l2 = concat l1 l3 -> l2 = l3.
1 subgoal

  ============================
  forall l1 l2 l3 : list, concat l1 l2 = concat l1 l3 -> l2 = l3

concat_injective_right < induction l1 ; intros ; simpl in H0.
2 subgoals

  l2, l3 : list
  H0 : l2 = l3
  ============================
  l2 = l3

subgoal 2 is:
 l2 = l3

concat_injective_right < exact H0.
1 subgoal

  n : nat
  l1 : list
  IHl1 : forall l2 l3 : list, concat l1 l2 = concat l1 l3 -> l2 = l3
  l2, l3 : list
  H0 : H n (concat l1 l2) = H n (concat l1 l3)
  ============================
  l2 = l3

concat_injective_right < apply IHl1.
1 subgoal

  n : nat
  l1 : list
  IHl1 : forall l2 l3 : list, concat l1 l2 = concat l1 l3 -> l2 = l3
  l2, l3 : list
  H0 : H n (concat l1 l2) = H n (concat l1 l3)
  ============================
  concat l1 l2 = concat l1 l3

concat_injective_right < injection H0.
1 subgoal

  n : nat
  l1 : list
  IHl1 : forall l2 l3 : list, concat l1 l2 = concat l1 l3 -> l2 = l3
  l2, l3 : list
  H0 : H n (concat l1 l2) = H n (concat l1 l3)
  ============================
  concat l1 l2 = concat l1 l3 -> concat l1 l2 = concat l1 l3

concat_injective_right < intro H1.
1 subgoal

  n : nat
  l1 : list
  IHl1 : forall l2 l3 : list, concat l1 l2 = concat l1 l3 -> l2 = l3
  l2, l3 : list
  H0 : H n (concat l1 l2) = H n (concat l1 l3)
  H1 : concat l1 l2 = concat l1 l3
  ============================
  concat l1 l2 = concat l1 l3

concat_injective_right < exact H1.
No more subgoals.

concat_injective_right < Qed.
(induction l1; intros **; simpl in H0).
 exact H0.

 (apply IHl1).
 injection H0.
 intro H1.
 exact H1.

Qed.
concat_injective_right is defined
```

In order to prove injectivity for the second argument, we will use reversal to swap it with the first. At the start, we use the reversal involution to rewrite our hypothesis,

```
Coq < Theorem concat_injective_left : forall l1 l2 l3 : list, concat l1 l3 = concat l2 l3 -> l1 = l2.
```

```
1 subgoal

  ============================
  forall l1 l2 l3 : list, concat l1 l3 = concat l2 l3 -> l1 = l2

concat_injective_left < intros l1 l2 l3 H.
1 subgoal

  l1, l2, l3 : list
  H : concat l1 l3 = concat l2 l3
  ============================
  l1 = l2

concat_injective_left < rewrite (reverse_involution (concat l1 l3)) in H.
1 subgoal

  l1, l2, l3 : list
  H : reverse (reverse (concat l1 l3)) = concat l2 l3
  ============================
  l1 = l2

concat_injective_left < rewrite (reverse_involution (concat l2 l3)) in H.
1 subgoal

  l1, l2, l3 : list
  H : reverse (reverse (concat l1 l3)) = reverse (reverse (concat l2 l3))
  ============================
  l1 = l2
```

Now we can pull concatenation through one reversal.

```
concat_injective_left < rewrite (reverse_concat l1 l3) in H.
1 subgoal

  l1, l2, l3 : list
  H : reverse (concat (reverse l3) (reverse l1)) =
      reverse (reverse (concat l2 l3))
  ============================
  l1 = l2

concat_injective_left < rewrite (reverse_concat l2 l3) in H.
1 subgoal

  l1, l2, l3 : list
  H : reverse (concat (reverse l3) (reverse l1)) =
      reverse (concat (reverse l3) (reverse l2))
  ============================
  l1 = l2
```

Finally, we use the fact that reversal is injective, and that concatenation is injective in the first argument.

```
concat_injective_left < apply reverse_injective in H.
1 subgoal

  l1, l2, l3 : list
  H : concat (reverse l3) (reverse l1) = concat (reverse l3) (reverse l2)
  ============================
  l1 = l2

concat_injective_left < apply concat_injective_right in H.
1 subgoal

  l1, l2, l3 : list
  H : reverse l1 = reverse l2
  ============================
  l1 = l2
```

```
concat_injective_left < apply reverse_injective.
1 subgoal

  l1, l2, l3 : list
  H : reverse l1 = reverse l2
  ============================
  reverse l1 = reverse l2

concat_injective_left < exact H.
No more subgoals.

concat_injective_left < Qed.
(intros l1 l2 l3 H).
(rewrite (reverse_involution (concat l1 l3)) in H).
(rewrite (reverse_involution (concat l2 l3)) in H).
(rewrite (reverse_concat l1 l3) in H).
(rewrite (reverse_concat l2 l3) in H).
(apply reverse_injective in H).
(apply concat_injective_right in H).
(apply reverse_injective).
exact H.

Qed.
concat_injective_left is defined
```

4.5.2 Palindromes

Suppose that we would like to know whether a list is palindromic, meaning it is equal to its reverse. We can simply define this as a predicate

```
Definition palindrome (l : list) : Prop := l = reverse l.
```

However, I could also give an explicit, inductive algorithm for constructing lists that are palindromes,

```
Inductive palindrome_i : list -> Prop :=
  | palindrome_i_nil : palindrome_i nil
  | palindrome_i_single (n : nat) : palindrome_i (H n nil)
  | palindrome_i_more (n : nat) (l : list) (P : palindrome_i l) : palindrome_i (H n (concat l (H n nil))).
```

Let's take a look at the induction theorem for this palindrome type

```
Coq < Check palindrome_i_ind.
palindrome_i_ind
     : forall P : list -> Prop,
       P nil ->
       (forall n : nat, P (H n nil)) ->
       (forall (n : nat) (l : list),
        palindrome_i l -> P l -> P (H n (concat l (H n nil)))) ->
       forall l : list, palindrome_i l -> P l
```

It says that for some predicate to be true on all palindromes, it must be true for the empty list, true for all one element lists, and finally if it is true for a given palindrome list, it must be true when we append any number to the front and back.

If our algorithm is correct, then any list it constructs ought to be judged a palindrome by our predicate, which we can easily formalize.

```
Coq < Theorem palindrome_equiv_def_1 : forall l : list, palindrome_i l -> palindrome l.
1 subgoal

  ============================
  forall l : list, palindrome_i l -> palindrome l
```

```
palindrome_equiv_def_1 < unfold palindrome.
1 subgoal

  ============================
  forall l : list, palindrome_i l -> l = reverse l

palindrome_equiv_def_1 < intros l H.
1 subgoal

  l : list
  H : palindrome_i l
  ============================
  l = reverse l
```

Now we want to do induction on the list, but we would like to use the induction theorem for our palindrome construction, meaning we will have to prove equivalence for the empty list, single element lists, and finally list with the same number stuck in front and back.

```
palindrome_equiv_def_1 < induction H.
3 subgoals

  ============================
  nil = reverse nil

subgoal 2 is:
 H n nil = reverse (H n nil)
subgoal 3 is:
 H n (concat l (H n nil)) = reverse (H n (concat l (H n nil)))
```

The base cases are straightforward.

```
palindrome_equiv_def_1 < reflexivity.
2 subgoals

  n : nat
  ============================
  H n nil = reverse (H n nil)

subgoal 2 is:
 H n (concat l (H n nil)) = reverse (H n (concat l (H n nil)))

palindrome_equiv_def_1 < reflexivity.
1 subgoal

  n : nat
  l : list
  H0 : palindrome_i l
  IHpalindrome_i : l = reverse l
  ============================
  H n (concat l (H n nil)) = reverse (H n (concat l (H n nil)))
```

Once we simplify, it is clear we need to interchange concatenation and reversal, and then use our induction hypothesis.

```
palindrome_equiv_def_1 < simpl.
1 subgoal

  n : nat
  l : list
  H0 : palindrome_i l
  IHpalindrome_i : l = reverse l
  ============================
  H n (concat l (H n nil)) = concat (reverse (concat l (H n nil))) (H n nil)
```

```
palindrome_equiv_def_1 < rewrite reverse_concat.
1 subgoal

  n : nat
  l : list
  H0 : palindrome_i l
  IHpalindrome_i : l = reverse l
  ============================
  H n (concat l (H n nil)) =
  concat (concat (reverse (H n nil)) (reverse l)) (H n nil)

palindrome_equiv_def_1 < rewrite <- IHpalindrome_i.
1 subgoal

  n : nat
  l : list
  H0 : palindrome_i l
  IHpalindrome_i : l = reverse l
  ============================
  H n (concat l (H n nil)) = concat (concat (reverse (H n nil)) l) (H n nil)

palindrome_equiv_def_1 < simpl.
1 subgoal

  n : nat
  l : list
  H0 : palindrome_i l
  IHpalindrome_i : l = reverse l
  ============================
  H n (concat l (H n nil)) = H n (concat l (H n nil))

palindrome_equiv_def_1 < reflexivity.
No more subgoals.

palindrome_equiv_def_1 < Qed.
(unfold palindrome).
(intros l H).
(induction H).
 reflexivity.

 reflexivity.

 (simpl).
 (rewrite reverse_concat).
 (rewrite <- IHpalindrome_i).
 (simpl).
 reflexivity.

Qed.
palindrome_equiv_def_1 is defined
```

This is a good start, but we would also like our algorithm to be complete. This means that it can generate any palindrome, or equivalently if our predicate says a list is a palindrome, so does our inductive predicate.

```
Theorem palindrome_equiv_def_2 : forall l : list, palindrome l -> palindrome_i l.
```

However, here we will need strong induction because we want to look at lists with both the first and last element removed. Therefore we will induct in the length of the list first.

```
Coq < Lemma palindrome_equiv_def_2_strong : forall (m : nat) (l : list), length l <= m -> palindrome l -> palindrome_i l.
1 subgoal

  ============================
  forall (m : nat) (l : list),
  length l <= m -> palindrome l -> palindrome_i l

palindrome_equiv_def_2_strong < induction m.
2 subgoals

  ============================
  forall l : list, length l <= 0 -> palindrome l -> palindrome_i l

subgoal 2 is:
 forall l : list, length l <= S m -> palindrome l -> palindrome_i l

palindrome_equiv_def_2_strong < intros l Hlen Hp.
2 subgoals

  l : list
  Hlen : length l <= 0
  Hp : palindrome l
  ============================
  palindrome_i l

subgoal 2 is:
 forall l : list, length l <= S m -> palindrome l -> palindrome_i l
```

Notice now that only the empty list satisfies the length requriement. We destruct l, and use the `constructor` tactic to pick the right theorem from our inductive type, and then use inversion to handle all other lists.

```
palindrome_equiv_def_2_strong < destruct l.
3 subgoals

  Hlen : length nil <= 0
  Hp : palindrome nil
  ============================
  palindrome_i nil

subgoal 2 is:
 palindrome_i (H n l)
subgoal 3 is:
 forall l : list, length l <= S m -> palindrome l -> palindrome_i l

palindrome_equiv_def_2_strong < constructor.
2 subgoals

  n : nat
  l : list
  Hlen : length (H n l) <= 0
  Hp : palindrome (H n l)
  ============================
  palindrome_i (H n l)

subgoal 2 is:
 forall l : list, length l <= S m -> palindrome l -> palindrome_i l

palindrome_equiv_def_2_strong < inversion Hlen.
1 subgoal

  m : nat
  IHm : forall l : list, length l <= m -> palindrome l -> palindrome_i l
  ============================
  forall l : list, length l <= S m -> palindrome l -> palindrome_i l
```

Now we destruct our list twice in order to prove our two base cases.

```
palindrome_equiv_def_2_strong < intros l Hlen Hp.
1 subgoal

  m : nat
  IHm : forall l : list, length l <= m -> palindrome l -> palindrome_i l
  l : list
  Hlen : length l <= S m
  Hp : palindrome l
  ============================
  palindrome_i l

palindrome_equiv_def_2_strong < destruct l.
2 subgoals

  m : nat
  IHm : forall l : list, length l <= m -> palindrome l -> palindrome_i l
  Hlen : length nil <= S m
  Hp : palindrome nil
  ============================
  palindrome_i nil

subgoal 2 is:
 palindrome_i (H n l)

palindrome_equiv_def_2_strong < constructor.
1 subgoal

  m : nat
  IHm : forall l : list, length l <= m -> palindrome l -> palindrome_i l
  n : nat
  l : list
  Hlen : length (H n l) <= S m
  Hp : palindrome (H n l)
  ============================
  palindrome_i (H n l)

palindrome_equiv_def_2_strong < destruct l.
2 subgoals

  m : nat
  IHm : forall l : list, length l <= m -> palindrome l -> palindrome_i l
  n : nat
  Hlen : length (H n nil) <= S m
  Hp : palindrome (H n nil)
  ============================
  palindrome_i (H n nil)

subgoal 2 is:
 palindrome_i (H n (H n0 l))

palindrome_equiv_def_2_strong < constructor.
1 subgoal

  m : nat
  IHm : forall l : list, length l <= m -> palindrome l -> palindrome_i l
  n, n0 : nat
  l : list
  Hlen : length (H n (H n0 l)) <= S m
  Hp : palindrome (H n (H n0 l))
  ============================
  palindrome_i (H n (H n0 l))
```

Now we use a series of rewrites to change our palindrome hypothesis into a form that individually reverses the parts.

```
palindrome_equiv_def_2_strong < unfold palindrome in Hp.
1 subgoal

  m : nat
  IHm : forall l : list, length l <= m -> palindrome l -> palindrome_i l
  n, n0 : nat
  l : list
  Hlen : length (H n (H n0 l)) <= S m
  Hp : H n (H n0 l) = reverse (H n (H n0 l))
  ============================
  palindrome_i (H n (H n0 l))

palindrome_equiv_def_2_strong < simpl in Hp.
1 subgoal

  m : nat
  IHm : forall l : list, length l <= m -> palindrome l -> palindrome_i l
  n, n0 : nat
  l : list
  Hlen : length (H n (H n0 l)) <= S m
  Hp : H n (H n0 l) = concat (concat (reverse l) (H n0 nil)) (H n nil)
  ============================
  palindrome_i (H n (H n0 l))

palindrome_equiv_def_2_strong < rewrite (reverse_single n0) in Hp.
1 subgoal

  m : nat
  IHm : forall l : list, length l <= m -> palindrome l -> palindrome_i l
  n, n0 : nat
  l : list
  Hlen : length (H n (H n0 l)) <= S m
  Hp : H n (H n0 l) =
       concat (concat (reverse l) (reverse (H n0 nil))) (H n nil)
  ============================
  palindrome_i (H n (H n0 l))

palindrome_equiv_def_2_strong < rewrite <- reverse_concat in Hp.
1 subgoal

  m : nat
  IHm : forall l : list, length l <= m -> palindrome l -> palindrome_i l
  n, n0 : nat
  l : list
  Hlen : length (H n (H n0 l)) <= S m
  Hp : H n (H n0 l) = concat (reverse (concat (H n0 nil) l)) (H n nil)
  ============================
  palindrome_i (H n (H n0 l))

palindrome_equiv_def_2_strong < simpl (concat (H n0 nil) l) in Hp.
1 subgoal

  m : nat
  IHm : forall l : list, length l <= m -> palindrome l -> palindrome_i l
  n, n0 : nat
  l : list
  Hlen : length (H n (H n0 l)) <= S m
  Hp : H n (H n0 l) = concat (reverse (H n0 l)) (H n nil)
  ============================
  palindrome_i (H n (H n0 l))
```

Now we want to pull an element from the back end. We do this by destructing the reversed list from our hypothesis.

```
palindrome_equiv_def_2_strong < remember (reverse (H n0 l)).
1 subgoal

  m : nat
  IHm : forall l : list, length l <= m -> palindrome l -> palindrome_i l
  n, n0 : nat
  l : list
  Hlen : length (H n (H n0 l)) <= S m
  l0 : list
  Heql0 : l0 = reverse (H n0 l)
  Hp : H n (H n0 l) = concat l0 (H n nil)
  ============================
  palindrome_i (H n (H n0 l))

palindrome_equiv_def_2_strong < destruct l0.
2 subgoals

  m : nat
  IHm : forall l : list, length l <= m -> palindrome l -> palindrome_i l
  n, n0 : nat
  l : list
  Hlen : length (H n (H n0 l)) <= S m
  Heql0 : nil = reverse (H n0 l)
  Hp : H n (H n0 l) = concat nil (H n nil)
  ============================
  palindrome_i (H n (H n0 l))

subgoal 2 is:
 palindrome_i (H n (H n0 l))

palindrome_equiv_def_2_strong < simpl in Hp.
2 subgoals

  m : nat
  IHm : forall l : list, length l <= m -> palindrome l -> palindrome_i l
  n, n0 : nat
  l : list
  Hlen : length (H n (H n0 l)) <= S m
  Heql0 : nil = reverse (H n0 l)
  Hp : H n (H n0 l) = H n nil
  ============================
  palindrome_i (H n (H n0 l))

subgoal 2 is:
 palindrome_i (H n (H n0 l))

palindrome_equiv_def_2_strong < discriminate.
1 subgoal

  m : nat
  IHm : forall l : list, length l <= m -> palindrome l -> palindrome_i l
  n, n0 : nat
  l : list
  Hlen : length (H n (H n0 l)) <= S m
  n1 : nat
  l0 : list
  Heql0 : H n1 l0 = reverse (H n0 l)
  Hp : H n (H n0 l) = concat (H n1 l0) (H n nil)
  ============================
  palindrome_i (H n (H n0 l))
```

Now we use the fact that `H` is injective to show that n and n_0 have to be the same, which then shows that we have a palindrome if l_0 is a palindrome.

```
palindrome_equiv_def_2_strong < injection Hp.
1 subgoal

  m : nat
  IHm : forall l : list, length l <= m -> palindrome l -> palindrome_i l
  n, n0 : nat
  l : list
  Hlen : length (H n (H n0 l)) <= S m
  n1 : nat
  l0 : list
  Heql0 : H n1 l0 = reverse (H n0 l)
  Hp : H n (H n0 l) = concat (H n1 l0) (H n nil)
  ============================
  H n0 l = concat l0 (H n nil) -> n = n1 -> palindrome_i (H n (H n0 l))
```

```
palindrome_equiv_def_2_strong < intros.
1 subgoal

  m : nat
  IHm : forall l : list, length l <= m -> palindrome l -> palindrome_i l
  n, n0 : nat
  l : list
  Hlen : length (H n (H n0 l)) <= S m
  n1 : nat
  l0 : list
  Heql0 : H n1 l0 = reverse (H n0 l)
  Hp : H n (H n0 l) = concat (H n1 l0) (H n nil)
  H0 : H n0 l = concat l0 (H n nil)
  H1 : n = n1
  ============================
  palindrome_i (H n (H n0 l))

palindrome_equiv_def_2_strong < rewrite <- H1 in Hp.
1 subgoal

  m : nat
  IHm : forall l : list, length l <= m -> palindrome l -> palindrome_i l
  n, n0 : nat
  l : list
  Hlen : length (H n (H n0 l)) <= S m
  n1 : nat
  l0 : list
  Heql0 : H n1 l0 = reverse (H n0 l)
  Hp : H n (H n0 l) = concat (H n l0) (H n nil)
  H0 : H n0 l = concat l0 (H n nil)
  H1 : n = n1
  ============================
  palindrome_i (H n (H n0 l))

palindrome_equiv_def_2_strong < rewrite Hp.
1 subgoal

  m : nat
  IHm : forall l : list, length l <= m -> palindrome l -> palindrome_i l
  n, n0 : nat
  l : list
  Hlen : length (H n (H n0 l)) <= S m
  n1 : nat
  l0 : list
  Heql0 : H n1 l0 = reverse (H n0 l)
  Hp : H n (H n0 l) = concat (H n l0) (H n nil)
  H0 : H n0 l = concat l0 (H n nil)
  H1 : n = n1
  ============================
  palindrome_i (concat (H n l0) (H n nil))

palindrome_equiv_def_2_strong < constructor.
1 subgoal

  m : nat
  IHm : forall l : list, length l <= m -> palindrome l -> palindrome_i l
  n, n0 : nat
  l : list
  Hlen : length (H n (H n0 l)) <= S m
  n1 : nat
  l0 : list
  Heql0 : H n1 l0 = reverse (H n0 l)
  Hp : H n (H n0 l) = concat (H n l0) (H n nil)
  H0 : H n0 l = concat l0 (H n nil)
  H1 : n = n1
  ============================
  palindrome_i l0
```

Finally we can use our induction hypothesis

The first thing we have to prove is the the length of our new list is less than m. We do this by first using the fact that equal strings have equal lengths in the definition of our new list. Then we use transitivity along with our induction hypothesis about the length of our list.

```
palindrome_equiv_def_2_strong < apply length_equal in Heql0.
2 subgoals

  m : nat
  IHm : forall l : list, length l <= m -> palindrome l -> palindrome_i l
  n, n0 : nat
  l : list
  Hlen : length (H n (H n0 l)) <= S m
  n1 : nat
  l0 : list
  Heql0 : length (H n1 l0) = length (reverse (H n0 l))
  Hp : H n (H n0 l) = concat (H n l0) (H n nil)
  H0 : H n0 l = concat l0 (H n nil)
  H1 : n = n1
  ============================
  length l0 <= m

subgoal 2 is:
 palindrome l0

palindrome_equiv_def_2_strong < rewrite <- reverse_length in Heql0.
2 subgoals

  m : nat
  IHm : forall l : list, length l <= m -> palindrome l -> palindrome_i l
  n, n0 : nat
  l : list
  Hlen : length (H n (H n0 l)) <= S m
  n1 : nat
  l0 : list
  Heql0 : length (H n1 l0) = length (H n0 l)
  Hp : H n (H n0 l) = concat (H n l0) (H n nil)
  H0 : H n0 l = concat l0 (H n nil)
  H1 : n = n1
  ============================
  length l0 <= m

subgoal 2 is:
 palindrome l0

palindrome_equiv_def_2_strong < simpl in Heql0.
2 subgoals

  m : nat
  IHm : forall l : list, length l <= m -> palindrome l -> palindrome_i l
  n, n0 : nat
  l : list
  Hlen : length (H n (H n0 l)) <= S m
  n1 : nat
  l0 : list
  Heql0 : S (length l0) = S (length l)
  Hp : H n (H n0 l) = concat (H n l0) (H n nil)
  H0 : H n0 l = concat l0 (H n nil)
  H1 : n = n1
  ============================
  length l0 <= m

subgoal 2 is:
 palindrome l0
```

```
palindrome_equiv_def_2_strong < simpl in Hlen.
2 subgoals

  m : nat
  IHm : forall l : list, length l <= m -> palindrome l -> palindrome_i l
  n, n0 : nat
  l : list
  Hlen : S (S (length l)) <= S m
  n1 : nat
  l0 : list
  Heql0 : S (length l0) = S (length l)
  Hp : H n (H n0 l) = concat (H n l0) (H n nil)
  H0 : H n0 l = concat l0 (H n nil)
  H1 : n = n1
  ============================
  length l0 <= m

subgoal 2 is:
 palindrome l0

palindrome_equiv_def_2_strong < rewrite <- Heql0 in Hlen.
2 subgoals

  m : nat
  IHm : forall l : list, length l <= m -> palindrome l -> palindrome_i l
  n, n0 : nat
  l, l0 : list
  Hlen : S (S (length l0)) <= S m
  n1 : nat
  Heql0 : S (length l0) = S (length l)
  Hp : H n (H n0 l) = concat (H n l0) (H n nil)
  H0 : H n0 l = concat l0 (H n nil)
  H1 : n = n1
  ============================
  length l0 <= m

subgoal 2 is:
 palindrome l0

palindrome_equiv_def_2_strong < apply le_S_n in Hlen.
2 subgoals

  m : nat
  IHm : forall l : list, length l <= m -> palindrome l -> palindrome_i l
  n, n0 : nat
  l, l0 : list
  Hlen : S (length l0) <= m
  n1 : nat
  Heql0 : S (length l0) = S (length l)
  Hp : H n (H n0 l) = concat (H n l0) (H n nil)
  H0 : H n0 l = concat l0 (H n nil)
  H1 : n = n1
  ============================
  length l0 <= m

subgoal 2 is:
 palindrome l0

palindrome_equiv_def_2_strong < apply (Nat.le_trans (length l0) (S (length l0)) m).
3 subgoals

  m : nat
  IHm : forall l : list, length l <= m -> palindrome l -> palindrome_i l
  n, n0 : nat
  l, l0 : list
  Hlen : S (length l0) <= m
  n1 : nat
  Heql0 : S (length l0) = S (length l)
  Hp : H n (H n0 l) = concat (H n l0) (H n nil)
  H0 : H n0 l = concat l0 (H n nil)
  H1 : n = n1
  ============================
  length l0 <= S (length l0)

subgoal 2 is:
 S (length l0) <= m
subgoal 3 is:
 palindrome l0

palindrome_equiv_def_2_strong < constructor.
3 subgoals

  m : nat
  IHm : forall l : list, length l <= m -> palindrome l -> palindrome_i l
  n, n0 : nat
  l, l0 : list
  Hlen : S (length l0) <= m
  n1 : nat
  Heql0 : S (length l0) = S (length l)
  Hp : H n (H n0 l) = concat (H n l0) (H n nil)
  H0 : H n0 l = concat l0 (H n nil)
  H1 : n = n1
  ============================
  length l0 <= length l0

subgoal 2 is:
 S (length l0) <= m
subgoal 3 is:
 palindrome l0

palindrome_equiv_def_2_strong < constructor.
2 subgoals

  m : nat
  IHm : forall l : list, length l <= m -> palindrome l -> palindrome_i l
  n, n0 : nat
  l, l0 : list
  Hlen : S (length l0) <= m
  n1 : nat
  Heql0 : S (length l0) = S (length l)
  Hp : H n (H n0 l) = concat (H n l0) (H n nil)
  H0 : H n0 l = concat l0 (H n nil)
  H1 : n = n1
  ============================
  S (length l0) <= m

subgoal 2 is:
 palindrome l0

palindrome_equiv_def_2_strong < exact Hlen.
1 subgoal

  m : nat
  IHm : forall l : list, length l <= m -> palindrome l -> palindrome_i l
  n, n0 : nat
  l : list
  Hlen : length (H n (H n0 l)) <= S m
  n1 : nat
  l0 : list
  Heql0 : H n1 l0 = reverse (H n0 l)
  Hp : H n (H n0 l) = concat (H n l0) (H n nil)
  H0 : H n0 l = concat l0 (H n nil)
  H1 : n = n1
  ============================
  palindrome l0
```

Now we have to prove that our new list is a palindrome. We start by flipping the reversal to the other side of our definition. This allows us to rewrite the injectivity requirement for our new list, and finally use the injectivity of concatenation.

```
palindrome_equiv_def_2_strong < apply reverse_left_right in Heql0.
1 subgoal

  m : nat
  IHm : forall l : list, length l <= m -> palindrome l -> palindrome_i l
  n, n0 : nat
  l : list
  Hlen : length (H n (H n0 l)) <= S m
  n1 : nat
  l0 : list
  Heql0 : reverse (H n1 l0) = H n0 l
  Hp : H n (H n0 l) = concat (H n l0) (H n nil)
  H0 : H n0 l = concat l0 (H n nil)
  H1 : n = n1
  ============================
  palindrome l0

palindrome_equiv_def_2_strong < rewrite <- Heql0 in H0.
1 subgoal

  m : nat
  IHm : forall l : list, length l <= m -> palindrome l -> palindrome_i l
  n, n0 : nat
  l : list
  Hlen : length (H n (H n0 l)) <= S m
  n1 : nat
  l0 : list
  Heql0 : reverse (H n1 l0) = H n0 l
  Hp : H n (H n0 l) = concat (H n l0) (H n nil)
  H0 : reverse (H n1 l0) = concat l0 (H n nil)
  H1 : n = n1
  ============================
  palindrome l0

palindrome_equiv_def_2_strong < simpl in H0.
1 subgoal

  m : nat
  IHm : forall l : list, length l <= m -> palindrome l -> palindrome_i l
  n, n0 : nat
  l : list
  Hlen : length (H n (H n0 l)) <= S m
  n1 : nat
  l0 : list
  Heql0 : reverse (H n1 l0) = H n0 l
  Hp : H n (H n0 l) = concat (H n l0) (H n nil)
  H0 : concat (reverse l0) (H n1 nil) = concat l0 (H n nil)
  H1 : n = n1
  ============================
  palindrome l0

palindrome_equiv_def_2_strong < rewrite H1 in H0.
1 subgoal

  m : nat
  IHm : forall l : list, length l <= m -> palindrome l -> palindrome_i l
  n, n0 : nat
  l : list
  Hlen : length (H n (H n0 l)) <= S m
  n1 : nat
  l0 : list
  Heql0 : reverse (H n1 l0) = H n0 l
  Hp : H n (H n0 l) = concat (H n l0) (H n nil)
  H0 : concat (reverse l0) (H n1 nil) = concat l0 (H n1 nil)
  H1 : n = n1
  ============================
  palindrome l0

palindrome_equiv_def_2_strong < apply concat_injective_left in H0.
1 subgoal

  m : nat
  IHm : forall l : list, length l <= m -> palindrome l -> palindrome_i l
  n, n0 : nat
  l : list
  Hlen : length (H n (H n0 l)) <= S m
  n1 : nat
  l0 : list
  Heql0 : reverse (H n1 l0) = H n0 l
  Hp : H n (H n0 l) = concat (H n l0) (H n nil)
  H0 : reverse l0 = l0
  H1 : n = n1
  ============================
  palindrome l0

palindrome_equiv_def_2_strong < symmetry in H0.
1 subgoal

  m : nat
  IHm : forall l : list, length l <= m -> palindrome l -> palindrome_i l
  n, n0 : nat
  l : list
  Hlen : length (H n (H n0 l)) <= S m
  n1 : nat
  l0 : list
  Heql0 : reverse (H n1 l0) = H n0 l
  Hp : H n (H n0 l) = concat (H n l0) (H n nil)
  H0 : l0 = reverse l0
  H1 : n = n1
  ============================
  palindrome l0

palindrome_equiv_def_2_strong < exact H0.
No more subgoals.

palindrome_equiv_def_2_strong < Qed.
(induction m).
 (intros l Hlen Hp).
 (destruct l).
  constructor.

  (inversion Hlen).

 (intros l Hlen Hp).
 (destruct l).
  constructor.

  (destruct l).
   constructor.

   (unfold palindrome in Hp).
   (simpl in Hp).
   (rewrite (reverse_single n0) in Hp).
   (rewrite <- reverse_concat in Hp).
   (simpl (concat (H n0 nil) l) in Hp).
   (remember (reverse (H n0 l))).
   (destruct l0).
    (simpl in Hp).
    discriminate.
    injection Hp.
    (intros **).
    (rewrite <- H1 in Hp).
    (rewrite Hp).
    constructor.
    (apply IHm).
     (apply length_equal in Heql0).
     (rewrite <- reverse_length in Heql0).
     (simpl in Heql0).
     (simpl in Hlen).
     (rewrite <- Heql0 in Hlen).
     (apply le_S_n in Hlen).
     (apply (Nat.le_trans (length l0) (S (length l0)) m)).
      constructor.
      constructor.

      exact Hlen.

     (apply reverse_left_right in Heql0).
     (rewrite <- Heql0 in H0).
     (simpl in H0).
     (rewrite H1 in H0).
     (apply concat_injective_left in H0).
     symmetry in H0.
     exact H0.

Qed.
palindrome_equiv_def_2_strong is defined
```

Now we can prove our original statement,

```
Coq < Theorem palindrome_equiv_def_2 : forall l : list, palindrome l -> palindrome_i l.
1 subgoal

  ============================
  forall l : list, palindrome l -> palindrome_i l

palindrome_equiv_def_2 < intros l H.
1 subgoal

  l : list
  H : palindrome l
  ============================
  palindrome_i l

palindrome_equiv_def_2 < apply (palindrome_equiv_def_2_strong (length l)).
2 subgoals

  l : list
  H : palindrome l
  ============================
  length l <= length l

subgoal 2 is:
 palindrome l

palindrome_equiv_def_2 < apply le_n.
1 subgoal

  l : list
  H : palindrome l
  ============================
  palindrome l

palindrome_equiv_def_2 < exact H.
No more subgoals.

palindrome_equiv_def_2 < Qed.
```

```
(intros l H).
(apply (palindrome_equiv_def_2_strong (length l))).
 (apply le_n).

 exact H.

Qed.
palindrome_equiv_def_2 is defined
```

and complete equivalance

```
Coq < Theorem palindrome_equiv_def : forall l : list, palindrome_i l <-> palindrome l.
1 subgoal

  ============================
  forall l : list, palindrome_i l <-> palindrome l

palindrome_equiv_def < split.
2 subgoals

  l : list
  ============================
  palindrome_i l -> palindrome l

subgoal 2 is:
 palindrome l -> palindrome_i l

palindrome_equiv_def < apply palindrome_equiv_def_1.
1 subgoal

  l : list
  ============================
  palindrome l -> palindrome_i l

palindrome_equiv_def < apply palindrome_equiv_def_2.
No more subgoals.

palindrome_equiv_def < Qed.
split.
 (apply palindrome_equiv_def_1).

 (apply palindrome_equiv_def_2).

Qed.
palindrome_equiv_def is defined
```

This is a good discussion of the Pigeonhole Principle using lists.

4.6 Problems

Problem IV.1 Prove by contradiction that there are an infinite number of primes. A *prime* number is one that is only divisible by itself and one. A composite number, one that is not prime, must be divisible by a prime number. You might also use the fact that the number $ab+1$, for natural numbers a and b, is not divisible by a or b since the remainder is one in both cases.

Problem IV.2 Use the Well Ordering Principle to prove that any fraction m/n can be written in lowest terms. A fraction is in lowest terms if there is no natural number k which divides both m and n. It is a good idea to define the set of counterexamples based upon the numerator, namely

$$C := \{m \in \mathbb{N} \mid \exists n \in \mathbb{N}^+, \frac{m}{n} \text{cannot be written in lowest terms}\}. \tag{4.71}$$

This proof follows the format of Well Ordering proofs in the notes and is not constructive.

Problem IV.3 The Fibonacci numbers $F(k)$ are defined as follows,

$$F(0) = 0 \tag{4.72}$$
$$F(1) = 1 \tag{4.73}$$
$$F(n) = F(n-1) + F(n-2) \qquad \text{for} \quad n \geq 2 \tag{4.74}$$

Thus, the Fibonacci numbers are $0, 1, 1, 2, 3, 5, 8, 13, 21, \ldots$. Prove by strong induction that for all $n \geq 1$,

$$F(n+1)F(n-1) - F(n)^2 = (-1)^n. \tag{4.75}$$

Problem IV.4 Using strong induction, prove that every positive integer can be written as the sum of one or more Fibonacci numbers.

Problem IV.5 Use induction to prove that

$$\sum_{i=0}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}. \tag{4.76}$$

by generating **Proof:** *sum_square_p*

```
Lemma sum_square_p : forall n, 6 * sum_n2 n = n * (n + 1) * (2 * n + 1).
```

where you define `sum_n2` using the `Fixpoint` operator in Coq as

```
Fixpoint sum_n2 n :=
match n with
  O => 0
| S p => n*n + sum_n2 p
end.
```

Problem IV.6 Use induction to prove that

$$\sum_{i=0}^{n} i^3 = \left(\sum_{i=0}^{n} i\right)^2. \tag{4.77}$$

by generating **Proof:** *sum_cube_p*

```
Lemma sum_cube_p : forall n, sum_n3 n = (sum_n n)*(sum_n n).
```

where you define `sum_n3` using the `Fixpoint` operator in Coq as

```
Fixpoint sum_n3 n :=
match n with
  0 => 0
| S p => p*p*p + sum_n3 p
end.
```

Note that I used the `sum_n_p` proof for the sum of the first n numbers from the text, and the definition

```
Fixpoint sum_n n :=
  match n with
      0 => 0
  | S p => p + sum_n p
end.
```

Problem IV.7 Prove that the sum of the first n odd natural numbers is n^2 by generating **Proof:** *odd_sum*

```
  Lemma odd_sum : forall n:nat, sum_odd_n n = n*n.
```

where you define `sum_odd_n` using the `Fixpoint` operator in Coq as

```
Fixpoint sum_odd_n (n:nat) : nat :=
match n with
  0 => 0
| S p => 1 + 2 * p + sum_odd_n p
end.
```

Problem IV.8 We would like to prove the following statement by contraposition,

For all natural numbers x and y, if $x + y$ is odd, then x is odd or y is odd.

1. Translate the statement into a statement of predicate logic.
2. Provide the antecedent required for a proof by contraposition for the given statement.
3. Provide the consequent for a proof by contraposition for the given statement.
4. Prove the contrapositive statement is true, from which you can conclude that the original statement is true. You may use either Coq or the informal proof shown in the text.

Problem IV.9 We would like to prove the following statement by contradiction,

For all natural numbers n, if a^n is even, then a is even.

1. Translate the statement into a statement of predicate logic.
2. Provide the assumption required for a proof by contradiction for the given statement.
3. Prove the statement is true by contradiction.

Problem IV.10 Prove that any number greater than seven can be written as a sum of multiples of three and five. The Coq statement is

```
Theorem three_and_five : forall n : nat, exists (a b : nat), n + 8 = 3 * a + 5 * b.
```

You should use the strong induction tactic, `strong_nat_ind`, as shown in the text.

Prove reversal is bijective
Use inversion instead of injection on p.237
Introduction to Functional Programming
Technical details in Coq Boxes
Move truth tables to Classical Logic Appendix
Move all by-hand proofs to one chapter and appendix

References

Lehman, Eric, F Thomson Leighton, and Albert R Meyer (2015). *Mathematics for Computer Science*. Tech. rep. Lecture Notes. Massachussetts Institute of Technology.

Graham, Ronald L, Donald E Knuth, and Oren Patashnik (1989). *Concrete Mathematics*. Addison-Wesley.

Chapter 5

Modular Arithmetic

5.1 Integers

We would like to define integers inductively, just as we defined the natural numbers. Suppose that we introduce a P operation, indicating predecessor, which subtracts one from the input number. We could try to define integers as

```
  Inductive integer : Set :=
  O : integer
| S : integer -> integer
| P : integer -> integer
```

However, this is problematic because we would not have a unique definition of each number. For example, the number 3 could be represented as

$$(S(S(SO))) = (S(S(S(S(PO)))))$$

and an infinite number of other ways. It seems that we should have separate ways to handle positive and negative integers, and this is, in fact, how Coq manages things.

```
Coq < Print Z.
Inductive Z : Set :=
  Z0 : Z
| Zpos : positive -> Z
| Zneg : positive -> Z
```

Coq uses Z to refer to integers, often written $\mathbb{Z}$, standing for the German word Zahlen, meaning "numbers". We can construct an integer either as zero, from a positive natural number n giving the integer n, and from a positive natural number n giving the integer $-n$. Thus we have only one way of constructing every integer if we have only one way of making every positive natural number. We could define the positive numbers in the same way as natural numbers

```
  Inductive positive : Set :=
  I : positive
| S : positive -> positive
```

However, Coq does not do this, in order to simplify some proofs later. Instead it uses the following definition

```
Inductive positive : Set :=
  xI : positive -> positive
| xO : positive -> positive
| xH : positive
```

Here xH corresponds to the number one, or I in our simple definition, the base case for the inductive definition. The increment operation S has been replaced by two operations xO and xI, which correspond to multiplication by two and adding one after doubling. An example should make this scheme clearer. Suppose we want to express the number 5,

```
  5 = (xI (xO xH))
```

which we can see as the reverse binary representation of the integer, with xI and xH being one and xO being zero,

```
  5 = 101.
```

We can see that this reverses the bits by looking at 11,

```
  11 = (xI (xI (xO xH))) = 1011.
```

We can briefly look at the induction theorems for these types. For the positive numbers we have

```
Coq < Print positive_ind.
positive_ind =
fun P : positive -> Prop => positive_rect P
     : forall P : positive -> Prop,
       (forall p : positive, P p -> P (p~1)%positive) ->
       (forall p : positive, P p -> P (p~0)%positive) ->
       P 1%positive -> forall p : positive, P p
```

Thus, in order to prove that some predicate P is true on all positive numbers, we first prove it for the base case 1. Notice here that Coq gives the type explicitly using `1%positive` in order to distinguish it from 1 the natural number or 1 the integer. After the base case, we have two induction steps, one for even numbers and one for odd. The even numbers are those constructed using `(xO p)` so that they have a 0 in the least-significant bit, and odd numbers are `(xI p)` with a trailing 1. Instead of using xO and xI, this definition uses an infix notation that is equivalent

```
  (xO p) = p~0
  (xI p) = p~1
```

so that the tilde operator appends either a 0 or a 1 bit to the end of the number p. The integer induction theorem is different, in that it has no induction step.

```
Coq < Print Z_ind.
Z_ind =
fun P : Z -> Prop => Z_rect P
     : forall P : Z -> Prop,
       P 0%Z ->
       (forall p : positive, P (Z.pos p)) ->
       (forall p : positive, P (Z.neg p)) -> forall z : Z, P z
```

We first prove that P is true for integer 0, then prove that P holds for all positive number mapped to positive integers, and finally that P holds for all positive numbers mapped to negative integers. In order to prove the latter two statements, we would likely use the induction principle for positive numbers above.

To illustrate induction on the integers, consider first a simple theorem stating that if the product of two integers is zero, and one of them is not zero, then the other one must be zero.

```
Coq < Check Zmult_integral_l.
Zmult_integral_l
     : forall n m : Z, n <> 0 -> m * n = 0 -> m = 0
```

We can use this to prove a slightly different theorem that immediately occurs to most of us, namely that one or the other of these integers must be zero. Since it is a universal statetment, we will try to prove this using induction on n.

```
Coq < Lemma mult_integral : forall n m : Z, n*m = 0 -> n = 0 \/ m = 0.
1 subgoal

  ============================
  forall n m : Z, n * m = 0 -> n = 0 \/ m = 0

mult_integral < intros n m.
1 subgoal

  n, m : Z
  ============================
  n * m = 0 -> n = 0 \/ m = 0

mult_integral < induction n.
3 subgoals

  m : Z
  ============================
  0 * m = 0 -> 0 = 0 \/ m = 0

subgoal 2 is:
 Z.pos p * m = 0 -> Z.pos p = 0 \/ m = 0
subgoal 3 is:
 Z.neg p * m = 0 -> Z.neg p = 0 \/ m = 0
```

Notice that we have a single base case, but two separate induction steps. The base case is easily handled by choosing the left branch of the disjunction.

```
mult_integral < intro H.
3 subgoals

  m : Z
  H : 0 * m = 0
  ============================
  0 = 0 \/ m = 0

subgoal 2 is:
 Z.pos p * m = 0 -> Z.pos p = 0 \/ m = 0
subgoal 3 is:
 Z.neg p * m = 0 -> Z.neg p = 0 \/ m = 0

mult_integral < left.
3 subgoals

  m : Z
  H : 0 * m = 0
  ============================
  0 = 0

subgoal 2 is:
 Z.pos p * m = 0 -> Z.pos p = 0 \/ m = 0
```

```
subgoal 3 is:
 Z.neg p * m = 0 -> Z.neg p = 0 \/ m = 0

mult_integral < reflexivity.
2 subgoals

  p : positive
  m : Z
  ============================
  Z.pos p * m = 0 -> Z.pos p = 0 \/ m = 0

subgoal 2 is:
 Z.neg p * m = 0 -> Z.neg p = 0 \/ m = 0
```

Now we must prove the statement for all positive numbers p. For this we again use induction, but on the number p this time.

```
mult_integral < intro H.
2 subgoals

  p : positive
  m : Z
  H : Z.pos p * m = 0
  ============================
  Z.pos p = 0 \/ m = 0

subgoal 2 is:
 Z.neg p * m = 0 -> Z.neg p = 0 \/ m = 0

mult_integral < induction p.
4 subgoals

  p : positive
  m : Z
  H : Z.pos p~1 * m = 0
  IHp : Z.pos p * m = 0 -> Z.pos p = 0 \/ m = 0
  ============================
  Z.pos p~1 = 0 \/ m = 0

subgoal 2 is:
 Z.pos p~0 = 0 \/ m = 0
subgoal 3 is:
 1 = 0 \/ m = 0
subgoal 4 is:
 Z.neg p * m = 0 -> Z.neg p = 0 \/ m = 0
```

We again have three cases, this time odd numbers, even numbers, and the base case, which is put last. Clearly the left branch of our goal is false since the number p is positive. Thus, we must choose the right branch. In addition, we can use the original theorem above, since we know that p is not zero, to prove that m is indeed zero.

```
mult_integral < right.
4 subgoals

  p : positive
  m : Z
  H : Z.pos p~1 * m = 0
  IHp : Z.pos p * m = 0 -> Z.pos p = 0 \/ m = 0
  ============================
  m = 0

subgoal 2 is:
 Z.pos p~0 = 0 \/ m = 0
subgoal 3 is:
 1 = 0 \/ m = 0
```

```
subgoal 4 is:
 Z.neg p * m = 0 -> Z.neg p = 0 \/ m = 0

mult_integral < rewrite Z.mul_comm in H.
4 subgoals

  p : positive
  m : Z
  H : m * Z.pos p~1 = 0
  IHp : Z.pos p * m = 0 -> Z.pos p = 0 \/ m = 0
  ============================
  m = 0

subgoal 2 is:
 Z.pos p~0 = 0 \/ m = 0
subgoal 3 is:
 1 = 0 \/ m = 0
subgoal 4 is:
 Z.neg p * m = 0 -> Z.neg p = 0 \/ m = 0

mult_integral < apply Zmult_integral_l in H.
5 subgoals

  p : positive
  m : Z
  H : m = 0
  IHp : Z.pos p * m = 0 -> Z.pos p = 0 \/ m = 0
  ============================
  m = 0

subgoal 2 is:
 Z.pos p~1 <> 0
subgoal 3 is:
 Z.pos p~0 = 0 \/ m = 0
subgoal 4 is:
 1 = 0 \/ m = 0
subgoal 5 is:
 Z.neg p * m = 0 -> Z.neg p = 0 \/ m = 0

mult_integral < exact H.
4 subgoals

  p : positive
  m : Z
  H : m * Z.pos p~1 = 0
  IHp : Z.pos p * m = 0 -> Z.pos p = 0 \/ m = 0
  ============================
  Z.pos p~1 <> 0

subgoal 2 is:
 Z.pos p~0 = 0 \/ m = 0
subgoal 3 is:
 1 = 0 \/ m = 0
subgoal 4 is:
 Z.neg p * m = 0 -> Z.neg p = 0 \/ m = 0

mult_integral < intro Absurd.
4 subgoals

  p : positive
  m : Z
  H : m * Z.pos p~1 = 0
  IHp : Z.pos p * m = 0 -> Z.pos p = 0 \/ m = 0
  Absurd : Z.pos p~1 = 0
  ============================
  False

subgoal 2 is:
 Z.pos p~0 = 0 \/ m = 0
subgoal 3 is:
 1 = 0 \/ m = 0
subgoal 4 is:
 Z.neg p * m = 0 -> Z.neg p = 0 \/ m = 0

mult_integral < discriminate.
3 subgoals

  p : positive
  m : Z
  H : Z.pos p~0 * m = 0
  IHp : Z.pos p * m = 0 -> Z.pos p = 0 \/ m = 0
  ============================
  Z.pos p~0 = 0 \/ m = 0

subgoal 2 is:
 1 = 0 \/ m = 0
subgoal 3 is:
 Z.neg p * m = 0 -> Z.neg p = 0 \/ m = 0
```

Now we use the same strategy to prove the result for all even numbers.

```
mult_integral < right.
3 subgoals

  p : positive
  m : Z
  H : Z.pos p~0 * m = 0
  IHp : Z.pos p * m = 0 -> Z.pos p = 0 \/ m = 0
  ============================
  m = 0

subgoal 2 is:
 1 = 0 \/ m = 0
subgoal 3 is:
 Z.neg p * m = 0 -> Z.neg p = 0 \/ m = 0

mult_integral < rewrite Z.mul_comm in H.
3 subgoals

  p : positive
  m : Z
  H : m * Z.pos p~0 = 0
  IHp : Z.pos p * m = 0 -> Z.pos p = 0 \/ m = 0
  ============================
  m = 0

subgoal 2 is:
 1 = 0 \/ m = 0
subgoal 3 is:
 Z.neg p * m = 0 -> Z.neg p = 0 \/ m = 0

mult_integral < apply Zmult_integral_l in H.
4 subgoals

  p : positive
  m : Z
  H : m = 0
  IHp : Z.pos p * m = 0 -> Z.pos p = 0 \/ m = 0
  ============================
  m = 0

subgoal 2 is:
 Z.pos p~0 <> 0
subgoal 3 is:
 1 = 0 \/ m = 0
subgoal 4 is:
 Z.neg p * m = 0 -> Z.neg p = 0 \/ m = 0

mult_integral < exact H.
3 subgoals

  p : positive
  m : Z
  H : m * Z.pos p~0 = 0
  IHp : Z.pos p * m = 0 -> Z.pos p = 0 \/ m = 0
  ============================
  Z.pos p~0 <> 0

subgoal 2 is:
 1 = 0 \/ m = 0
subgoal 3 is:
 Z.neg p * m = 0 -> Z.neg p = 0 \/ m = 0

mult_integral < intro Absurd.
3 subgoals

  p : positive
  m : Z
  H : m * Z.pos p~0 = 0
  IHp : Z.pos p * m = 0 -> Z.pos p = 0 \/ m = 0
  Absurd : Z.pos p~0 = 0
  ============================
  False

subgoal 2 is:
 1 = 0 \/ m = 0
subgoal 3 is:
 Z.neg p * m = 0 -> Z.neg p = 0 \/ m = 0

mult_integral < discriminate.
2 subgoals

  m : Z
  H : 1 * m = 0
  ============================
  1 = 0 \/ m = 0

subgoal 2 is:
 Z.neg p * m = 0 -> Z.neg p = 0 \/ m = 0
```

Now we prove the base case for positive integers, namely $n = 1$,

```
mult_integral < right.
2 subgoals

  m : Z
  H : 1 * m = 0
  ============================
  m = 0

subgoal 2 is:
 Z.neg p * m = 0 -> Z.neg p = 0 \/ m = 0

mult_integral < rewrite <- H.
2 subgoals

  m : Z
  H : 1 * m = 0
  ============================
  m = 1 * m

subgoal 2 is:
 Z.neg p * m = 0 -> Z.neg p = 0 \/ m = 0
```

```
mult_integral < ring.
1 subgoal

  p : positive
  m : Z
  ============================
  Z.neg p * m = 0 -> Z.neg p = 0 \/ m = 0
```

Finally, the same procedure used above to prove the statement for all positive integers can again be used to prove the result for all negative integers,

```
mult_integral < intro H.
1 subgoal

  p : positive
  m : Z
  H : Z.neg p * m = 0
  ============================
  Z.neg p = 0 \/ m = 0

mult_integral < right.
1 subgoal

  p : positive
  m : Z
  H : Z.neg p * m = 0
  ============================
  m = 0

mult_integral < rewrite Z.mul_comm in H.
1 subgoal

  p : positive
  m : Z
  H : m * Z.neg p = 0
  ============================
  m = 0

mult_integral < apply Zmult_integral_l in H.
2 subgoals

  p : positive
  m : Z
  H : m = 0
  ============================
  m = 0

subgoal 2 is:
 Z.neg p <> 0

mult_integral < exact H.
1 subgoal

  p : positive
  m : Z
  H : m * Z.neg p = 0
  ============================
  Z.neg p <> 0

mult_integral < intro Absurd.
1 subgoal

  p : positive
  m : Z
  H : m * Z.neg p = 0
  Absurd : Z.neg p = 0
  ============================
  False
```

```
mult_integral < discriminate.
No more subgoals.

mult_integral < Qed.
(intros n m).
(induction n).
 intro H.
 left.
 reflexivity.

 intro H.
 (induction p).
  right.
  (rewrite Z.mul_comm in H).
  (apply Zmult_integral_l in H).
   exact H.

   intro Absurd.
   discriminate.

  right.
  (rewrite Z.mul_comm in H).
  (apply Zmult_integral_l in H).
   exact H.

   intro Absurd.
   discriminate.

  right.
  (rewrite <- H).
  ring.

 intro H.
 right.
 (rewrite Z.mul_comm in H).
 (apply Zmult_integral_l in H).
  exact H.

  intro Absurd.
  discriminate.

Qed.
mult_integral is defined
```

5.2 Integer Division and Modulo

Let $n \in \mathbb{Z}$ and let $m \in Z^+$. Then there are unique integers q and r, with $0 \leq r < m$, such that

$$n = qm + r. \tag{5.1}$$

We can prove this by starting with the true statement for $q = 0$ and $r = n$,

$$n = (0)m + n. \tag{5.2}$$

If $m > n$, we are done since $r = n \leq m - 1$. Thus let $m \leq n$ and add zero to the equation

$$\begin{aligned} n &= (0)m + (m - m) + n, && (5.3)\\ &= (1)m + (n - m). && (5.4) \end{aligned}$$

If $r = n - m < m$ we are done, otherwise we subtract again. We are guaranteed not to fall below zero, because initially $r = n \geq m$, and at each iteration we check that $r \geq m$, and then we subtract m. We are guaranteed to subtract a finite number of times because $\exists k \in \mathbb{N}, km > n$. Thus q and r exist. Suppose (q, r) and (q', r') exist such that the equation is true,

$$qm + r = q'm + r' \tag{5.5}$$

$$(q - q')m = (r' - r) \tag{5.6}$$

which implies that $(r' - r)$ is a multiple of m. However, $|r' - r| < m$, and thus we must have $r = r'$, which implies $q = q'$. We call q the *quotient*, denoted $\lfloor n/m \rfloor$, and r the *remainder*, denoted $n \bmod m$. In programming languages like C, quotient is represented by `n/m` and remainder, or modulo, by `n % m`. Our proof is related to the Euclidean Algorithm for computing the greatest common divisor of two integers.

Let m and n be integers. We say that m *divides* n, denoted $m \mid n$, if there is an integer k such that $n = km$, or that division of n by m leaves no remainder,

$$m \mid n \iff \exists k \in \mathbb{Z}, km = n. \tag{5.7}$$

We say that n is a multiple of m, and m is a factor or divisor of n. We see that $m \mid n$ is equivalent to saying that $r = 0$. A very useful identity is

$$\forall n, m \in \mathbb{N}, d|x \wedge d|y \implies d|nx + my. \tag{5.8}$$

which we will prove in the next section.

Having an idea of integer divisor, we can define *prime* and *composite* integers. A prime number p is a natural number greater than one whose only divisors are one and p. A composite number is a natural number greater than one which is not prime. Therefore a composite number must have a divisor different from itself and one. Two numbers a and b are said to be *coprime*, or relatively prime, if their only common divisor is one. We denote coprimality by $a \perp b$.

Let us try to prove that every natural number greater than 1 is divisible by a prime, where we can assume that all numbers are either prime or composite. The Well-Ordering Principle can be used to help. Let us call our set of counterexamples C, where C contains all natural numbers greater than 1 which are not divisible by a prime. Let us take the least number m in C. If m is prime, it is divisible by itself, so m must be composite, meaning

$$m = ab \wedge a < m \wedge b < m.$$

Since a is less than m, it cannot be a member of C, and thus is must be divisible by some prime p, $p|a$. Now we can use the theorem you will prove in Problem 1

$$\forall p, a, b \in \mathbb{Z}, p|a \vee p|b \implies p|ab.$$

This theorem shows that p must also divide m, which contradicts the definition. Thus, every natural number greater than 1 is divisible by a prime.

5.3 Modular Rings

A ring is an algebraic structure which is a generalization of arithmetic on the integers. A *ring* consists of a set equipped with two binary operations that generalize the arithmetic operations of addition and multiplication. This abstraction allows us to apply theorems from arithmetic to mathematical objects which look quite different from integers, such as function, polynomials, series, and matrices. The first binary operation, which we will always call *addition* in the ring, is associative, commutative, and has an identity element. The second binary operation, which we will always call *multiplication* in the ring, is associative, distributive over the addition operation, and has an identity element. These are the properties we associate with the normal arithmetic operations. However, they are also satisfied by a whole host of different operations.

For example, if we choose the ring addition operation to be the maximum function, and ring multiplication to be integer addition, we have another ring, often called *max-plus*. We can verify the properties of the operators. First, ring addition

Associativity	$\forall abc : \mathbb{N},$	$\max(a, \max(b, c)) = \max(\max(a, b), c)$
Commutativity	$\forall ab : \mathbb{N},$	$\max(a, b) = \max(b, a)$
Identity	$\forall a : \mathbb{N},$	$\max(a, -\infty) = a$

and we see that the identity element for ring addition is negative infinity. For ring multiplication,

Associativity	$\forall abc : \mathbb{N},$	$a + (b + c) = (a + b) + c$
Distributivity	$\forall abc : \mathbb{N},$	$a + \max(b, c) = \max(a + b, a + c)$
Identity	$\forall a : \mathbb{N},$	$a + 0 = a$

the identity element is 0. Notice that if we multiply in the ring by the additive identity, $-\infty$, we always get it back, $a - \infty = -\infty$. This is the analogue of multiplying by 0 in normal arithmetic.

It turns out that we can redefine the algebraic operations on integers to operate only on $\mathbb{Z}_n = \{0, \dots, n - 1\}$, but have the same algebraic properties. To do this, we make use of the *modulo* operator

$$a \bmod b = r \qquad \text{where} \qquad a = qb + r, \tag{5.9}$$

where q and r are defined using integer division from Section 5.2. Note that

$$a \bmod b < b \tag{5.10}$$

due to the definition of the remainder. Thus the modulo operation is *idempotent*, meaning that applying it a second time does not change the answer,

$$(a \bmod b) \bmod b = a \bmod b. \tag{5.11}$$

We have a notation for equations that hold modulo a number n,

$$(x \bmod n) = (y \bmod n) \qquad \equiv \qquad x = y \pmod{n}, \tag{5.12}$$

which we call a *congruence*, and the "x equals y modulo n". We can also see that the modulo operation can be pulled into or pushed out of addition,

$$\begin{aligned}((x \bmod n) + (y \bmod n)) \bmod n &= (x + y) \bmod n\\ (r_x + r_y) \bmod n &= (q_x n + r_x + q_y n + r_y) \bmod n\\ (r_x + r_y) \bmod n &= (r_x + r_y) \bmod n.\end{aligned}$$

We define addition in the ring to be integer addition modulo n, $x + y \pmod n$. For example, suppose $n = 7$,

$$4 + 6 \pmod 7 = 10 \pmod 7 = 3, \tag{5.13}$$
$$3 + 4 \pmod 7 = \ \ 7 \pmod 7 = 0. \tag{5.14}$$

Note that we still have the equivalent of negative numbers (additive inverses), since $3 + 4 \pmod 7 = 0$, so $-3 \pmod 7 = 4$ and $-4 \pmod 7 = 3$. We can do the same thing with multiplication,

$$4 * 6 \pmod 7 = 24 \pmod 7 = 3, \tag{5.15}$$
$$3 * 4 \pmod 7 = 12 \pmod 7 = 5, \tag{5.16}$$
$$2 * 4 \pmod 7 = \ \ 8 \pmod 7 = 1. \tag{5.17}$$

and we see that there are multiplicative inverses $1/2 \pmod 7 = 4$ and $1/4 \pmod 7 = 2$. We can remain in $\mathbb{Z}_n$ due to

$$x + y \pmod n = (x \bmod n) + (y \bmod n) \pmod n, \tag{5.18}$$
$$x \ * y \pmod n = (x \bmod n) \ * (y \bmod n) \pmod n. \tag{5.19}$$

so that the numbers we begin with can always be collapsed down to the size of our remainder set before applying the operations. Thus, since modulo addition and multiplication satisfy the properties above, $\mathbb{Z}_n$ with these operations is a ring. We choose a prime number, 7, as our modulus for the examples. This has the desirable property that nothing but 0 can multiply another number and give 0. For example, suppose $n = 6$, then we have

$$3 * 4 \pmod 6 = 12 \pmod 6 = 0. \tag{5.20}$$

5.3.1 Divisibility in Coq

We must load the `ZArith` package since we will be working with integers rather than natural numbers. It is also important to operate in the `Z_scope` since definitions for the divisibility operator | have been made, as well as some basic theorems.

```
Require Import ZArith.
Open Scope Z_scope.
```

We would like to prove the divisibility is a partial order, and thus we start with reflexivity.

```
Coq < Lemma divide_refl : forall n, (n | n).
1 subgoal

  A : Set
  ============================
  forall n : Z, (n | n)
divide_refl < intro n.
1 subgoal

  A : Set
  n : Z
  ============================
  (n | n)
```

When we unfold the definition of the divibility operator, we see that it contains an existence statement, just as we defined it above.

```
divide_refl < unfold Z.divide.
1 subgoal

  A : Set
  n : Z
  ============================
  exists z : Z, n = z * n
```

We know that the other divisor is one, so we provide it and use the `ring` tactic to prove the results.

```
divide_refl < exists 1.
1 subgoal

  A : Set
  n : Z
  ============================
  n = 1 * n

divide_refl < ring.
No more subgoals.

divide_refl < Qed.
intro n.
(unfold Z.divide).
exists 1.
ring.

Qed.
divide_refl is defined
```

Transitivity can be proved in the same manner. We first state the problem, introduce hypotheses, and unfold the definition of divisibility.

```
Coq < Lemma divide_trans : forall n m p, (n | m) -> (m | p) -> (n | p).
1 subgoal

  A : Set
  ============================
  forall n m p : Z, (n | m) -> (m | p) -> (n | p)

divide_trans < intros n m p.
1 subgoal

  A : Set
  n, m, p : Z
  ============================
  (n | m) -> (m | p) -> (n | p)

divide_trans < unfold Z.divide.
```

```
1 subgoal

  A : Set
  n, m, p : Z
  ============================
  (exists z : Z, m = z * n) ->
  (exists z : Z, p = z * m) -> exists z : Z, p = z * n
```

Then we introduce the hypotheses about divisibility, and ask for witnesses for the existence statements.

```
divide_trans < intros H1 H2.
1 subgoal

  A : Set
  n, m, p : Z
  H1 : exists z : Z, m = z * n
  H2 : exists z : Z, p = z * m
  ============================
  exists z : Z, p = z * n

divide_trans < destruct H1 as [z1 H1].
1 subgoal

  n, m, p, z1 : Z
  H1 : m = z1 * n
  H2 : exists z : Z, p = z * m
  ============================
  exists z : Z, p = z * n

divide_trans < destruct H2 as [z2 H2].
1 subgoal

  n, m, p, z1 : Z
  H1 : m = z1 * n
  z2 : Z
  H2 : p = z2 * m
  ============================
  exists z : Z, p = z * n
```

We know now that the answer is $z = z_1 z_2$,

```
divide_trans < exists (z1*z2).
1 subgoal

  n, m, p, z1 : Z
  H1 : m = z1 * n
  z2 : Z
  H2 : p = z2 * m
  ============================
  p = z1 * z2 * n
```

and we use the witnesses to rewrite the goal,

```
divide_trans < rewrite H2.
1 subgoal

  n, m, p, z1 : Z
  H1 : m = z1 * n
  z2 : Z
  H2 : p = z2 * m
  ============================
  z2 * m = z1 * z2 * n

divide_trans < rewrite H1.
1 subgoal

  n, m, p, z1 : Z
  H1 : m = z1 * n
  z2 : Z
  H2 : p = z2 * m
  ============================
  z2 * (z1 * n) = z1 * z2 * n
```

and then we can verify this using the `ring` tactic.

```
divide_trans < ring.
No more subgoals.

divide_trans < Qed.
(intros n m p).
(unfold Z.divide).
(intros H1 H2).
(destruct H1 as [z1 H1]).
(destruct H2 as [z2 H2]).
exists (z1 * z2).
(rewrite H2).
(rewrite H1).
ring.

Qed.
divide_trans is defined
```

Finally we prove that the divisibility relation is antisymmetric. However, this is only strictly true if both m and n are non-negative. If we allow either one to be negative, then we have a sign ambiguity. For example, consider m and $-m$. They both divide each other, but are not equal. Thus, in our proof we must require that m and n are non-negative. Here I require that they are positive to avoid the special case for 0, which is left for Exercise 9. We begin by introducing hypotheses and extracting witnesses.

```
Coq < Lemma divide_antisymmetric : forall m n : Z, (m > 0) -> (n > 0) -> (m | n) -> (n | m) -> (m = n).
1 subgoal

  ============================
  forall m n : Z, m > 0 -> n > 0 -> (m | n) -> (n | m) -> m = n

divide_antisymmetric < intros m n Pm Pn H1 H2.
1 subgoal

  m, n : Z
  Pm : m > 0
  Pn : n > 0
  H1 : (m | n)
  H2 : (n | m)
  ============================
  m = n

divide_antisymmetric < destruct H1 as [k H1].
1 subgoal

  m, n : Z
  Pm : m > 0
  Pn : n > 0
  k : Z
  H1 : n = k * m
  H2 : (n | m)
  ============================
  m = n

divide_antisymmetric < destruct H2 as [l H2].
1 subgoal
```

```
  m, n : Z
  Pm : m > 0
  Pn : n > 0
  k : Z
  H1 : n = k * m
  l : Z
  H2 : m = l * n
  ============================
  m = n
```

We now copy hypothesis H2 so that we can rewrite it into the identity $k*l = 1$,

```
divide_antisymmetric < pose (H3 := H2).
1 subgoal

  m, n : Z
  Pm : m > 0
  Pn : n > 0
  k : Z
  H1 : n = k * m
  l : Z
  H2 : m = l * n
  H3 := H2 : m = l * n
  ============================
  m = n

divide_antisymmetric < rewrite H1 in H3.
1 subgoal

  m, n : Z
  Pm : m > 0
  Pn : n > 0
  k : Z
  H1 : n = k * m
  l : Z
  H2 : m = l * n
  H3 : m = l * (k * m)
  ============================
  m = n

divide_antisymmetric < rewrite Z.mul_assoc in H3.
1 subgoal

  m, n : Z
  Pm : m > 0
  Pn : n > 0
  k : Z
  H1 : n = k * m
  l : Z
  H2 : m = l * n
  H3 : m = l * k * m
  ============================
  m = n

divide_antisymmetric < rewrite <- Z.mul_1_l with (n:=m) in H3 at 1.
1 subgoal

  m, n : Z
  Pm : m > 0
  Pn : n > 0
  k : Z
  H1 : n = k * m
  l : Z
  H2 : m = l * n
  H3 : 1 * m = l * k * m
  ============================
  m = n
```

```
divide_antisymmetric < apply Z.mul_cancel_r in H3.
2 subgoals

  m, n : Z
  Pm : m > 0
  Pn : n > 0
  k : Z
  H1 : n = k * m
  l : Z
  H2 : m = l * n
  H3 : 1 = l * k
  ============================
  m = n

subgoal 2 is:
 m <> 0

divide_antisymmetric < symmetry in H3.
2 subgoals

  m, n : Z
  Pm : m > 0
  Pn : n > 0
  k : Z
  H1 : n = k * m
  l : Z
  H2 : m = l * n
  H3 : l * k = 1
  ============================
  m = n

subgoal 2 is:
 m <> 0
```

Next we can use a theorem in the proof assistant saying that if the product of two integers is one, and one of the integers is non-negative, both integers are one themselves. This allows us to rewrite the goal and solve it easily.

```
divide_antisymmetric < apply Z.eq_mul_1_nonneg in H3 as H4.
3 subgoals

  m, n : Z
  Pm : m > 0
  Pn : n > 0
  k : Z
  H1 : n = k * m
  l : Z
  H2 : m = l * n
  H3 : l * k = 1
  H4 : l = 1 /\ k = 1
  ============================
  m = n

subgoal 2 is:
 0 <= l
subgoal 3 is:
 m <> 0

divide_antisymmetric < destruct H4 as [H4 H5].
3 subgoals

  m, n : Z
  Pm : m > 0
  Pn : n > 0
  k : Z
  H1 : n = k * m
  l : Z
  H2 : m = l * n
  H3 : l * k = 1
  H4 : l = 1
  H5 : k = 1
  ============================
  m = n

subgoal 2 is:
 0 <= l
subgoal 3 is:
 m <> 0

divide_antisymmetric < rewrite H1.
3 subgoals

  m, n : Z
  Pm : m > 0
  Pn : n > 0
  k : Z
  H1 : n = k * m
  l : Z
  H2 : m = l * n
  H3 : l * k = 1
  H4 : l = 1
  H5 : k = 1
  ============================
  m = k * m

subgoal 2 is:
 0 <= l
subgoal 3 is:
 m <> 0

divide_antisymmetric < rewrite H5.
3 subgoals

  m, n : Z
  Pm : m > 0
  Pn : n > 0
  k : Z
  H1 : n = k * m
  l : Z
  H2 : m = l * n
  H3 : l * k = 1
  H4 : l = 1
  H5 : k = 1
  ============================
  m = 1 * m

subgoal 2 is:
 0 <= l
subgoal 3 is:
 m <> 0

divide_antisymmetric < ring.
2 subgoals

  m, n : Z
  Pm : m > 0
  Pn : n > 0
  k : Z
  H1 : n = k * m
  l : Z
  H2 : m = l * n
  H3 : l * k = 1
  ============================
  0 <= l
```

```
subgoal 2 is:
 m <> 0
```

Next we need to prove that l is non-negative. We know this because $m = ln$ by Hypothesis 2, and both m and n are positive. We can use a theorem that says that if we multiply both sides of a less-than-or-equal-to statement by a positive integer, the statement remains true.

```
     : forall n m p : Z, 0 < p -> n * p <= m * p -> n <= m
divide_antisymmetric < apply Zmult_lt_0_le_reg_r with (n:=0)(m:=l)(p:=n).
3 subgoals

  m, n : Z
  Pm : m > 0
  Pn : n > 0
  k : Z
  H1 : n = k * m
  l : Z
  H2 : m = l * n
  H3 : l * k = 1
  ============================
  0 < n

subgoal 2 is:
 0 * n <= l * n
subgoal 3 is:
 m <> 0

divide_antisymmetric < apply Z.gt_lt.
3 subgoals

  m, n : Z
  Pm : m > 0
  Pn : n > 0
  k : Z
  H1 : n = k * m
  l : Z
  H2 : m = l * n
  H3 : l * k = 1
  ============================
  n > 0

subgoal 2 is:
 0 * n <= l * n
subgoal 3 is:
 m <> 0

divide_antisymmetric < exact Pn.
2 subgoals

  m, n : Z
  Pm : m > 0
  Pn : n > 0
  k : Z
  H1 : n = k * m
  l : Z
  H2 : m = l * n
  H3 : l * k = 1
  ============================
  0 * n <= l * n

subgoal 2 is:
 m <> 0

divide_antisymmetric < rewrite <- H2.
2 subgoals
```

```
  m, n : Z
  Pm : m > 0
  Pn : n > 0
  k : Z
  H1 : n = k * m
  l : Z
  H2 : m = l * n
  H3 : l * k = 1
  ============================
  0 * n <= m

subgoal 2 is:
 m <> 0

divide_antisymmetric < simpl.
2 subgoals

  m, n : Z
  Pm : m > 0
  Pn : n > 0
  k : Z
  H1 : n = k * m
  l : Z
  H2 : m = l * n
  H3 : l * k = 1
  ============================
  0 <= m

subgoal 2 is:
 m <> 0

divide_antisymmetric < apply Z.gt_lt in Pm.
2 subgoals

  m, n : Z
  Pm : 0 < m
  Pn : n > 0
  k : Z
  H1 : n = k * m
  l : Z
  H2 : m = l * n
  H3 : l * k = 1
  ============================
  0 <= m

subgoal 2 is:
 m <> 0

divide_antisymmetric < Print Z.lt_le_incl.
Fetching opaque proofs from disk for Coq.Numbers.NatInt.NZOrder
Z.lt_le_incl =
fun (n m : Z) (H : n < m) =>
let H0 :=
  (fun n0 m0 : Z => match Z.lt_eq_cases n0 m0 with
                  | conj _ x0 => x0
                  end)
  :
  forall n0 m0 : Z, n0 < m0 \/ n0 = m0 -> n0 <= m0 in
H0 n m (or_introl H)
     : forall n m : Z, n < m -> n <= m

Argument scopes are [Z_scope Z_scope _]

divide_antisymmetric < apply Z.lt_le_incl in Pm.
2 subgoals

  m, n : Z
  Pm : 0 <= m
  Pn : n > 0
  k : Z
  H1 : n = k * m
  l : Z
  H2 : m = l * n
  H3 : l * k = 1
  ============================
  0 <= m

subgoal 2 is:
 m <> 0

divide_antisymmetric < exact Pm.
1 subgoal

  m, n : Z
  Pm : m > 0
  Pn : n > 0
  k : Z
  H1 : n = k * m
  l : Z
  H2 : m = l * n
  H3 : 1 * m = l * k * m
  H : forall n m p : Z, p <> 0 -> n * p = m * p -> n = m
  ============================
  m <> 0

divide_antisymmetric < intro H4.
1 subgoal

  m, n : Z
  Pm : m > 0
  Pn : n > 0
  k : Z
  H1 : n = k * m
  l : Z
  H2 : m = l * n
  H3 : 1 * m = l * k * m
  H : forall n m p : Z, p <> 0 -> n * p = m * p -> n = m
  H4 : m = 0
  ============================
  False

divide_antisymmetric < rewrite H4 in Pm.
1 subgoal

  m, n : Z
  Pm : 0 > 0
  Pn : n > 0
  k : Z
  H1 : n = k * m
  l : Z
  H2 : m = l * n
  H3 : 1 * m = l * k * m
  H : forall n m p : Z, p <> 0 -> n * p = m * p -> n = m
  H4 : m = 0
  ============================
  False

divide_antisymmetric < discriminate.
No more subgoals.

divide_antisymmetric < Qed.
(intros m n Pm Pn H1 H2).
(destruct H1 as [k H1]).
(destruct H2 as [l H2]).
(pose (H3 := H2)).
(rewrite H1 in H3).
```

```
(rewrite Z.mul_assoc in H3).
(rewrite <- Z.mul_1_l with (n := m) in H3 at 1).
(apply Z.mul_cancel_r in H3).
 symmetry in H3.
 (apply Z.eq_mul_1_nonneg in H3 as H4).
  (destruct H4 as [H4 H5]).
  (rewrite H1).
  (rewrite H5).
  ring.

  (apply Zmult_lt_0_le_reg_r with (n := 0) (m := 1) (p := n)).
   (apply Z.gt_lt).
   exact Pn.

   (rewrite <- H2).
   (simpl).
   (apply Z.gt_lt in Pm).
   (apply Z.lt_le_incl in Pm).
   exact Pm.

 intro H4.
 (rewrite H4 in Pm).
 discriminate.

Qed.
divide_antisymmetric is defined
```

Finally, we can also prove the lemma we stated above, starting by introducing all hypotheses.

Not only is the divisibility relation a partial order, but the set of all integers divisible by some d forms a ring.

```
Coq < Lemma div_ring : forall m n d x y : Z, (d | x) /\ (d | y) -> (d | m*x + n*y).
1 subgoal

  ============================
  forall m n d x y : Z, (d | x) /\ (d | y) -> (d | m * x + n * y)

div_ring < intros m n d x y.
1 subgoal

  m, n, d, x, y : Z
  ============================
  (d | x) /\ (d | y) -> (d | m * x + n * y)

div_ring < intro H.
1 subgoal

  m, n, d, x, y : Z
  H : (d | x) /\ (d | y)
  ============================
  (d | m * x + n * y)
```

We destruct the divisibility hypotheses to get the witnesses making the statements true.

```
div_ring < destruct H as [Dx Dy].
1 subgoal

  m, n, d, x, y : Z
  Dx : (d | x)
  Dy : (d | y)
  ============================
  (d | m * x + n * y)
```

```
div_ring < unfold Z.divide.
1 subgoal

  m, n, d, x, y : Z
  Dx : (d | x)
  Dy : (d | y)
  ============================
  exists z : Z, m * x + n * y = z * d

div_ring < destruct Dx as [k Dx].
1 subgoal

  m, n, d, x, y, k : Z
  Dx : x = k * d
  Dy : (d | y)
  ============================
  exists z : Z, m * x + n * y = z * d

div_ring < destruct Dy as [l Dy].
1 subgoal

  m, n, d, x, y, k : Z
  Dx : x = k * d
  l : Z
  Dy : y = l * d
  ============================
  exists z : Z, m * x + n * y = z * d
```

Now we can specify the witness for the goal, rewrite using our hypotheses, and use the ring tactic to verify the equality.

```
div_ring < exists (m*k + n*l).
1 subgoal

  m, n, d, x, y, k : Z
  Dx : x = k * d
  l : Z
  Dy : y = l * d
  ============================
  m * x + n * y = (m * k + n * l) * d

div_ring < rewrite Dx.
1 subgoal

  m, n, d, x, y, k : Z
  Dx : x = k * d
  l : Z
  Dy : y = l * d
  ============================
  m * (k * d) + n * y = (m * k + n * l) * d

div_ring < rewrite Dy.
1 subgoal

  m, n, d, x, y, k : Z
  Dx : x = k * d
  l : Z
  Dy : y = l * d
  ============================
  m * (k * d) + n * (l * d) = (m * k + n * l) * d

div_ring < ring.
No more subgoals.

div_ring < Qed.
(intros m n d x y).
intro H.
(destruct H as [Dx Dy]).
```

```
(unfold Z.divide).
(destruct Dx as [k Dx]).
(destruct Dy as [l Dy]).
exists (m * k + n * l).
(rewrite Dx).
(rewrite Dy).
ring.

Qed.
div_ring is defined
```

5.4 The GCD

The *greatest common divisor* (GCD) is a fundamental concept in number theory, and as such is known by many different names, such as the greatest common factor (GCF), the highest common factor (HCF), the highest common divisor (HCD), and the greatest common measure (GCM). The greatest common divisor g is the largest natural number that divides both a and b without leaving a remainder, or

$$\gcd(a,b) = \max\{g \in \mathbb{N} \mid g|a \wedge g|b\}. \tag{5.21}$$

Notice that the set is finite because for any g greater than $\min(a,b)$ the predicate is false. If d is any common divisor of positive integers m and n, then we can tile a rectangle of dimension $m \times n$ with squares of side d, as shown in Figure 5.1.

If $\gcd(a,b) = 1$, then a and b are coprime, or relatively prime, since they have no common factors. Remember that we denote this by $a \perp b$. This does not imply that a or b are themselves prime numbers. For example, neither 14 nor 15 is a prime number, since they both have two prime factors: $14 = 2 \cdot 7$ and $15 = 5 \cdot 3$, but 14 and 15 are coprime since they have no factors in common. Note that $\gcd(a,0) = |a|$ because any number divides zero.

Let $g = \gcd(a,b)$. Since $g|a$ and $g|b$, they can be written $a = mg$ and $b = ng$, and there is no larger number $g' > g$ for which this is true. The natural numbers m and n must be coprime, since any common factor could be factored out of m and n to make g greater. We can prove that any other number c that divides both a and b must also divide g, so

$$c|a \wedge c|b \implies c|g. \tag{5.22}$$

First we start by assuming that $c \perp g$. Now $cm = a$ and $cn = b$, but $g|a$ and $g \perp c$, so $g|m$ and likewise $g|n$. This means that $gc > g$ must divide a and b, which is impossible since g is the GCD, therefore $c|g$. Now suppose c and g have some common factor d. This must also be a common factor of a and b. We divide c, g, and a by d, giving c', g', and a'. This is exactly the problem we had before, since now $c' \perp g'$ and $g' = \gcd(a',b)$. This implies $c'|g'$ and thus $c|g$, so that the GCD of a and b is the unique, positive common divisor of a and b that is divisible by any other common divisor. This is an alternate definition of the GCD that is superior for many problems.

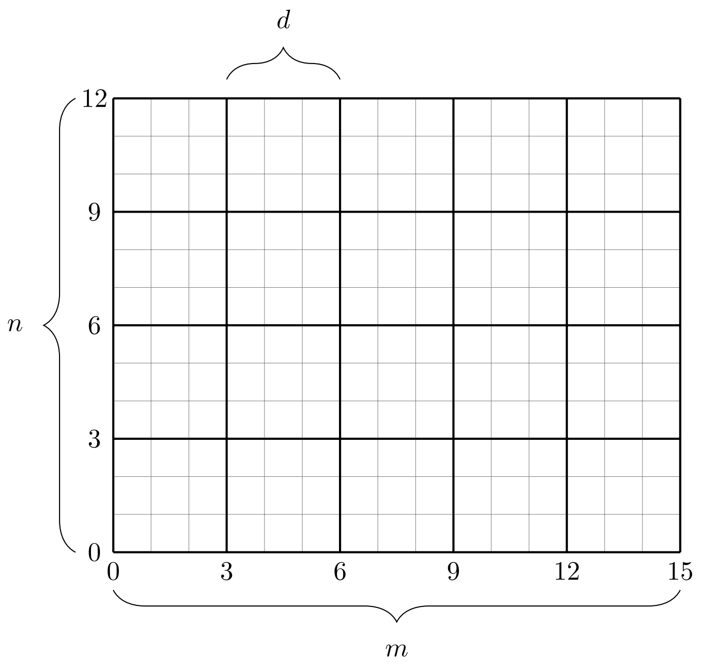


Figure 5.1: Illustration of $d|m$ and $d|n$.

This is easier to see with yet another definition of the GCD. The greatest common divisor g of two nonzero numbers a and b is also their smallest positive integral linear combination, that is, the smallest positive number of the form $ua+vb$ where u and v are integers. Suppose that c is a common divisor, so that we have $a = ck$ and $b = cl$ for some integers k and l. Thus

$$\begin{aligned} d &= ua + vb, && (5.23)\\ &= u(ck) + v(cl), && (5.24)\\ &= c(uk + vl). && (5.25) \end{aligned}$$

Thus clearly $c|d$. We will revisit this proof in Section 5.4.1 on Bezout's Theorem. We can visualize this relation between d and $ua + vb$ in three dimensions. We will think of all the integers $z = ua + vb$ as being points in a plane in (u, v, z) space. For example, we have drawn such as plane in Figure 5.2 for $a = 6$ and $b = 3$. This plane always goes through the origin when we choose $u = v = 0$, since 0 is divisible by any number. If we increase u by one, we increase the sum z by a, so we have slope a in the u-direction. Likewise, if we increase v by one, we increase z by b, so we have slope b in the v-direction. This means that the vectors $(1, 0, a)$ and $(0, 1, b)$ lie in this plane, and the normal to the plane is given by $(-a, -b, 1)$, the cross product of these two vectors. As the factors a and b become large, the normal almost lies in the (u, v)-plane.

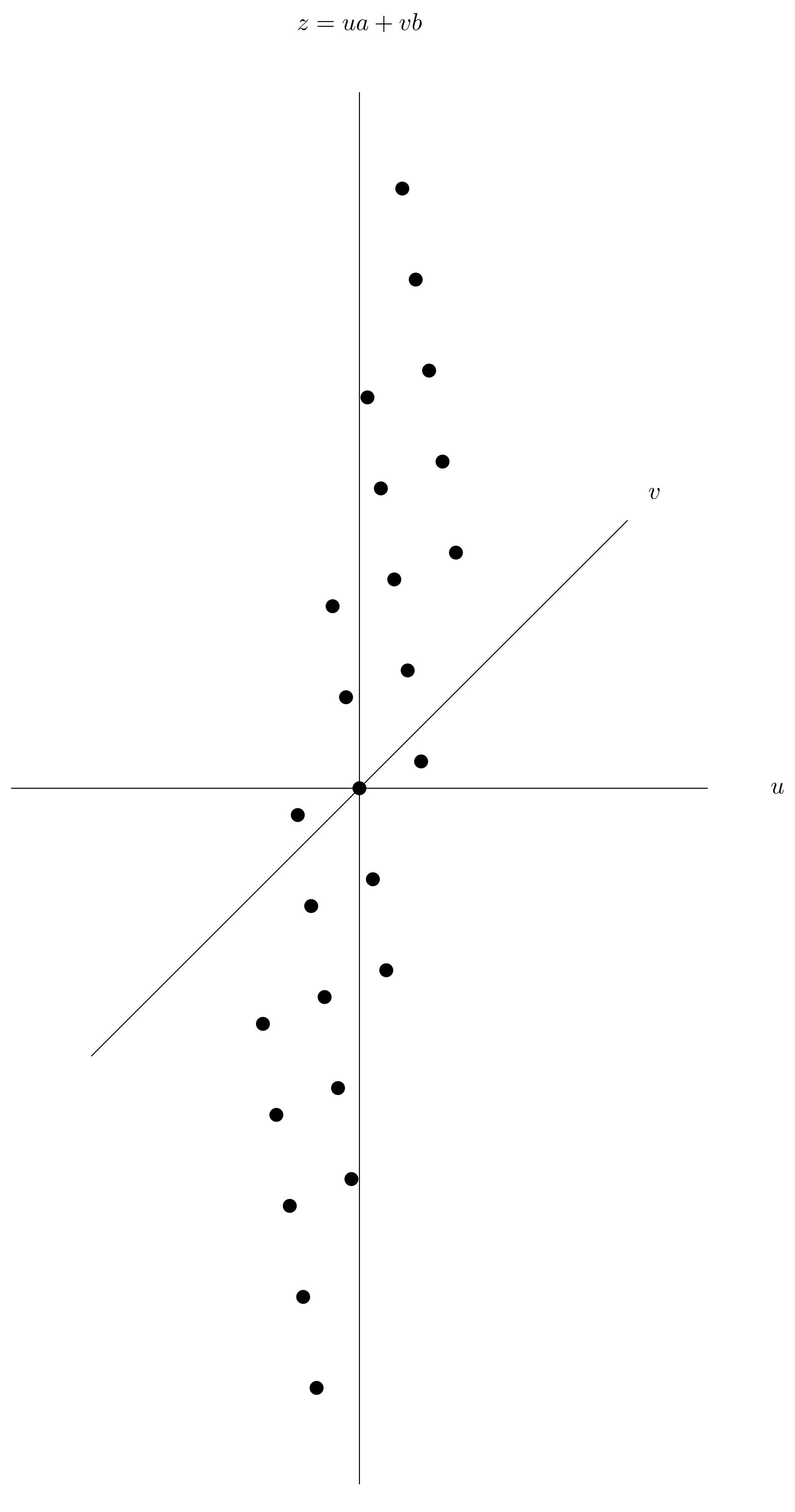


Figure 5.2: Plane of numbers divisible by d for $a = 6$ and $b = 3$.

We know that all points in our plane traced out by z are divisible by all common divisors, by Equation 5.8. We also know that the smallest *positive* point must be a common divisor, since its remainder is also in the plane, and smaller, meaning that the remainder is 0. This is a graphical proof of Bezout's Theorem, which we will prove algebraically in Section 5.4.1. In Section 5.5, we will see that the Euclidean Algorithm chooses lines of points in this plane through which to descend to the GCD.

The GCD of three or more numbers equals the product of the prime factors common to all the numbers, but it can also be calculated by repeatedly taking the GCDs of pairs of numbers. For example,

$$\gcd(a, b, c) = \gcd(a, \gcd(b, c)) = \gcd(\gcd(a, b), c) = \gcd(\gcd(a, c), b). \tag{5.26}$$

Thus any algorithm which computes the GCD of two integers suffices to calculate the GCD of arbitrarily many integers. Also useful is

$$\forall m \in \mathbb{Z}, \gcd(a + mb, b) = \gcd(a, b). \tag{5.27}$$

To prove this, we first observe that since for $g = \gcd(a, b)$ we have $g|a$ and $g|b$, then $g|a + mb$ so that g is a common divisor of $a + mb$ and b. Conversely, if $h|b$ and $h|a + mb$, then $h|a + mb + (-m)b = a$ so that h is a common divisor of a and b. Thus the common divisors of a, b and $a, a + mb$ are identical, and therefore the GCD are also identical. This is an elementary use of Eq. (5.8).

5.4.1 Bézout's Theorem

Bézout's Theorem states that the GCD g of two integers a and b can be represented as a linear sum of the original two numbers,

$$\exists s, t \in \mathbb{Z}, g = sa + tb. \tag{5.28}$$

We can justify this statement using an argument based on the Well-Ordering Principle for positive numbers. Given any nonzero integers a and b, let

$$\mathcal{S} = \{ax + by \mid x, y \in \mathbb{Z} \wedge ax + by > 0\}.$$

The set $\mathcal{S}$ is nonempty since it contains either a or $-a$ (with $x = \pm 1$ and $y = 0$). Since $\mathcal{S}$ is a nonempty set of positive integers, it has a minimum element $d = as + bt$, by the Well-Ordering Principle. To prove that d is the greatest common divisor of a and b, we must prove that d is a common divisor of a and b, and that for any other common divisor c, one has $c < d$.

The Euclidean division of a by d may be written

$$a = dq + r \quad \text{with} \quad 0 \leq r < d. \tag{5.29}$$

The remainder r is in $\mathcal{S} \cup \{0\}$, because

$$\begin{aligned} r &= a - qd && (5.30)\\ &= a - q(as + bt) && (5.31)\\ &= a(1 - qs) - bqt. && (5.32) \end{aligned}$$

As d is the smallest positive integer in $\mathcal{S}$, the remainder r is necessarily 0, and this implies that d is a divisor of a. Similarly d is also a divisor of b, so that d is a common divisor of a and b.

Now, let c be any common divisor of a and b; that is there exist u and v such that $a = cu$ and $b = cv$. One has thus

$$\begin{aligned} d &= as + bt && (5.33)\\ &= cus + cvt && (5.34)\\ &= c(us + vt). && (5.35) \end{aligned}$$

That is c is a divisor of d, and therefore $c \le d$.

5.4.2 Coq Proofs

We define $\gcd(a, b)$ as an inductive predicate, rather than a function, since it will be a proposition telling us that g is the gcd of a and b. Later, we will verify that it is unique up to a sign. Notice that the definition uses the fact we proved above, that all common divisors divide the gcd. This provides an alternate starting point, so that we do not have to formalize the notion of "greatest". First we must import packages with the definition of the GCD.

```
Require Import ZArith.
Require Import Znumtheory.
Open Scope Z_scope.
```

Notice that we name the implication `Zis_gcd_intro` below, which will allow us to use that theorem in our proofs.

```
Coq> Print Zis_gcd.
Inductive Zis_gcd (a b g : Z) : Prop :=
   Zis_gcd_intro : (g | a) ->
                 (g | b) ->
                 (forall x : Z, (x | a) -> (x | b) -> (x | g)) ->
                 Zis_gcd a b g
```

As a first step, we will prove that gcd is a symmetric predicate. We state the lemma, introduce hypotheses, and remove the implication.

```
Coq < Lemma Zis_gcd_sym : forall a b d, Zis_gcd a b d -> Zis_gcd b a d.
1 subgoal

  A : Set
  ============================
  forall a b d : Z, Zis_gcd a b d -> Zis_gcd b a d

Zis_gcd_sym < intros a b d.
1 subgoal

  A : Set
  a, b, d : Z
  ============================
  Zis_gcd a b d -> Zis_gcd b a d

Zis_gcd_sym < intro H.
1 subgoal

  A : Set
  a, b, d : Z
  H : Zis_gcd a b d
  ============================
  Zis_gcd b a d
```

Now we would like to somehow unfold the inductive definition of the gcd. We do this by applying the theorem above from the definition of the gcd. This conforms with our intuition about the greatest common divisor, namely that the gcd d must divide both a and b and that any other divisor must divide d.

```
Zis_gcd_sym < apply Zis_gcd_intro.
3 subgoals

  A : Set
  a, b, d : Z
  H : Zis_gcd a b d
  ============================
  (d | b)

subgoal 2 is:
 (d | a)
subgoal 3 is:
 forall x : Z, (x | b) -> (x | a) -> (x | d)
```

The analogue for our hypothesis H is to destruct it, replacing it with the left side of the implication. This makes it a simple matter to prove the first goal, since it is just an assumption.

```
Zis_gcd_sym < destruct H.
3 subgoals

  A : Set
  a, b, d : Z
  H : (d | a)
  H0 : (d | b)
  H1 : forall x : Z, (x | a) -> (x | b) -> (x | d)
  ============================
  (d | b)

subgoal 2 is:
 (d | a)
subgoal 3 is:
 forall x : Z, (x | b) -> (x | a) -> (x | d)

Zis_gcd_sym < assumption.
2 subgoals

  A : Set
  a, b, d : Z
  H : Zis_gcd a b d
  ============================
  (d | a)

subgoal 2 is:
 forall x : Z, (x | b) -> (x | a) -> (x | d)
```

We can use an identical procedure for the second goal.

```
Zis_gcd_sym < destruct H.
2 subgoals

  A : Set
  a, b, d : Z
  H : (d | a)
  H0 : (d | b)
  H1 : forall x : Z, (x | a) -> (x | b) -> (x | d)
  ============================
  (d | a)

subgoal 2 is:
 forall x : Z, (x | b) -> (x | a) -> (x | d)

Zis_gcd_sym < assumption.
1 subgoal

  A : Set
  a, b, d : Z
  H : Zis_gcd a b d
  ============================
  forall x : Z, (x | b) -> (x | a) -> (x | d)
```

We now introduce hypotheses from the last goal, and apply the hypothesis H_1 which will be instantiated for integer x.

```
Zis_gcd_sym < intros x divB divA.
1 subgoal

  A : Set
  a, b, d : Z
  H : (d | a)
  H0 : (d | b)
  H1 : forall x : Z, (x | a) -> (x | b) -> (x | d)
  x : Z
  divB : (x | b)
  divA : (x | a)
  ============================
  (x | d)

Zis_gcd_sym < apply H1.
2 subgoals

  A : Set
  a, b, d : Z
  H : (d | a)
  H0 : (d | b)
  H1 : forall x : Z, (x | a) -> (x | b) -> (x | d)
  x : Z
  divB : (x | b)
  divA : (x | a)
  ============================
  (x | a)

subgoal 2 is:
 (x | b)
```

Now we just use assumptions to prove the theorem.

```
Zis_gcd_sym < assumption.
1 subgoal

  A : Set
  a, b, d : Z
  H : (d | a)
  H0 : (d | b)
  H1 : forall x : Z, (x | a) -> (x | b) -> (x | d)
  x : Z
  divB : (x | b)
  divA : (x | a)
  ============================
  (x | b)

Zis_gcd_sym < assumption.
No more subgoals.

Zis_gcd_sym < Qed.
```

```
(intros a b d).
(simpl).
intro H.
(apply Zis_gcd_intro).
 (destruct H).
 assumption.

 (destruct H).
 assumption.

 (destruct H).
 (intros x).
 (intros divB divA).
 (apply H1).
  assumption.

  assumption.

Qed.
Zis_gcd_sym is defined
```

We can now also prove Eq. (5.27). We begin with introduction of hypotheses and eliminating the implication.

```
Coq < Lemma Zis_gcd_for_euclid : forall a b d q:Z, Zis_gcd b (a - q * b) d -> Zis_gcd a b d.
1 subgoal

  A : Set
  ============================
  forall a b d q : Z, Zis_gcd b (a - q * b) d -> Zis_gcd a b d

Zis_gcd_for_euclid < intros a b d q.
1 subgoal

  A : Set
  a, b, d, q : Z
  ============================
  Zis_gcd b (a - q * b) d -> Zis_gcd a b d

Zis_gcd_for_euclid < intro H.
1 subgoal

  A : Set
  a, b, d, q : Z
  H : Zis_gcd b (a - q * b) d
  ============================
  Zis_gcd a b d
```

Just as in our prior proof, we apply the gcd theorem,

```
Zis_gcd_for_euclid < apply Zis_gcd_intro.
3 subgoals

  A : Set
  a, b, d, q : Z
  H : Zis_gcd b (a - q * b) d
  ============================
  (d | a)

subgoal 2 is:
 (d | b)
subgoal 3 is:
 forall x : Z, (x | a) -> (x | b) -> (x | d)
```

and destruct our hypothesis H.

```
Zis_gcd_for_euclid < destruct H.
3 subgoals
```

```
  A : Set
  a, b, d, q : Z
  H : (d | b)
  H0 : (d | a - q * b)
  H1 : forall x : Z, (x | b) -> (x | a - q * b) -> (x | d)
  ============================
  (d | a)

subgoal 2 is:
 (d | b)
subgoal 3 is:
 forall x : Z, (x | a) -> (x | b) -> (x | d)
```

In this proof, however, we must pick apart the notion of divisibility. We can unfold the definition in the goal, and produce witnesses for hypotheses H and H_0,

```
Zis_gcd_for_euclid < unfold Z.divide.
3 subgoals

  A : Set
  a, b, d, q : Z
  H : (d | b)
  H0 : (d | a - q * b)
  H1 : forall x : Z, (x | b) -> (x | a - q * b) -> (x | d)
  ============================
  exists z : Z, a = z * d

subgoal 2 is:
 (d | b)
subgoal 3 is:
 forall x : Z, (x | a) -> (x | b) -> (x | d)

Zis_gcd_for_euclid < destruct H.
3 subgoals

  A : Set
  a, b, d, q, x : Z
  H : b = x * d
  H0 : (d | a - q * b)
  H1 : forall x : Z, (x | b) -> (x | a - q * b) -> (x | d)
  ============================
  exists z : Z, a = z * d

subgoal 2 is:
 (d | b)
subgoal 3 is:
 forall x : Z, (x | a) -> (x | b) -> (x | d)

Zis_gcd_for_euclid < destruct H0.
3 subgoals

  A : Set
  a, b, d, q, x : Z
  H : b = x * d
  x0 : Z
  H0 : a - q * b = x0 * d
  H1 : forall x : Z, (x | b) -> (x | a - q * b) -> (x | d)
  ============================
  exists z : Z, a = z * d

subgoal 2 is:
 (d | b)
subgoal 3 is:
 forall x : Z, (x | a) -> (x | b) -> (x | d)
```

Now we can use the equality in the new H to rewrite H_0

```
Zis_gcd_for_euclid < rewrite H in H0.
3 subgoals

  A : Set
  a, b, d, q, x : Z
  H : b = x * d
  x0 : Z
  H0 : a - q * (x * d) = x0 * d
  H1 : forall x : Z, (x | b) -> (x | a - q * b) -> (x | d)
  ============================
  exists z : Z, a = z * d

subgoal 2 is:
 (d | b)
subgoal 3 is:
 forall x : Z, (x | a) -> (x | b) -> (x | d)
```

and then we shift the negative term in $H0$ to the other side,

```
Zis_gcd_for_euclid < apply Z.sub_move_r in H0.
3 subgoals

  A : Set
  a, b, d, q, x : Z
  H : b = x * d
  x0 : Z
  H0 : a = x0 * d + q * (x * d)
  H1 : forall x : Z, (x | b) -> (x | a - q * b) -> (x | d)
  ============================
  exists z : Z, a = z * d

subgoal 2 is:
 (d | b)
subgoal 3 is:
 forall x : Z, (x | a) -> (x | b) -> (x | d)
```

rewrite the goal,

```
Zis_gcd_for_euclid < rewrite H0.
3 subgoals

  A : Set
  a, b, d, q, x : Z
  H : b = x * d
  x0 : Z
  H0 : a = x0 * d + q * (x * d)
  H1 : forall x : Z, (x | b) -> (x | a - q * b) -> (x | d)
  ============================
  exists z : Z, x0 * d + q * (x * d) = z * d

subgoal 2 is:
 (d | b)
subgoal 3 is:
 forall x : Z, (x | a) -> (x | b) -> (x | d)
```

and we have our witness

```
Zis_gcd_for_euclid < exists (x0 + q*x).
3 subgoals

  A : Set
  a, b, d, q, x : Z
  H : b = x * d
  x0 : Z
  H0 : a = x0 * d + q * (x * d)
  H1 : forall x : Z, (x | b) -> (x | a - q * b) -> (x | d)
  ============================
  x0 * d + q * (x * d) = (x0 + q * x) * d

subgoal 2 is:
 (d | b)
subgoal 3 is:
 forall x : Z, (x | a) -> (x | b) -> (x | d)

Zis_gcd_for_euclid < ring.
2 subgoals

  A : Set
  a, b, d, q : Z
  H : Zis_gcd b (a - q * b) d
  ============================
  (d | b)

subgoal 2 is:
 forall x : Z, (x | a) -> (x | b) -> (x | d)
```

The second goal is easier since it becomes an assumption.

```
Zis_gcd_for_euclid < destruct H.
2 subgoals

  A : Set
  a, b, d, q : Z
  H : (d | b)
  H0 : (d | a - q * b)
  H1 : forall x : Z, (x | b) -> (x | a - q * b) -> (x | d)
  ============================
  (d | b)

subgoal 2 is:
 forall x : Z, (x | a) -> (x | b) -> (x | d)

Zis_gcd_for_euclid < assumption.
1 subgoal

  A : Set
  a, b, d, q : Z
  H : Zis_gcd b (a - q * b) d
  ============================
  forall x : Z, (x | a) -> (x | b) -> (x | d)
```

For the last goal, we begin by introducing hypotheses and expanding the gcd definition.

```
Zis_gcd_for_euclid < intros y divA divB.
1 subgoal

  A : Set
  a, b, d, q : Z
  H : Zis_gcd b (a - q * b) d
  y : Z
  divA : (y | a)
  divB : (y | b)
  ============================
  (y | d)

Zis_gcd_for_euclid < destruct H.
1 subgoal

  A : Set
  a, b, d, q : Z
  H : (d | b)
  H0 : (d | a - q * b)
  H1 : forall x : Z, (x | b) -> (x | a - q * b) -> (x | d)
  y : Z
  divA : (y | a)
  divB : (y | b)
  ============================
  (y | d)
```

Now we can apply hypothesis H_1, generating two new goals. The first is trivial, as it is already an assumption.

```
Zis_gcd_for_euclid < apply H1.
2 subgoals

  A : Set
  a, b, d, q : Z
  H : (d | b)
  H0 : (d | a - q * b)
  H1 : forall x : Z, (x | b) -> (x | a - q * b) -> (x | d)
  y : Z
  divA : (y | a)
  divB : (y | b)
  ============================
  (y | b)

subgoal 2 is:
 (y | a - q * b)

Zis_gcd_for_euclid < assumption.
1 subgoal

  A : Set
  a, b, d, q : Z
  H : (d | b)
  H0 : (d | a - q * b)
  H1 : forall x : Z, (x | b) -> (x | a - q * b) -> (x | d)
  y : Z
  divA : (y | a)
  divB : (y | b)
  ============================
  (y | a - q * b)
```

The second needs a suitable witness.

```
Zis_gcd_for_euclid < unfold Z.divide.
1 subgoal

  A : Set
  a, b, d, q : Z
  H : (d | b)
  H0 : (d | a - q * b)
  H1 : forall x : Z, (x | b) -> (x | a - q * b) -> (x | d)
  y : Z
  divA : (y | a)
  divB : (y | b)
  ============================
  exists z : Z, a - q * b = z * y
```

We can construct the witness for the goal by first extracting the witnesses from our divisibility hypotheses.

```
Zis_gcd_for_euclid < destruct divA.
1 subgoal

  A : Set
  a, b, d, q : Z
  H : (d | b)
  H0 : (d | a - q * b)
  H1 : forall x : Z, (x | b) -> (x | a - q * b) -> (x | d)
  y, x : Z
  H2 : a = x * y
  divB : (y | b)
  ============================
  exists z : Z, a - q * b = z * y

Zis_gcd_for_euclid < destruct divB.
1 subgoal

  A : Set
  a, b, d, q : Z
  H : (d | b)
  H0 : (d | a - q * b)
  H1 : forall x : Z, (x | b) -> (x | a - q * b) -> (x | d)
  y, x : Z
  H2 : a = x * y
  x0 : Z
  H3 : b = x0 * y
  ============================
  exists z : Z, a - q * b = z * y
```

After rewriting the goal, our witness $x - qx_0$ is clear.

```
Zis_gcd_for_euclid < rewrite H2.
1 subgoal

  A : Set
  a, b, d, q : Z
  H : (d | b)
  H0 : (d | a - q * b)
  H1 : forall x : Z, (x | b) -> (x | a - q * b) -> (x | d)
  y, x : Z
  H2 : a = x * y
  x0 : Z
  H3 : b = x0 * y
  ============================
  exists z : Z, x * y - q * b = z * y

Zis_gcd_for_euclid < rewrite H3.
1 subgoal

  A : Set
  a, b, d, q : Z
  H : (d | b)
  H0 : (d | a - q * b)
  H1 : forall x : Z, (x | b) -> (x | a - q * b) -> (x | d)
  y, x : Z
  H2 : a = x * y
  x0 : Z
  H3 : b = x0 * y
  ============================
  exists z : Z, x * y - q * (x0 * y) = z * y

Zis_gcd_for_euclid < exists (x - q*x0).
1 subgoal

  A : Set
  a, b, d, q : Z
  H : (d | b)
  H0 : (d | a - q * b)
  H1 : forall x : Z, (x | b) -> (x | a - q * b) -> (x | d)
  y, x : Z
  H2 : a = x * y
  x0 : Z
  H3 : b = x0 * y
  ============================
  x * y - q * (x0 * y) = (x - q * x0) * y

Zis_gcd_for_euclid < ring.
```

```
No more subgoals.

Zis_gcd_for_euclid < Qed.
(intros a b d q).
intro H.
(apply Zis_gcd_intro).
 (destruct H).
 (unfold Z.divide).
 (destruct H).
 (destruct H0).
 (rewrite H in H0).
 (apply Z.sub_move_r in H0).
 (rewrite H0).
 exists (x0 + q * x).
 ring.

 (destruct H).
 assumption.

 (intros y divA divB).
 (destruct H).
 (apply H1).
  assumption.

  (unfold Z.divide).
  (destruct divA).
  (destruct divB).
  (rewrite H2).
  (rewrite H3).
  exists (x - q * x0).
  ring.

Qed.
Zis_gcd_for_euclid is defined
```

5.5 The Euclidean Algorithm

The *Euclidean Algorithm* is an iterative process that computes the GCD of two integers. At each step, it uses the division algorithm from Section 5.2 to compute a remainder r_n from the division of two previous numbers r_{n-2}/r_{n-1}, which we can express as

$$r_{n-2} = q_n r_{n-1} + r_n. \tag{5.36}$$

What is the motivation for this computation? Very often, an iterative method can be defined by an invariant preserved by each iteration of the loop. This is the basis for reasoning about loops in Hoare logic. Now, we are looking for $g = \gcd(a, b)$, and without loss of generality we assume $a \geq b$. Our induction hypothesis $P(n)$ will be that

$$P(n) := g = \gcd(r_{n-1}, r_n) \tag{5.37}$$

where $g = \gcd(a, b)$. For our base case $P(0)$, suppose that we start with $r_{-2} = a$ and $r_{-1} = b$, meaning that $P(-1)$ is true. Then we have

$$\begin{aligned} P(0) := g &= \gcd(r_{-1}, r_0) & (5.38)\\ &= \gcd(b, r_0) & (5.39)\\ &= \mathrm{T}, & (5.40) \end{aligned}$$

which follows from Eq. (5.27). Now, assuming $P(n)$ is true, the inductive step is

$$\begin{aligned} P(n+1) := g &= \gcd(r_n, r_{n+1}) && (5.41)\\ &= \gcd(r_n, q_n r_n + r_{n-1}) && (5.42)\\ &= \gcd(r_n, r_{n-1}) && (5.43)\\ &= \mathrm{T}, && (5.44) \end{aligned}$$

where we again used Eq. (5.27), which follows from Eq. (5.8). By the division algorithm $0 \le r_n < r_{n-1}$, so at each step the remainder decreases. At some step N it must vanish, so that

$$\begin{aligned} P(N) := g &= \gcd(r_{N-1}, r_N) && (5.45)\\ &= \gcd(r_{N-1}, 0) && (5.46)\\ &= r_{N-1}. && (5.47) \end{aligned}$$

Our induction is proved, and the final nonegative result r_{N-1} is the GCD of a and b.

For illustration, we use the Euclidean algorithm to find the GCD of $a = 1071 = 3 \cdot 3 \cdot 7 \cdot 17$ and $b = 462 = 2 \cdot 3 \cdot 7 \cdot 11$, which should be 21 from comparing the prime factorizations. To begin, multiples of 462 are subtracted from 1071 until the remainder is less than 462. Two such multiples can be subtracted ($q_0 = 2$), leaving a remainder $r_0 = 147$,

$$1071 = 2 \cdot 462 + 147. \tag{5.48}$$

Then multiples of 147 are subtracted from 462 until the remainder is less than 147. Three multiples can be subtracted ($q_1 = 3$), leaving a remainder $r_1 = 21$,

$$462 = 3 \cdot 147 + 21. \tag{5.49}$$

Then multiples of 21 are subtracted from 147 until the remainder is less than 21. Seven multiples can be subtracted ($q_2 = 7$), leaving no remainder $r_2 = 0$,

$$147 = 7 \cdot 21 + 0. \tag{5.50}$$

Since the last remainder is zero, the algorithm ends with 21 as the GCD of 1071 and 462. This agrees with the gcd(1071, 462) found by prime factorization above. In tabular form, the steps are

Step	Equation	Quotient	Remainder
0	$1071 = 2 \cdot 462 + 147$	2	147
1	$462 = 3 \cdot 147 + 21$	3	21
2	$147 = 7 \cdot 21 + 0$	7	0

Note also that we can replace the division step with a direct modulo operation

$$r_n = r_{n-2} \mod r_{n-1}. \tag{5.51}$$

We remember that Bézout's Theorem states that the GCD g of two integers a and b can be represented as a linear sum of the original two numbers, Eq. (5.28),

$$\exists s, t \in \mathbb{Z}, g = sa + tb.$$

The integers s and t of Bézout's Theorem can be computed efficiently using the *Extended Euclidean Algorithm*. The extension adds two recursive equations to original algorithm,

$$s_n = s_{n-2} - q_n s_{n-1} \tag{5.52}$$
$$t_n = t_{n-2} - q_n t_{n-1} \tag{5.53}$$

with the starting values

$$s_{-2} = 1, \qquad s_{-1} = 0 \tag{5.54}$$
$$t_{-2} = 0, \qquad t_{-1} = 1. \tag{5.55}$$

Using this recursion, the integers s and t in Eq. (5.28) are given by $s = s_{N-1}$ and $t = t_{N-1}$, where N is the step on which the algorithm terminates with $r_N = 0$, which we will prove by induction.

Our induction hypothesis $P(n)$ follows from Bézout's Theorem,

$$P(n) := r_n = s_n a + t_n b. \tag{5.56}$$

The base cases $P(-2)$ and $P(-1)$ are true

$$P(-2) := a = 1 \cdot a + 0 \cdot b = \mathrm{T} \tag{5.57}$$
$$P(-1) := b = 0 \cdot a + 1 \cdot b = \mathrm{T} \tag{5.58}$$

Assuming $P(n-1)$ and $P(n)$, the inductive step is

$$\begin{aligned}
P(n+1) := r_{n+1} &= s_{n+1}a + t_{n+1}b, && (5.59)\\
r_{n-1} - q_{n+1}r_n &= (s_{n-1} - q_{n+1}s_n)a + (t_{n-1} - q_{n+1}t_n)b, && (5.60)\\
-q_{n+1}r_n &= (-q_{n+1}s_n)a + (-q_{n+1}t_n)b, && (5.61)\\
r_n &= s_n a + t_n b, && (5.62)\\
&P(n),\\
&\mathrm{T}
\end{aligned}$$

so our induction is successful and the theorem is proved. Note that this type of induction, where multiple prior hypotheses are used, is called *strong induction*.

We can carry out the Extended Euclid Algorithm on our pair of number $(1071, 462)$ from the last example. In tabular form, the steps are

Step	Euclid Equation	q	r	Bézout Equation	s	t
0	$1071 = 2 \cdot 462 + 147$	2	147	$147 = 1 \cdot 1071 + -2 \cdot 462$	1	-2
1	$462 = 3 \cdot 147 + 21$	3	21	$21 = -3 \cdot 1071 + 7 \cdot 462$	-3	7
2	$147 = 7 \cdot 21 + 0$	7	0	N/A	–	–

5.5.1 Binary GCD

An alternative algorithm exists for finding the GCD using only division by two, instead of division by an arbitrary integer, called the *Binary GCD*. It was originally proposed by Stein (Stein 1967), but may have been known in 1st century China (Knuth 1998). The algorithm works recursively by checking a few cases for input (a, b):

1. If $a = 0$ or $b = 0$, return the other argument.
2. If a and b are even, return $2\gcd(a/2, b/2)$ since 2 is a common factor.
3. If only a is even, return $\gcd(a/2, b)$ since 2 is *not* a common factor.
4. If only b is even, return $\gcd(a, b/2)$ since 2 is *not* a common factor.
5. If neither is even, return $\gcd(\frac{a-b}{2}, b)$ is a is larger, or $\gcd(a, \frac{b-a}{2})$.

The GCD is $2^k r_N$, where r_N is the final remainder and k is the number of times we apply Rule 2. The last step works because $a - b$ is even if both numbers are odd, and then we can apply step 3. This algorithm still requires $\mathcal{O}(b_0^2)$ iterations for b-bit input. The runtime improvement has not been very convincing on modern architectures which have fast integer division. We will work through the same example from above for $\gcd(1071, 462)$.

Step	a	b	Rule
0	1071	462	4
1	1071	231	5
2	420	231	3
3	210	231	3
4	105	231	5
5	105	63	5
6	21	63	5
7	21	21	5
8	21	0	—

Since we never applied Rule 2, we have $\gcd(1071, 462) = 21$, which matches our answer from before. Notice that Euclid's Algorithm took 2 steps, whereas the Binary GCD took 8 steps. However, asymptotically both algorithms take a number of steps proportional to the number of input bits squared.

5.5.2 Running Time

We will denote by $T(a, b)$ the number of steps in Euclid's Algorithm to compute their GCD g. Since g is a common divisor, then $a = mg$ and $b = ng$ where m and n are coprime. Then

$$T(a, b) = T(m, n)$$

since we can divide all the steps in the Euclidean algorithm by g. Likewise,

$$T(a,b) = T(na,nb)$$

for any n. We can also derive a recursion for T using the algorithm itself,

$$\begin{aligned} T(a,b) &= 1 + T(b,r_0) \\ &= 2 + T(r_0,r_1) \\ &= \vdots \\ &= N + T(r_{N-2},r_{N-1}), \end{aligned}$$

terminating when we hit $T(n,0) = 0$.

If Euclid's Algorithm requires N steps for a pair of natural numbers $a > b > 0$, the smallest values of a and b for which this is true are the *Fibonacci numbers* F_{N+2} and F_{N+1} (Wikipedia 2019), respectively. More precisely, we will show by induction that if Euclid's Algorithm requires N steps for the pair $a > b$, then one has $a \geq F_{N+2}$ and $b \geq F_{N+1}$.

If $N = 1$, then $r_0 = 0$ or b evenly divides a; the smallest integers for which this is true are $b = 1$ and $a = 2$, meaning $a = F_3$ and $b = F_2$. Now we assume that the result holds for all values up to $N - 1$. The first step of a run taking N steps is $a = q_0 b + r_0$, and then Euclid's Algorithm requires $N - 1$ steps for the pair (b, r_0). By our induction hypothesis, we has $b \geq F_{N+1}$ and $r_0 \geq F_N$. Therefore,

$$\begin{aligned} a &= q_0 b + r_0 \\ &\geq b + r_0 \\ &\geq F_{N+1} + F_N \\ &= F_{N+2} \end{aligned}$$

where we used the Fibonacci relation in the last line, and we have shown our inductive implication. This proof was first published by Gabriel Lamé in 1844.

Now suppose that Euclid's Algorithm takes N steps, so that $b > F_{N+1}$ as we proved above. Using the closed form solution for Fibonacci numbers, we know that

$$\begin{aligned} b &\geq F_{N+1}, \\ &= \left\lfloor \frac{\phi^{N+1}}{\sqrt{5}} + \frac{1}{2} \right\rfloor, \end{aligned}$$

where ϕ is the *Golden Ratio*

$$\phi = \frac{1+\sqrt{5}}{2}.$$

Thus we have

$$\begin{aligned}\log_{10} b &\geq \log_{10} \frac{\phi^{N+1}}{\sqrt{5}} \\ &\geq (N+1)\log_{10}\phi \\ &\geq \frac{N+1}{5}\end{aligned}$$

since $\log_{10}\phi > 1/5$. Thus the number of Euclid iterations is less than five times the number of decimal digits in the smaller argument to the GCD, and Euclid's Algorithm is linear in the size of b.

We know that the number of bit operations required to divide a k-bit positive integer by an l-bit positive integer, $k \geq l$, is $(k-l+1)$. Thus, if we let $b_i = \log_{10} r_i$ be the number of bits in the ith remainder, then the total algorithmic cost is given by

$$\begin{aligned}\sum_{i<N} b_i(b_i - b_{i+1} + 1) &\leq b_0 \sum_{i<N}(b_i - b_{i+1} + 1) \\ &= b_0(b_0 + N) \\ &\leq b_0(b_0 + 5b_0) \\ &\leq 6b_0^2\end{aligned}$$

so that the running time of Euclid's Algorithm is in $\mathcal{O}(b_0^2)$, where b_0 is the number of bits in the input.

5.5.3 Coq Proofs

We would like a constructive realization of the GCD algorithm embodying the Euclidean algorithm. In order to get this inductively, we need some variable to decrease. When we described the Euclidean Algorithm above, we concluded that there was a finite number of steps because the remainder must vanish at some step N. In defining our inductive function, we will instead decrease in our argument n, demanding that we start with an adequate step limit N.

```
Fixpoint Zgcdn (n:nat) : Z -> Z -> Z := fun a b =>
  match n with
    | O => 1 (* arbitrary, since n should be big enough *)
    | S n => match a with
               | Z0 => Zabs b
               | Zpos _ => Zgcdn n (Zmod b a) a
               | Zneg a => Zgcdn n (Zmod b (Zpos a)) (Zpos a)
             end
  end.
```

This is given in the `ZArith.Zcgd_alt` package in Coq. We can give a simple estimate for the number of iterations N. If we start with $a = 0$, then the answer is just $|b|$, so we do one iterate. Otherwise, the number of iterates is less than twice the number of binary digits given by `Psize`, which follows from our bound using the number of decimal digits.

```
Definition Zgcd_bound (a:Z) :=
   match a with
     | Z0 => S O
     | Zpos p => let n := Psize p in (n+n)%nat
     | Zneg p => let n := Psize p in (n+n)%nat
   end.
```

Using this bound, we can define our complete algorithm

```
Definition Zgcd_alt a b := Zgcdn (Zgcd_bound a) a b.
```

As a demonstration, let us prove that this function is always non-negative, assuring us that our definition for arbitrary integers is correct.

```
Coq < Lemma Zgcdn_pos : forall n a b, 0 <= Zgcdn n a b.
1 subgoal

  ============================
  forall (n : nat) (a b : Z), 0 <= Zgcdn n a b
```

We will do induction on n

```
Zgcdn_pos < induction n.
2 subgoals

  ============================
  forall a b : Z, 0 <= Zgcdn 0 a b

subgoal 2 is:
 forall a b : Z, 0 <= Zgcdn (S n) a b

Zgcdn_pos < intros a b.
2 subgoals

  a, b : Z
  ============================
  0 <= Zgcdn 0 a b

subgoal 2 is:
 forall a b : Z, 0 <= Zgcdn (S n) a b
```

and take care of the base case with `auto`, although we can do this by hand using the theorems for the less than operator.

```
Zgcdn_pos < simpl.
2 subgoals

  a, b : Z
  ============================
  0 <= 1

subgoal 2 is:
 forall a b : Z, 0 <= Zgcdn (S n) a b

Zgcdn_pos < auto with zarith.
1 subgoal

  n : nat
  IHn : forall a b : Z, 0 <= Zgcdn n a b
  ============================
  forall a b : Z, 0 <= Zgcdn (S n) a b
```

Now we can do induction on a, which will give us the three case for integers, namely 0, positive, and negative. This can also be obtained using `destruct a` in the proof assistant.

```
Zgcdn_pos < intros a b.
1 subgoal

  n : nat
  IHn : forall a b : Z, 0 <= Zgcdn n a b
  a, b : Z
  ============================
  0 <= Zgcdn (S n) a b

Zgcdn_pos < induction a.
3 subgoals

  n : nat
  IHn : forall a b : Z, 0 <= Zgcdn n a b
  b : Z
  ============================
  0 <= Zgcdn (S n) 0 b

subgoal 2 is:
 0 <= Zgcdn (S n) (Z.pos p) b
subgoal 3 is:
 0 <= Zgcdn (S n) (Z.neg p) b
```

The zero case amounts to showing that the absolute value of a number is non-negative, which `auto` can handle.

```
Zgcdn_pos < simpl.
3 subgoals

  n : nat
  IHn : forall a b : Z, 0 <= Zgcdn n a b
  b : Z
  ============================
  0 <= Z.abs b

subgoal 2 is:
 0 <= Zgcdn (S n) (Z.pos p) b
subgoal 3 is:
 0 <= Zgcdn (S n) (Z.neg p) b

Zgcdn_pos < auto with zarith.
2 subgoals

  n : nat
  IHn : forall a b : Z, 0 <= Zgcdn n a b
  p : positive
  b : Z
  ============================
  0 <= Zgcdn (S n) (Z.pos p) b

subgoal 2 is:
 0 <= Zgcdn (S n) (Z.neg p) b
```

For the positive case, we can expand the inductive definition one step, which gives us something that exactly fits our induction hypothesis.

```
Zgcdn_pos < simpl.
2 subgoals

  n : nat
  IHn : forall a b : Z, 0 <= Zgcdn n a b
  p : positive
  b : Z
  ============================
  0 <= Zgcdn n (b mod Z.pos p) (Z.pos p)

subgoal 2 is:
 0 <= Zgcdn (S n) (Z.neg p) b

Zgcdn_pos < apply IHn.
1 subgoal

  n : nat
  IHn : forall a b : Z, 0 <= Zgcdn n a b
  p : positive
  b : Z
  ============================
  0 <= Zgcdn (S n) (Z.neg p) b
```

The negative case is handled identically, completing our proof.

```
Zgcdn_pos < simpl.
1 subgoal

  n : nat
  IHn : forall a b : Z, 0 <= Zgcdn n a b
  p : positive
  b : Z
  ============================
  0 <= Zgcdn n (b mod Z.pos p) (Z.pos p)

Zgcdn_pos < apply IHn.
No more subgoals.

Zgcdn_pos < Qed.
(induction n).
 (intros a b).
 (simpl).
 auto with zarith.

 (intros a b).
 (induction a).
  (simpl).
  auto with zarith.

  (simpl).
  (apply IHn).

  (simpl).
  (apply IHn).

Qed.
Zgcdn_pos is defined
```

We can use our proof above to justify the result for our full algorithm.

```
Coq < Lemma Zgcd_alt_pos : forall a b, 0 <= Zgcd_alt a b.
1 subgoal

  ============================
  forall a b : Z, 0 <= Zgcd_alt a b

Zgcd_alt_pos < intros a b.
1 subgoal

  a, b : Z
  ============================
  0 <= Zgcd_alt a b

Zgcd_alt_pos < unfold Zgcd_alt.
1 subgoal

  a, b : Z
  ============================
  0 <= Zgcdn (Zgcd_bound a) a b
```

```
Zgcd_alt_pos < apply Zgcdn_pos.
No more subgoals.

Zgcd_alt_pos < Qed.
(intros a b).
(unfold Zgcd_alt).
(apply Zgcdn_pos).

Qed.
Zgcd_alt_pos is defined
```

Now we are ready for an initial verification of our algorithm. We would like to prove that the output of our function is indeed the gcd of the two inputs. To verify this, we can use the `Zis_gcd` predicate we saw in Section 5.4.2. We also need to bound the number of steps the algorithm will take. We know from our complexity estimate that this is bounded by the number of bits in a, but here we will use the much weaker bound a since it is easier to prove. We begin with a statement of the theorem, and use induction on n.

```
Coq < Lemma Zgcdn_linear_bound : forall n a b, Zabs a < Z_of_nat n -> Zis_gcd a b (Zgcdn n a b).
1 subgoal

  ============================
  forall (n : nat) (a b : Z),
  Z.abs a < Z.of_nat n -> Zis_gcd a b (Zgcdn n a b)

Zgcdn_linear_bound < induction n.
2 subgoals

  ============================
  forall a b : Z, Z.abs a < Z.of_nat 0 -> Zis_gcd a b (Zgcdn 0 a b)

subgoal 2 is:
 forall a b : Z, Z.abs a < Z.of_nat (S n) -> Zis_gcd a b (Zgcdn (S n) a b)
```

The base case produces a hypothesis H which is impossible, $|a| < 0$. Since a false antecedent proves anything, we can replace our goal with `F` using the tactic `exfalso`. This comes from the Latin phrase *ex falso quodlibet* which means "from falsehood, anything follows", first proved by 12th century French philosopher William of Soissons (Priest 2011).

```
Zgcdn_linear_bound < intros a b H.
2 subgoals

  a, b : Z
  H : Z.abs a < Z.of_nat 0
  ============================
  Zis_gcd a b (Zgcdn 0 a b)

subgoal 2 is:
 forall a b : Z, Z.abs a < Z.of_nat (S n) -> Zis_gcd a b (Zgcdn (S n) a b)

Zgcdn_linear_bound < simpl in H.
2 subgoals

  a, b : Z
  H : Z.abs a < 0
  ============================
  Zis_gcd a b (Zgcdn 0 a b)

subgoal 2 is:
 forall a b : Z, Z.abs a < Z.of_nat (S n) -> Zis_gcd a b (Zgcdn (S n) a b)
```

```
Zgcdn_linear_bound < exfalso.
2 subgoals

  a, b : Z
  H : Z.abs a < 0
  ============================
  False

subgoal 2 is:
 forall a b : Z, Z.abs a < Z.of_nat (S n) -> Zis_gcd a b (Zgcdn (S n) a b)
```

Now we need a way to expose this as a contradiction. First, we introduce a new hypothesis $H1$ which tells us that the absolute value of any integer must be non-negative. Then we can use the built-in solver `omega` for Pressburger arithmetic, which can handle integer math and inequalities.

```
Zgcdn_linear_bound < pose (H1 := Zabs_pos a).
2 subgoals

  a, b : Z
  H : Z.abs a < 0
  H1 := Z.abs_nonneg a : 0 <= Z.abs a
  ============================
  False

subgoal 2 is:
 forall a b : Z, Z.abs a < Z.of_nat (S n) -> Zis_gcd a b (Zgcdn (S n) a b)

Zgcdn_linear_bound < omega.
1 subgoal

  n : nat
  IHn : forall a b : Z, Z.abs a < Z.of_nat n -> Zis_gcd a b (Zgcdn n a b)
  ============================
  forall a b : Z, Z.abs a < Z.of_nat (S n) -> Zis_gcd a b (Zgcdn (S n) a b)
```

Now we have to prove the induction step, for which we will use induction on the input a. Remember that induction over integers amounts to proving the statement for zero, positive numbers, and negative numbers. We start with the zero case.

```
Zgcdn_linear_bound < intros a b H.
1 subgoal

  n : nat
  IHn : forall a b : Z, Z.abs a < Z.of_nat n -> Zis_gcd a b (Zgcdn n a b)
  a, b : Z
  H : Z.abs a < Z.of_nat (S n)
  ============================
  Zis_gcd a b (Zgcdn (S n) a b)

Zgcdn_linear_bound < induction a.
3 subgoals

  n : nat
  IHn : forall a b : Z, Z.abs a < Z.of_nat n -> Zis_gcd a b (Zgcdn n a b)
  b : Z
  H : Z.abs 0 < Z.of_nat (S n)
  ============================
  Zis_gcd 0 b (Zgcdn (S n) 0 b)

subgoal 2 is:
 Zis_gcd (Z.pos p) b (Zgcdn (S n) (Z.pos p) b)
subgoal 3 is:
 Zis_gcd (Z.neg p) b (Zgcdn (S n) (Z.neg p) b)

Zgcdn_linear_bound < simpl.
3 subgoals

  n : nat
  IHn : forall a b : Z, Z.abs a < Z.of_nat n -> Zis_gcd a b (Zgcdn n a b)
  b : Z
  H : Z.abs 0 < Z.of_nat (S n)
  ============================
  Zis_gcd 0 b (Z.abs b)

subgoal 2 is:
 Zis_gcd (Z.pos p) b (Zgcdn (S n) (Z.pos p) b)
subgoal 3 is:
 Zis_gcd (Z.neg p) b (Zgcdn (S n) (Z.neg p) b)
```

We will expand the gcd predicate using the theorem we saw in the last section, which gives us three new goals. The first divisibility statement just needs a witness, which is clearly zero.

```
Zgcdn_linear_bound < apply Zis_gcd_intro.
5 subgoals

  n : nat
  IHn : forall a b : Z, Z.abs a < Z.of_nat n -> Zis_gcd a b (Zgcdn n a b)
  b : Z
  H : Z.abs 0 < Z.of_nat (S n)
  ============================
  (Z.abs b | 0)

subgoal 2 is:
 (Z.abs b | b)
subgoal 3 is:
 forall x : Z, (x | 0) -> (x | b) -> (x | Z.abs b)
subgoal 4 is:
 Zis_gcd (Z.pos p) b (Zgcdn (S n) (Z.pos p) b)
subgoal 5 is:
 Zis_gcd (Z.neg p) b (Zgcdn (S n) (Z.neg p) b)

Zgcdn_linear_bound < exists 0.
5 subgoals

  n : nat
  IHn : forall a b : Z, Z.abs a < Z.of_nat n -> Zis_gcd a b (Zgcdn n a b)
  b : Z
  H : Z.abs 0 < Z.of_nat (S n)
  ============================
  0 = 0 * Z.abs b

subgoal 2 is:
 (Z.abs b | b)
subgoal 3 is:
 forall x : Z, (x | 0) -> (x | b) -> (x | Z.abs b)
subgoal 4 is:
 Zis_gcd (Z.pos p) b (Zgcdn (S n) (Z.pos p) b)
subgoal 5 is:
 Zis_gcd (Z.neg p) b (Zgcdn (S n) (Z.neg p) b)

Zgcdn_linear_bound < ring.
4 subgoals

  n : nat
  IHn : forall a b : Z, Z.abs a < Z.of_nat n -> Zis_gcd a b (Zgcdn n a b)
  b : Z
  H : Z.abs 0 < Z.of_nat (S n)
  ============================
  (Z.abs b | b)

subgoal 2 is:
 forall x : Z, (x | 0) -> (x | b) -> (x | Z.abs b)
subgoal 3 is:
 Zis_gcd (Z.pos p) b (Zgcdn (S n) (Z.pos p) b)
subgoal 4 is:
 Zis_gcd (Z.neg p) b (Zgcdn (S n) (Z.neg p) b)
```

The second statement asks for m such that $m|b| = b$, so it is just the sign of b. It takes a few more steps until we can apply a theorem telling us that this is true.

```
Zgcdn_linear_bound < exists (Zsgn b).
4 subgoals

  n : nat
  IHn : forall a b : Z, Z.abs a < Z.of_nat n -> Zis_gcd a b (Zgcdn n a b)
  b : Z
  H : Z.abs 0 < Z.of_nat (S n)
  ============================
  b = Z.sgn b * Z.abs b

subgoal 2 is:
 forall x : Z, (x | 0) -> (x | b) -> (x | Z.abs b)
subgoal 3 is:
 Zis_gcd (Z.pos p) b (Zgcdn (S n) (Z.pos p) b)
subgoal 4 is:
 Zis_gcd (Z.neg p) b (Zgcdn (S n) (Z.neg p) b)

Zgcdn_linear_bound < symmetry.
4 subgoals

  n : nat
  IHn : forall a b : Z, Z.abs a < Z.of_nat n -> Zis_gcd a b (Zgcdn n a b)
  b : Z
  H : Z.abs 0 < Z.of_nat (S n)
  ============================
  Z.sgn b * Z.abs b = b

subgoal 2 is:
 forall x : Z, (x | 0) -> (x | b) -> (x | Z.abs b)
subgoal 3 is:
 Zis_gcd (Z.pos p) b (Zgcdn (S n) (Z.pos p) b)
subgoal 4 is:
 Zis_gcd (Z.neg p) b (Zgcdn (S n) (Z.neg p) b)

Zgcdn_linear_bound < rewrite Z.mul_comm.
4 subgoals

  n : nat
  IHn : forall a b : Z, Z.abs a < Z.of_nat n -> Zis_gcd a b (Zgcdn n a b)
  b : Z
  H : Z.abs 0 < Z.of_nat (S n)
  ============================
  Z.abs b * Z.sgn b = b

subgoal 2 is:
 forall x : Z, (x | 0) -> (x | b) -> (x | Z.abs b)
subgoal 3 is:
 Zis_gcd (Z.pos p) b (Zgcdn (S n) (Z.pos p) b)
subgoal 4 is:
 Zis_gcd (Z.neg p) b (Zgcdn (S n) (Z.neg p) b)

Zgcdn_linear_bound < apply Z.abs_sgn.
3 subgoals
```

```
  n : nat
  IHn : forall a b : Z, Z.abs a < Z.of_nat n -> Zis_gcd a b (Zgcdn n a b)
  b : Z
  H : Z.abs 0 < Z.of_nat (S n)
  ============================
  forall x : Z, (x | 0) -> (x | b) -> (x | Z.abs b)

subgoal 2 is:
 Zis_gcd (Z.pos p) b (Zgcdn (S n) (Z.pos p) b)
subgoal 3 is:
 Zis_gcd (Z.neg p) b (Zgcdn (S n) (Z.neg p) b)
```

Now we have to prove the defining implication for the gcd, namely that it is divisible by any other common divisor. The main complication here is from the sign ambiguity. For the first step, we know that $b = kx$ for some k, so that $|b| = \mathrm{sgn}(b)kx$.

```
Zgcdn_linear_bound < intros x H1 H2.
3 subgoals

  n : nat
  IHn : forall a b : Z, Z.abs a < Z.of_nat n -> Zis_gcd a b (Zgcdn n a b)
  b : Z
  H : Z.abs 0 < Z.of_nat (S n)
  x : Z
  H1 : (x | 0)
  H2 : (x | b)
  ============================
  (x | Z.abs b)

subgoal 2 is:
 Zis_gcd (Z.pos p) b (Zgcdn (S n) (Z.pos p) b)
subgoal 3 is:
 Zis_gcd (Z.neg p) b (Zgcdn (S n) (Z.neg p) b)

Zgcdn_linear_bound < destruct H2 as [k H2].
3 subgoals

  n : nat
  IHn : forall a b : Z, Z.abs a < Z.of_nat n -> Zis_gcd a b (Zgcdn n a b)
  b : Z
  H : Z.abs 0 < Z.of_nat (S n)
  x : Z
  H1 : (x | 0)
  k : Z
  H2 : b = k * x
  ============================
  (x | Z.abs b)

subgoal 2 is:
 Zis_gcd (Z.pos p) b (Zgcdn (S n) (Z.pos p) b)
subgoal 3 is:
 Zis_gcd (Z.neg p) b (Zgcdn (S n) (Z.neg p) b)

Zgcdn_linear_bound < exists (Z.sgn b * k).
3 subgoals

  n : nat
  IHn : forall a b : Z, Z.abs a < Z.of_nat n -> Zis_gcd a b (Zgcdn n a b)
  b : Z
  H : Z.abs 0 < Z.of_nat (S n)
  x : Z
  H1 : (x | 0)
  k : Z
  H2 : b = k * x
  ============================
  Z.abs b = Z.sgn b * k * x
```

```
subgoal 2 is:
 Zis_gcd (Z.pos p) b (Zgcdn (S n) (Z.pos p) b)
subgoal 3 is:
 Zis_gcd (Z.neg p) b (Zgcdn (S n) (Z.neg p) b)

Zgcdn_linear_bound < rewrite <- Z.mul_assoc.
3 subgoals

  n : nat
  IHn : forall a b : Z, Z.abs a < Z.of_nat n -> Zis_gcd a b (Zgcdn n a b)
  b : Z
  H : Z.abs 0 < Z.of_nat (S n)
  x : Z
  H1 : (x | 0)
  k : Z
  H2 : b = k * x
  ============================
  Z.abs b = Z.sgn b * (k * x)

subgoal 2 is:
 Zis_gcd (Z.pos p) b (Zgcdn (S n) (Z.pos p) b)
subgoal 3 is:
 Zis_gcd (Z.neg p) b (Zgcdn (S n) (Z.neg p) b)

Zgcdn_linear_bound < rewrite <- H2.
3 subgoals

  n : nat
  IHn : forall a b : Z, Z.abs a < Z.of_nat n -> Zis_gcd a b (Zgcdn n a b)
  b : Z
  H : Z.abs 0 < Z.of_nat (S n)
  x : Z
  H1 : (x | 0)
  k : Z
  H2 : b = k * x
  ============================
  Z.abs b = Z.sgn b * b

subgoal 2 is:
 Zis_gcd (Z.pos p) b (Zgcdn (S n) (Z.pos p) b)
subgoal 3 is:
 Zis_gcd (Z.neg p) b (Zgcdn (S n) (Z.neg p) b)

Zgcdn_linear_bound < symmetry.
3 subgoals

  n : nat
  IHn : forall a b : Z, Z.abs a < Z.of_nat n -> Zis_gcd a b (Zgcdn n a b)
  b : Z
  H : Z.abs 0 < Z.of_nat (S n)
  x : Z
  H1 : (x | 0)
  k : Z
  H2 : b = k * x
  ============================
  Z.sgn b * b = Z.abs b

subgoal 2 is:
 Zis_gcd (Z.pos p) b (Zgcdn (S n) (Z.pos p) b)
subgoal 3 is:
 Zis_gcd (Z.neg p) b (Zgcdn (S n) (Z.neg p) b)

Zgcdn_linear_bound < rewrite Z.mul_comm.
3 subgoals

  n : nat
  IHn : forall a b : Z, Z.abs a < Z.of_nat n -> Zis_gcd a b (Zgcdn n a b)
  b : Z
  H : Z.abs 0 < Z.of_nat (S n)
  x : Z
  H1 : (x | 0)
  k : Z
  H2 : b = k * x
  ============================
  b * Z.sgn b = Z.abs b

subgoal 2 is:
 Zis_gcd (Z.pos p) b (Zgcdn (S n) (Z.pos p) b)
subgoal 3 is:
 Zis_gcd (Z.neg p) b (Zgcdn (S n) (Z.neg p) b)

Zgcdn_linear_bound < apply Z.sgn_abs.
2 subgoals

  n : nat
  IHn : forall a b : Z, Z.abs a < Z.of_nat n -> Zis_gcd a b (Zgcdn n a b)
  p : positive
  b : Z
  H : Z.abs (Z.pos p) < Z.of_nat (S n)
  ============================
  Zis_gcd (Z.pos p) b (Zgcdn (S n) (Z.pos p) b)

subgoal 2 is:
 Zis_gcd (Z.neg p) b (Zgcdn (S n) (Z.neg p) b)
```

This completes our proof for the zero case. Now we will deal with the positive integers. We execute the first step of our Euclidean Algorithm, replacing (b, a) with $(a, b \bmod a)$. Now we would like to replace the modulo operator with an explicit representation in terms of the remainder. To do this, we use the definition of modulo and integer division, employing the `generalize` tactic, which like `pose`, helps us create an additional hypothesis.

```
Zgcdn_linear_bound < simpl.
2 subgoals

  n : nat
  IHn : forall a b : Z, Z.abs a < Z.of_nat n -> Zis_gcd a b (Zgcdn n a b)
  p : positive
  b : Z
  H : Z.abs (Z.pos p) < Z.of_nat (S n)
  ============================
  Zis_gcd (Z.pos p) b (Zgcdn n (b mod Z.pos p) (Z.pos p))

subgoal 2 is:
 Zis_gcd (Z.neg p) b (Zgcdn (S n) (Z.neg p) b)

Zgcdn_linear_bound < unfold Zmod.
2 subgoals

  n : nat
  IHn : forall a b : Z, Z.abs a < Z.of_nat n -> Zis_gcd a b (Zgcdn n a b)
  p : positive
  b : Z
  H : Z.abs (Z.pos p) < Z.of_nat (S n)
  ============================
  Zis_gcd (Z.pos p) b
    (Zgcdn n (let (_, r) := Z.div_eucl b (Z.pos p) in r) (Z.pos p))

subgoal 2 is:
 Zis_gcd (Z.neg p) b (Zgcdn (S n) (Z.neg p) b)

Zgcdn_linear_bound < generalize (Z_div_mod b (Zpos p) (refl_equal Gt)).
2 subgoals
```

```
  n : nat
  IHn : forall a b : Z, Z.abs a < Z.of_nat n -> Zis_gcd a b (Zgcdn n a b)
  p : positive
  b : Z
  H : Z.abs (Z.pos p) < Z.of_nat (S n)
  ============================
  (let (q, r) := Z.div_eucl b (Z.pos p) in
   b = Z.pos p * q + r /\ 0 <= r < Z.pos p) ->
  Zis_gcd (Z.pos p) b
    (Zgcdn n (let (_, r) := Z.div_eucl b (Z.pos p) in r) (Z.pos p))

subgoal 2 is:
 Zis_gcd (Z.neg p) b (Zgcdn (S n) (Z.neg p) b)

Zgcdn_linear_bound < destruct (Zdiv_eucl b (Zpos p)) as (q, r).
2 subgoals

  n : nat
  IHn : forall a b : Z, Z.abs a < Z.of_nat n -> Zis_gcd a b (Zgcdn n a b)
  p : positive
  b : Z
  H : Z.abs (Z.pos p) < Z.of_nat (S n)
  q, r : Z
  ============================
  b = Z.pos p * q + r /\ 0 <= r < Z.pos p ->
  Zis_gcd (Z.pos p) b (Zgcdn n r (Z.pos p))

subgoal 2 is:
 Zis_gcd (Z.neg p) b (Zgcdn (S n) (Z.neg p) b)

Zgcdn_linear_bound < intros (H0,H1).
2 subgoals

  n : nat
  IHn : forall a b : Z, Z.abs a < Z.of_nat n -> Zis_gcd a b (Zgcdn n a b)
  p : positive
  b : Z
  H : Z.abs (Z.pos p) < Z.of_nat (S n)
  q, r : Z
  H0 : b = Z.pos p * q + r
  H1 : 0 <= r < Z.pos p
  ============================
  Zis_gcd (Z.pos p) b (Zgcdn n r (Z.pos p))

subgoal 2 is:
 Zis_gcd (Z.neg p) b (Zgcdn (S n) (Z.neg p) b)
```

Now we need to handle the $S(n)$ for our step bound in H. Since $p < n+1$ and $r < p$, we know that $r < n$. This can be proved automatically by Coq using the inequalities in our hypotheses.

```
Zgcdn_linear_bound < rewrite inj_S in H.
2 subgoals

  n : nat
  IHn : forall a b : Z, Z.abs a < Z.of_nat n -> Zis_gcd a b (Zgcdn n a b)
  p : positive
  b : Z
  H : Z.abs (Z.pos p) < Z.succ (Z.of_nat n)
  q, r : Z
  H0 : b = Z.pos p * q + r
  H1 : 0 <= r < Z.pos p
  ============================
  Zis_gcd (Z.pos p) b (Zgcdn n r (Z.pos p))

subgoal 2 is:
 Zis_gcd (Z.neg p) b (Zgcdn (S n) (Z.neg p) b)

Zgcdn_linear_bound < simpl Zabs in H.
2 subgoals

  n : nat
  IHn : forall a b : Z, Z.abs a < Z.of_nat n -> Zis_gcd a b (Zgcdn n a b)
  p : positive
  b : Z
  H : Z.pos p < Z.succ (Z.of_nat n)
  q, r : Z
  H0 : b = Z.pos p * q + r
  H1 : 0 <= r < Z.pos p
  ============================
  Zis_gcd (Z.pos p) b (Zgcdn n r (Z.pos p))

subgoal 2 is:
 Zis_gcd (Z.neg p) b (Zgcdn (S n) (Z.neg p) b)

Zgcdn_linear_bound < assert (H2 : Zabs r < Z_of_nat n).
3 subgoals

  n : nat
  IHn : forall a b : Z, Z.abs a < Z.of_nat n -> Zis_gcd a b (Zgcdn n a b)
  p : positive
  b : Z
  H : Z.pos p < Z.succ (Z.of_nat n)
  q, r : Z
  H0 : b = Z.pos p * q + r
  H1 : 0 <= r < Z.pos p
  ============================
  Z.abs r < Z.of_nat n

subgoal 2 is:
 Zis_gcd (Z.pos p) b (Zgcdn n r (Z.pos p))
subgoal 3 is:
 Zis_gcd (Z.neg p) b (Zgcdn (S n) (Z.neg p) b)

Zgcdn_linear_bound < rewrite Zabs_eq.
4 subgoals

  n : nat
  IHn : forall a b : Z, Z.abs a < Z.of_nat n -> Zis_gcd a b (Zgcdn n a b)
  p : positive
  b : Z
  H : Z.pos p < Z.succ (Z.of_nat n)
  q, r : Z
  H0 : b = Z.pos p * q + r
  H1 : 0 <= r < Z.pos p
  ============================
  r < Z.of_nat n

Zgcdn_linear_bound < auto with zarith.
3 subgoals

  n : nat
  IHn : forall a b : Z, Z.abs a < Z.of_nat n -> Zis_gcd a b (Zgcdn n a b)
  p : positive
  b : Z
  H : Z.pos p < Z.succ (Z.of_nat n)
  q, r : Z
  H0 : b = Z.pos p * q + r
  H1 : 0 <= r < Z.pos p
  ============================
  0 <= r

Zgcdn_linear_bound < auto with zarith.
2 subgoals
```

```
  n : nat
  IHn : forall a b : Z, Z.abs a < Z.of_nat n -> Zis_gcd a b (Zgcdn n a b)
  p : positive
  b : Z
  H : Z.pos p < Z.succ (Z.of_nat n)
  q, r : Z
  H0 : b = Z.pos p * q + r
  H1 : 0 <= r < Z.pos p
  H2 : Z.abs r < Z.of_nat n
  ============================
  Zis_gcd (Z.pos p) b (Zgcdn n r (Z.pos p))
```

Now we are at the last step, and ready to use our induction hypothesis. We evaluate the hypothesis for $b = a$, $a = r$, and use the fact that $r < n$ from $H2$. After rewriting b using $H0$ from our integer division, we can employ the ring identity for gcd, Equation 5.27, which is `Zis_gcd_for_euclid2` in Coq.

```
Zgcdn_linear_bound < pose (IH := (IHn r (Zpos p) H2)).
2 subgoals

  n : nat
  IHn : forall a b : Z, Z.abs a < Z.of_nat n -> Zis_gcd a b (Zgcdn n a b)
  p : positive
  b : Z
  H : Z.pos p < Z.succ (Z.of_nat n)
  q, r : Z
  H0 : b = Z.pos p * q + r
  H1 : 0 <= r < Z.pos p
  H2 : Z.abs r < Z.of_nat n
  IH := IHn r (Z.pos p) H2 : Zis_gcd r (Z.pos p) (Zgcdn n r (Z.pos p))
  ============================
  Zis_gcd (Z.pos p) b (Zgcdn n r (Z.pos p))

subgoal 2 is:
 Zis_gcd (Z.neg p) b (Zgcdn (S n) (Z.neg p) b)

Zgcdn_linear_bound < rewrite H0.
2 subgoals

  n : nat
  IHn : forall a b : Z, Z.abs a < Z.of_nat n -> Zis_gcd a b (Zgcdn n a b)
  p : positive
  b : Z
  H : Z.pos p < Z.succ (Z.of_nat n)
  q, r : Z
  H0 : b = Z.pos p * q + r
  H1 : 0 <= r < Z.pos p
  H2 : Z.abs r < Z.of_nat n
  IH := IHn r (Z.pos p) H2 : Zis_gcd r (Z.pos p) (Zgcdn n r (Z.pos p))
  ============================
  Zis_gcd (Z.pos p) (Z.pos p * q + r) (Zgcdn n r (Z.pos p))

Zgcdn_linear_bound < apply Zis_gcd_for_euclid2.
2 subgoals

  n : nat
  IHn : forall a b : Z, Z.abs a < Z.of_nat n -> Zis_gcd a b (Zgcdn n a b)
  p : positive
  b : Z
  H : Z.pos p < Z.succ (Z.of_nat n)
  q, r : Z
  H0 : b = Z.pos p * q + r
  H1 : 0 <= r < Z.pos p
  H2 : Z.abs r < Z.of_nat n
  IH := IHn r (Z.pos p) H2 : Zis_gcd r (Z.pos p) (Zgcdn n r (Z.pos p))
  ============================
  Zis_gcd r (Z.pos p) (Zgcdn n r (Z.pos p))

subgoal 2 is:
 Zis_gcd (Z.neg p) b (Zgcdn (S n) (Z.neg p) b)

Zgcdn_linear_bound < exact IH.
1 subgoal

  n : nat
  IHn : forall a b : Z, Z.abs a < Z.of_nat n -> Zis_gcd a b (Zgcdn n a b)
  p : positive
  b : Z
  H : Z.abs (Z.neg p) < Z.of_nat (S n)
  ============================
  Zis_gcd (Z.neg p) b (Zgcdn (S n) (Z.neg p) b)
```

We perform nearly the same series of steps for the negative numbers as for the positive. The only difference is that we need to change the negative argument to positive in the gcd call using `Zis_gcd_minus` and the symmetry of the gcd.

```
Zgcdn_linear_bound < Qed.
(induction n).
 (intros a b H).
 (simpl in H).
 exfalso.
 (pose (H1 := Zabs_pos a)).
 omega.

 (intros a b H).
 (induction a).
  (simpl).
  (apply Zis_gcd_intro).
   exists 0.
   ring.

   exists (Zsgn b).
   symmetry.
   (rewrite Z.mul_comm).
   (apply Z.abs_sgn).

   (intros x H1 H2).
   (destruct H2 as [k H2]).
   exists (Z.sgn b * k).
   (rewrite <- Z.mul_assoc).
   (rewrite <- H2).
   symmetry.
   (rewrite Z.mul_comm).
   (apply Z.sgn_abs).

  (simpl).
  (unfold Zmod).
  (generalize (Z_div_mod b (Zpos p) (refl_equal Gt))).
  (destruct (Zdiv_eucl b (Zpos p)) as (q, r)).
  (intros (H0, H1)).
  (rewrite inj_S in H).
  (simpl Zabs in H).
  (assert (H2 : Zabs r < Z_of_nat n)).
   (rewrite Zabs_eq).
    auto with zarith.

    auto with zarith.

   (pose (IH := IHn r (Zpos p) H2)).
   (rewrite H0).
   (apply Zis_gcd_for_euclid2).
   exact IH.
  (simpl).
  (unfold Zmod).
  (generalize (Z_div_mod b (Zpos p) (refl_equal Gt))).
  (destruct (Zdiv_eucl b (Zpos p)) as (q, r)).
  (intros (H0, H1)).
  (rewrite inj_S in H).
  (simpl Zabs in H).
  (assert (H2 : Zabs r < Z_of_nat n)).
   (rewrite Zabs_eq).
    auto with zarith.

    auto with zarith.

   (pose (IH := IHn r (Zpos p) H2)).
   (apply Zis_gcd_minus).
   (apply Zis_gcd_sym).
   (rewrite H0).
   (apply Zis_gcd_for_euclid2).
   exact IH.

Qed.
Zgcdn_linear_bound is defined
```

We have proved that our algorithm does give $\gcd(a, b)$ as its result and takes no more than a steps. However, we know that it should take no more than $5 * \log_{10} a$ steps, so there is considerable improvement to make. In order to do that, we would have to formalize the definition of the Fibonacci numbers.

```
fibonacci_pos < Qed.
(enough (forall N n, (n < N)%nat -> 0 <= fibonacci n)).
 eauto.

 (induction N).
  (inversion 1).

  (intros **).
  (destruct n).
   (simpl).
   auto with zarith.

   (destruct n).
    (simpl).
    auto with zarith.

    (change (0 <= fibonacci (S n) + fibonacci n)).
    (generalize (IHN n), (IHN (S n))).
    omega.

Qed.
fibonacci_pos is defined
```

5.6 The Fundamental Theorem of Arithmetic

As a further example, let us consider *Euclid's Lemma*,

$$n|ab \implies \gcd(n, a) = 1 \implies n|b. \tag{5.63}$$

This can be specialized to the case n prime, as

If n divides ab and n is prime, then n divides at least one of a and b.

We can prove this theorem using Bézout's Theorem from Eq. (5.28),

$$\exists st \in \mathbb{Z},\ sn + ta = 1, \tag{5.64}$$

since $\gcd(n, a) = 1$. Now if we multiply both sides by b

$$\exists st \in \mathbb{Z},\ snb + tab = b, \tag{5.65}$$

we see that n divides both terms on the left, so by Eq. (5.8) it must divide the right, namely b.

We can now use Euclid's Lemma to prove the *Fundamental Theorem of Arithmetic*, namely that all natural numbers have a unique prime factorization. We first need to show that every natural number greater than 1 is either prime or a product of primes, which we will do by induction. For the base case, we note that 2 is prime. For the induction step, we will use strong induction. Assume that all $k \in [2, n)$ have a unique prime factorization. If n is prime, then the unique factorization is n itself. Otherwise,

$$\exists ab \in \mathbb{N},\ n = ab \wedge 1 < a \leq b < n.$$

Using the induction hypotheses, a and b are products of primes such that $a = p_1 p_2 \cdots p_j$ and $b = q_1 q_2 \cdots q_k$. Thus n is a product of primes

$$n = ab = p_1 p_2 \cdots p_j q_1 q_2 \cdots q_k. \tag{5.66}$$

Next we will show the uniqueness of this product. Assume that $n > 1$ is the product of prime numbers in two different ways:

$$\begin{aligned} n &= p_1 p_2 \cdots p_j && (5.67)\\ &= q_1 q_2 \cdots q_k. && (5.68) \end{aligned}$$

Since p_1 divides n, Euclid's Lemma implies that p_1 divides at least one of the q_j. We can permute the numbering so that p_1 divides q_1. However q_1 is prime, so its only divisors are itself and 1. Thus $p_1 = q_1$. We can repeat this argument for all p_i, associating each with q_i. Thus the factorizations are equal. Note that we do not have to worry about different numbers of factors. If we start with the shorter one, since the first j terms are equal, their product must be n, which means the other factors are unity. This is impossible since its a prime factorization, and thus the remaining factors cannot exist.

5.7 The Chinese Remainder Theorem

5.7.1 Divisibility Lemma

Before we discuss the main theorem, we will need a preliminary result. If two integers a and b divide a third integer c, $a|c \wedge b|c$, and they are relatively prime $a \perp b$, then their product also divides it, $ab|c$. This is intuitively clear since relatively prime means they have no prime factors in common. Thus the product of all their prime factors, which is just ab, divides c. However, we can prove this using just our results from above.

$$a|c \implies b|c \implies \gcd(a, b) = 1 \implies ab|c. \tag{5.69}$$

First, we convert the divisibility statements using witnesses,

$$ma = c \implies nb = c \implies \gcd(a, b) = 1 \implies kab = c. \tag{5.70}$$

Now, from substituting into the goal, we know that

$$m = kb \wedge n = ka, \tag{5.71}$$

which implies that

$$k|m \wedge k|n. \tag{5.72}$$

This suggest that we choose as our witness $k = \gcd(m, n)$, or by Bézout's Theorem Eq. (5.28),

$$k = qm + rn. \tag{5.73}$$

Now we can try and evaluate the goal

$$(qm + rn)ab = c, \tag{5.74}$$
$$qmab + rnab = c, \tag{5.75}$$
$$qbc + rac = c, \tag{5.76}$$
$$qb + ra = 1, \tag{5.77}$$

where we used our hypotheses and then divided by c which is not 0. However, this last statement is just Bézout's Theorem for a and b since they are relatively prime.

We will prove this for $c \neq 0$ since it is trivially true when c is zero. We start with a statement of the theorem and introduction of hypotheses.

```
Coq < Lemma div_a_perp_b : forall a b c : Z, c <> 0 -> (a | c) -> (b | c) -> Zis_gcd a b 1 -> (a * b | c).
1 subgoal

  A : Set
  ============================
  forall a b c : Z,
  c <> 0 -> (a | c) -> (b | c) -> Zis_gcd a b 1 -> (a * b | c)

div_a_perp_b < intros a b c CnotZero AdivC BdivC AperpB.
1 subgoal

  A : Set
  a, b, c : Z
  CnotZero : c <> 0
  AdivC : (a | c)
  BdivC : (b | c)
  AperpB : Zis_gcd a b 1
  ============================
  (a * b | c)
```

Next we turn the $\gcd(a, b)$ into a statement of Bézout's Theorem unfold all the divisibility defintions,

```
div_a_perp_b < apply rel_prime_bezout in AperpB.
1 subgoal

  A : Set
  a, b, c : Z
  CnotZero : c <> 0
  AdivC : (a | c)
  BdivC : (b | c)
  AperpB : Bezout a b 1
  ============================
  (a * b | c)

div_a_perp_b < destruct AdivC as [m H1].
1 subgoal

  A : Set
  a, b, c : Z
  CnotZero : c <> 0
  m : Z
  H1 : c = m * a
  BdivC : (b | c)
  AperpB : Bezout a b 1
  ============================
  (a * b | c)

div_a_perp_b < destruct BdivC as [n H2].
1 subgoal

  A : Set
  a, b, c : Z
  CnotZero : c <> 0
  m : Z
  H1 : c = m * a
  n : Z
  H2 : c = n * b
  AperpB : Bezout a b 1
  ============================
  (a * b | c)

div_a_perp_b < unfold Zdivide.
1 subgoal

  A : Set
  a, b, c : Z
  CnotZero : c <> 0
  m : Z
  H1 : c = m * a
  n : Z
  H2 : c = n * b
  AperpB : Bezout a b 1
  ============================
  exists z : Z, c = z * (a * b)

div_a_perp_b < destruct AperpB as [r q H3].
1 subgoal

  A : Set
  a, b, c : Z
  CnotZero : c <> 0
  m : Z
  H1 : c = m * a
  n : Z
  H2 : c = n * b
  r, q : Z
  H3 : r * a + q * b = 1
  ============================
  exists z : Z, c = z * (a * b)
```

Now we multiply the Bézout relation by c so that we can rewrite our goal,

```
div_a_perp_b < apply Z.mul_cancel_r with (p:=c) in H3.
2 subgoals

  a, b, c : Z
  CnotZero : c <> 0
  m : Z
  H1 : c = m * a
  n : Z
  H2 : c = n * b
  r, q : Z
  H3 : (r * a + q * b) * c = 1 * c
  ============================
  exists z : Z, c = z * (a * b)

subgoal 2 is:
 c <> 0

div_a_perp_b < rewrite Z.mul_1_l in H3.
2 subgoals

  a, b, c : Z
  CnotZero : c <> 0
  m : Z
  H1 : c = m * a
  n : Z
  H2 : c = n * b
  r, q : Z
  H3 : (r * a + q * b) * c = c
  ============================
  exists z : Z, c = z * (a * b)

subgoal 2 is:
 c <> 0

div_a_perp_b < rewrite Z.mul_add_distr_r in H3.
2 subgoals

  a, b, c : Z
  CnotZero : c <> 0
  m : Z
  H1 : c = m * a
  n : Z
  H2 : c = n * b
  r, q : Z
  H3 : r * a * c + q * b * c = c
  ============================
  exists z : Z, c = z * (a * b)

subgoal 2 is:
 c <> 0

div_a_perp_b < rewrite H2 in H3 at 1.
2 subgoals

  a, b, c : Z
  CnotZero : c <> 0
  m : Z
  H1 : c = m * a
  n : Z
  H2 : c = n * b
  r, q : Z
  H3 : r * a * (n * b) + q * b * c = c
  ============================
  exists z : Z, c = z * (a * b)

subgoal 2 is:
 c <> 0

div_a_perp_b < rewrite H1 in H3 at 1.
2 subgoals

  a, b, c : Z
  CnotZero : c <> 0
  m : Z
  H1 : c = m * a
  n : Z
  H2 : c = n * b
  r, q : Z
  H3 : r * a * (n * b) + q * b * (m * a) = c
  ============================
  exists z : Z, c = z * (a * b)

subgoal 2 is:
 c <> 0

div_a_perp_b < rewrite <- H3.
2 subgoals

  a, b, c : Z
  CnotZero : c <> 0
  m : Z
  H1 : c = m * a
  n : Z
  H2 : c = n * b
  r, q : Z
  H3 : r * a * (n * b) + q * b * (m * a) = c
  ============================
  exists z : Z, r * a * (n * b) + q * b * (m * a) = z * (a * b)

subgoal 2 is:
 c <> 0
```

Now we can provide the witness to the existence goal

```
div_a_perp_b < exists (q*m + r*n).
2 subgoals

  a, b, c : Z
  CnotZero : c <> 0
  m : Z
  H1 : c = m * a
  n : Z
  H2 : c = n * b
  r, q : Z
  H3 : r * a * (n * b) + q * b * (m * a) = c
  ============================
  r * a * (n * b) + q * b * (m * a) = (q * m + r * n) * (a * b)

subgoal 2 is:
 c <> 0

div_a_perp_b < ring.
1 subgoal

  a, b, c : Z
  CnotZero : c <> 0
  m : Z
  H1 : c = m * a
  n : Z
  H2 : c = n * b
  r, q : Z
  H3 : r * a + q * b = 1
  H : forall n m p : Z, p <> 0 -> n = m -> n * p = m * p
  ============================
  c <> 0

div_a_perp_b < exact CnotZero.
No more subgoals.

div_a_perp_b < Qed.
(intros a b c CnotZero AdivC BdivC AperpB).
(apply rel_prime_bezout in AperpB).
(destruct AdivC as [m H1]).
(destruct BdivC as [n H2]).
(unfold Zdivide).
(destruct AperpB as [r q H3]).
(apply Z.mul_cancel_r with (p := c) in H3).
 (rewrite Z.mul_1_l in H3).
 (rewrite Z.mul_add_distr_r in H3).
 (rewrite H2 in H3 at 1).
 (rewrite H1 in H3 at 1).
 (rewrite <- H3).
 exists (q * m + r * n).
 ring.

 exact CnotZero.

Qed.
div_a_perp_b is defined
```

5.7.2 Ring Isomorphism

The Chinese Remainder Theorem states that if one knows the remainders from Euclidean division of an integer n by several other integers, called moduli or divisors, then one can determine uniquely the remainder from division of n by the product of the divisors, under the condition that the divisors are pairwise coprime. More formally, let the divisors $n_1, \ldots, n_i, \ldots, n_k$ be integers greater than 1. Let us denote by N their product $N = \prod_i n_i$. The *Chinese Remainder Theorem* (CRT) asserts that if the n_i are pairwise coprime, and if the remainders $a_1, \ldots, a_k$ are integers such that $0 \leq a_i < n_i$ for every i, then there is one and only one integer x, such that $0 \leq x < N$ and the remainder of the Euclidean division of x by n_i is a_i for every i. This may be restated as follows in term of congruences: If the n_i are pairwise coprime, and if $a_1, \ldots, a_k$ are any integers, then there exists an integer x such that

$$x = a_1 \pmod{n_1} \tag{5.78}$$

$$\vdots$$

$$x = a_k \pmod{n_k} \tag{5.79}$$

and any two such x are congruent modulo N.

A simple argument shows that any solution to the congruences above is unique. Suppose that x and x' are both solutions to all the congruences. As x and x' give the same remainder when divided by n_i, their difference $x - x'$ is a multiple of each n_i. As the n_i are pairwise coprime, their product N also divides $x - x'$ by Eq. (5.69), and thus x and x' are congruent modulo N. If we assume the x and x' are non-negative and less than N, then their difference may be a multiple of N only if $x = x'$. However, we still need to show that such a solution exists.

Consider the function $f : \mathbb{Z}_N \to \mathbb{Z}_{n_1} \times \cdots \times \mathbb{Z}_{n_k}$

$$f(x) = (x \bmod n_1, \ldots, x \bmod n_k) \tag{5.80}$$

that maps congruence classes modulo N to k-tuples of congruence classes modulo n_i. The proof of uniqueness above shows that the function f is injective. As the domain, integers $0 \le k < N$, and the codomain, tuples $(k_1, \ldots, k_n)$ where $0 \le k_i < n_i$, of f have the same number of elements, the function is also surjective, which proves the existence of the solution. Since the map is bijective, we call the map between the two spaces an *isomorphism*. Unfortunately, this short proof does not tell us how to compute the solution, and is called a *nonconstructive proof*.

5.7.3 Constructive CRT

We will first show a *constructive proof* of CRT in the special case of only two divisors using the same idea we used to prove Eq. (5.69). Suppose that

$$x = a_1 \pmod{n_1} \tag{5.81}$$
$$x = a_2 \pmod{n_2}, \tag{5.82}$$

where $n_1 \perp n_2$ (meaning that they are coprime), and by Bézout's Theorem Eq. (5.28) we have

$$\exists s, t \in \mathbb{Z}, sn_1 + tn_2 = 1. \tag{5.83}$$

We can compute s and t using the Extended Euclidean Algorithm, and form the solution

$$x = a_2 s n_1 + a_1 t n_2. \tag{5.84}$$

We can verify the first congruence

$$x = a_1 \pmod{n_1} \tag{5.85}$$
$$a_2 s n_1 + a_1 t n_2 = a_1 \pmod{n_1} \tag{5.86}$$
$$a_2 s n_1 + a_1(1 - s n_1) = a_1 \pmod{n_1} \tag{5.87}$$
$$(a_2 s - a_1 s) n_1 + a_1 = a_1 \pmod{n_1} \tag{5.88}$$
$$a_1 = a_1 \pmod{n_1} \tag{5.89}$$

and the second

$$x = a_2 \pmod{n_2} \tag{5.90}$$
$$a_2 s n_1 + a_1 t n_2 = a_2 \pmod{n_2} \tag{5.91}$$
$$a_2(1 - t n_2) + a_1 t n_2 = a_2 \pmod{n_2} \tag{5.92}$$
$$a_2 + (a_1 t - a_2 t) n_2 = a_2 \pmod{n_2} \tag{5.93}$$
$$a_2 = a_2 \pmod{n_2} \tag{5.94}$$

Thus x is a solution to the congruences and CRT is true for the case of two congruences. Now we take the general case of k congruences

$$x = a_1 \pmod{n_1} \tag{5.95}$$

$$\vdots \tag{5.96}$$

$$x = a_k \pmod{n_k}, \tag{5.97}$$

where $n_i \perp n_j, i \neq j$. We use our solution algorithm above to solve the first two equations, giving solution a_{12}. Now all solutions of the general equation must also satisfy

$$x = a_{12} \pmod{n_1 n_2}, \tag{5.98}$$

so next we can solve the equation above and the third equation to get a_{123}, giving us

$$x = a_{123} \pmod{n_1 n_2 n_3}. \tag{5.99}$$

Each step of our algorithm removes a congruence. Thus at the end we will have the general solution to k congruences.

A Residue number system (RNS) represents a large integer using a set of smaller integers, so that computation may be performed more efficiently. By representing a large integer as a set of smaller integers, a large calculation can be performed as a series of smaller calculations, each of can be performed independently and in parallel. Addition or subtraction can be performed by simply adding or subtracting the remainder values, modulo their specific moduli, so that

$$C = A \pm B \pmod{N} \equiv c_i = a_i \pm b_i \pmod{n_i}. \tag{5.100}$$

Similarly for multiplication,

$$C = A \cdot B \pmod{N} \equiv c_i = a_i \cdot b_i \pmod{n_i}. \tag{5.101}$$

No overflow checking is necessary for either operation. Division however can be problematic if the denominator is not coprime to the large modulus N. In this case, b_i can be zero, and would not have a multiplicative inverse in the subring $\mathbb{Z}_{n_i}$.

For example, suppose we created a number system using 10 and 21. We would be able to represent numbers up to 210, while only doing calculations with numbers as large as 21. Suppose that we would like to multiply $4 \times 23 = 92$ using our new system. We start by representing our two numbers

$$4 \to (4 \bmod 10, 4 \bmod 21) \tag{5.102}$$

$$= (4, 4) \tag{5.103}$$

$$23 \to (23 \bmod 10, 23 \bmod 21) \tag{5.104}$$

$$= (3, 2) \tag{5.105}$$

Now we can perform the multiplication

$$\begin{aligned} 4 \times 23 &= (4 \times 3 \bmod 10, 4 \times 2 \bmod 21) && (5.106)\\ &= (12 \bmod 10, 8 \bmod 21) && (5.107)\\ &= (2, 8) && (5.108) \end{aligned}$$

We can verify that this is the solution,

$$\begin{aligned} 92 &\to (92 \bmod 10, 92 \bmod 21) && (5.109)\\ &= (2, 8) && (5.110) \end{aligned}$$

but how would we convert $(2, 8)$ directly to 92? We again use Bézout's Theorem Eq. (5.28) to write

$$1 = 21s + 10t \qquad (5.111)$$

which we can solve using the Extended Euclidean Algorithm, giving $s = 1$ and $t = -2$.

Step	Euclid Equation	q	r	Bézout Equation	s	t
0	$21 = 2 \cdot 10 + 1$	2	1	$1 = 1 \cdot 21 + -2 \cdot 10$	1	-2
1	$10 = 10 \cdot 1 + 0$	10	0	N/A	–	–

Now we use Eq. (5.84) for the solution to the Chinese Remainder Theorem congruences

$$\begin{aligned} x &= a_2 t n_1 + a_1 s n_2 \pmod{210} && (5.112)\\ &= 8 \cdot -2 \cdot 10 + 2 \cdot 1 \cdot 21 \pmod{210} && (5.113)\\ &= -160 + 42 \pmod{210} && (5.114)\\ &= -118 \pmod{210} && (5.115)\\ &= 92 && (5.116) \end{aligned}$$

5.8 Problems

Problem V.1 Show that

$$\forall a, b, c \in \mathbb{Z}, a|b \lor a|c \implies a|bc$$

by generating **Proof:** *mod_mult_or*

```
Lemma mod_mult_or : forall a b c : Z, (a|b) \/ (a|c) -> (a|(b*c)).
```

Problem V.2 Show that

$$\forall a, b, c \in \mathbb{Z}, a|b \land a|c \implies a|bc$$

by generating **Proof:** *mod_mult*

```
Lemma mod_mult : forall a b c : Z, (a|b) /\ (a|c) -> (a|(b*c)).
```

Problem V.3 Show that $x = y \mod n$ is an equivalence relation.

Problem V.4 Show that $x|y$ is a partial order on $\mathbb{Z}$ (if $x > 0$), and that it is not a total order.

Problem V.5 Consider the integers 3213 and 1386.

- Show the steps of the Euclidean Algorithm for 3213 and 1386.
- Show the steps of the Exended Euclidean Algorithm for 3213 and 1386.

Problem V.6 A finite field is a set of numbers with four generalized operations. The operations are called addition, subtraction, multiplication and division and have their usual properties, such as commutativity, associativity and distributivity. An example is our sets of integers modulo a prime p, $\mathbb{Z}_p$. In such a field with p numbers, every nonzero element a has a unique modular multiplicative inverse, a^{-1} such that $aa^{-1} = a^{-1}a = 1 \pmod p$. This inverse can be found by solving the congruence equation $ax = 1 \pmod p$, or the equivalent linear Diophantine equation

$$ax + py = 1. \tag{5.117}$$

Suppose that $p = 13$ and $a = 5$. Solve this equation by the Euclidean algorithm and find a^{-1}.

Problem V.7 Notice that as we compute the solution to a set of congruences with Chinese Remainder Theorem, our moduli increase in size at each step. This means that each computation will require numbers of greater size. Can you reorganize the solution process to reduce the size of numbers needed at each step?

Problem V.8 Prove Euclid's Lemma using Coq, generating **Proof:** *Euclid_lemma*

```
Lemma Euclid_lemma : forall n a b : Z, (b <> 0) -> (n | a*b) -> Zis_gcd n a 1 -> (n | b).
```

You will need the `Znumtheory` package, the `Zis_gcd` predicate, and the `rel_prime_bezout` theorem from Coq.

Problem V.9 Prove that the divisibility relation, $m|n$, is antisymmetric for all non-negative integers, rather than all positive integers as was done in the text.

Problem V.10 Let n and m be two integers, $d = \gcd(n, m)$, and $k \in \mathbb{Z}$. If $n \mid mk$, prove that $n \mid kd$, which in Coq is

```
Lemma red_div : forall m n d k: Z, (n | m * k) -> Zis_gcd m n d -> (n | d * k).
```

Problem V.11 Let n and m be two positive integers, $d = \gcd(n, m)$, then $\frac{n}{d}$ and $\frac{m}{d}$ are relatively prime, which in Coq is

```
Lemma red_pair : forall m n d k l : Z, m <> 0 -> n <> 0 -> Zis_gcd m n d -> k * d = n -> l * d = m -> Zis_gcd k l 1.
```

References

Stein, Josef (1967). "Computational problems associated with Racah algebra". In: *Journal of Computational Physics* 1.3, pp. 397–405. DOI: 10.1016/0021-9991(67)90047-2.

Knuth, Donald E. (1998). *Art of Computer Programming, Volume 2: Seminumerical Algorithms.* Addison-Wesley Professional. ISBN: 978-0-201-89684-8.

Wikipedia (2019). *Fibonacci Number.* https://en.wikipedia.org/wiki/Fibonacci_number. URL: https://en.wikipedia.org/wiki/Fibonacci_number.

Priest, Graham (2011). "What's so bad about contradictions?" In: *The Law of Non-Contradicton.* Ed. by Priest, Beal, and Armour-Garb. Clarendon Press, Oxford, p. 25.

Chapter 6

Graph Theory

6.1 Graphs

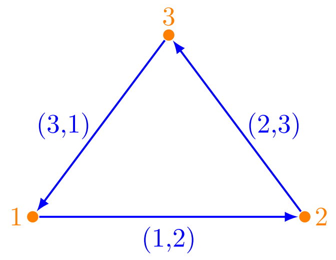
A *graph* G is really just a binary relation E on a set of vertices V,

$$G = (V, E) \tag{6.1}$$

and thus it is no surprise that we called the set of true pairs the graph of our relation. In graph theory, each true pair $v_a G v_b$ is called an *edge* of the graph, starting on vertex v_a and ending on vertex v_b. We can thus draw the relation by making each member of V a point and then adding a directed edge between points for each true pair. For example, consider the following graph on $V = \{1, 2, 3\}$,

$$G = (\{1, 2, 3\}, \{(1, 2), (2, 3), (3, 1)\}) \tag{6.2}$$

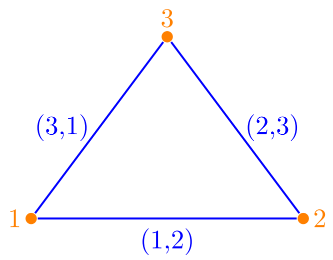
which we can represent as



If the relation is symmetric, we say that the graph is *undirected* since the edges are no longer arrows, just lines, as we always have an edge in both directions. If we make our relation symmetric,

$$G = (\{1, 2, 3\}, \{(1, 2), (1, 3), (2, 1), (2, 3), (3, 1), (3, 2)\}) \tag{6.3}$$

then we would have the picture,



Sometimes, it makes sense to allow multiple connections between the same pair of vertices. This is referred to as a *multigraph*, as opposed to a simple graph.

Once we visualize graphs as a diagram, we begin to classify them in different ways. We say that the *indegree* of a vertex is the number of edges terminating at that vertex, whereas the *outdegree* is the number of edges leaving the vertex. In an undirected graph, we just have the *degree* of a vertex, which is the number of attached edges. The complexity of many graph algorithms, and higher level algorithms, is dependent on the maximum degree. In fact, one definition of a *small world*, or social, graph is a prescribed degree sequence for the vertices. If every pair of vertices is connected, we call it a *complete graph*. The degree of every vertex in a complete graph is $n-1$, where $n = |V|$, since each vertex connects to all others. The number of edges is $\frac{n(n-1)}{2}$, which we can see in the following way. The first vertex makes $n-1$ new edges, the second vertex makes $n-2$ since it connects to every vertex but the first one, and so on. Thus we have

$$|E| = (n-1) + (n-2) + \cdots + 1 \tag{6.4}$$

$$= \sum_{i=1}^{n-1} i \tag{6.5}$$

$$= \frac{n(n-1)}{2} \tag{6.6}$$

where we recognize the sum from Equation (4.31). We denote the complete graph on n nodes as K_n, and exhibit K_5 in Figure 6.1. Why can we speak of *the* complete graph? For a given V there is only one complete graph since every possible edge is present, but surely we could have many different sets V with n elements. This is because mathematicians have a peculiar way of defining equality. We often say that two mathematical objects are the same if there is a bijection between them. If I have two sets V and W, both of cardinality n, then there must be a bijection between them $\phi : V \to W$. If we have the complete graph on V, then we can form the graph

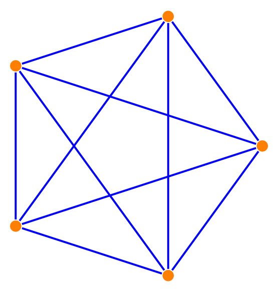
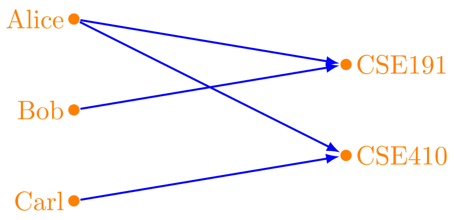


Figure 6.1: The K_5 complete graph

$$G_\phi = (W, E_\phi), E_\phi = \{(v_1, v_2) \in E \mid (\phi(v_1), \phi(v_2)\} \tag{6.7}$$

which is also complete.

We have relations for which the domain and range sets are different, for instance the student registration relation from Equation (3.27). We can also imagine these as graphs, but with a partitioned space of vertices. Edges always originate in the first vertex subset and end in the second subset. We call this a *bipartite graph*. The graph from Equation (3.27) can be represented as



If every vertex from the first subset is connected to every vertex in the second subset, then we call it a *complete bipartite graph*, and denote it $K_{m,n}$ for m vertices in the first set and n in the second.

6.2 Graph Tours

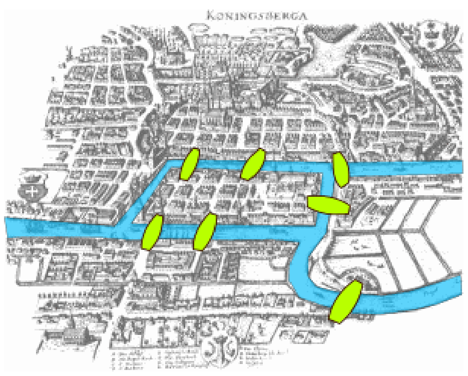
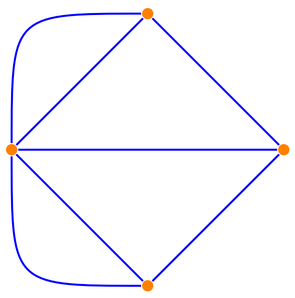


Figure 6.2: The city of Königsberg and its bridges

The original problem solved using graph theory was the Seven Bridges of Königsberg. The city of Königsberg in Prussia (currently Kaliningrad, Russia) was set on both sides of the Pregel River, and included two large islands which were connected to each other, and to the two mainland portions of the city, by seven bridges, as shown in Fig. 6.2. A long-standing problem was to take a walk through the city that would cross each of those bridges once, but only once. The problem was solved by Leonhard Euler (1707-1783), perhaps the greatest mathematician of all time, and certainly the greatest applied mathematician. Euler's genius was to abstract away the non-essential parts of the problem so that he could think through directly the kernel of the difficulty.

Euler sees that the shape of the city, the size, street configuration, and many other factors are all immaterial to the problem at hand. What matters is whether there is a connection (bridge) from one part of the city to another. He collapses each part of the city to a point, since nothing is important except the connections to other parts. There are four parts of the city (two shores and two islands) which become our vertex set V, and seven bridges connecting the parts which become seven edges E. This is the essence of mathematical modeling, stripping out the inessential in order to cleanly formulate a problem that can be solved. The bridge connection graph of Köingsberg is show below,



This is technically a multigraph since it has multiple edges between vertices, which correspond to multiple bridges between two banks. Euler reasoned that except possibly when starting or stopping, when on enters a part of the city (vertex) by one bridge (edge), one then leaves by another bridge (edge). If we are to traverse each bridge (edge) exactly once, each part of the city (vertex) must have an even number of bridges leading from it, except for the start and end. We recognize the "number of bridges leading from" a part of the city as the degree of each vertex. Thus, our trip is only possible if the graph has zero or two vertices of odd degree. The graph above has four vertices of odd degree, and thus such a walk is impossible. Euler's work was presented to the St. Petersburg Academy on 26 August 1735, and published as *Solutio problematis ad geometriam situs pertinentis* (The solution of a problem relating to the geometry of position) in the journal *Commentarii academiae scientiarum Petropolitanae* in 1741.

We will call a set of edges, each connected to the next by a vertex, a *path* through the graph, and if it starts and ends on the same vertex a *cycle*. If every vertex is reachable from every other by a path, then the graph is called *connected*. In honor of the above argument, a path which traverses every edge in the graph once is called *Eulerian path* or Euler walk, and a cycle that does so an Eulerian circuit or *Euler tour*. For an Eulerian path to exist, it was necessary that the graph be connected and the graph to have zero or two vertices of odd degree. This condition turns out also to be sufficient, as proven by Carl Hierholzer (Hierholzer 1873), and furthermore each path must start and end on a vertex of odd degree if they exist. A related notion, the *Hamiltonian path*, is a path which visits every vertex exactly once.

6.3 Trees

Another important type of graph is a *tree*, which is a connected graph not containing a cycle. This is exactly the tree you might have heard about being used for binary searches, or decision trees. To construct a tree, start with the set of vertices V with $|V| > 1$. We must have at least one edge for the graph to be connected. We add the first edge, between two vertices. We mark both of these vertices as seen, and we have $|V| = 2, |E| = 1$. If the graph is connected, we are done. If not, we add another edge starting at a marked vertex. The edge must not terminate at a marked vertex, since if it did we would have a cycle, since there is a path from any vertex to any other as the graph is connected. Thus the edge terminates on an unmarked vertex, which is then marked. At each stage, we add one edge and mark one vertex, and we have the invariant that all marked vertices are connected. The algorithm terminates in $|V| - 1$ steps, so that we have a connected graph with $|V| - 1$ edges and no cycles.

6.4 Planar Graphs

In graph theory, a *planar graph* is a graph that can be embedded in the plane, meaaning it can be drawn so that no edges cross each other, or equivalently it can be drawn on in such a way that its edges intersect only at their endpoints. Notice that planar graphs can also be drawn without edge crossing on the sphere, and vice versa. The special property of planarity means that planar graphs can be considered tesselations of the plane, meaning a division of the plane into pieces, or faces, rather than just vertices and edges. A *face* is a region bounded by a cycle. Thus a graph has one face for each cycle, and then one extra for the infinite (or finite on a sphere) face bounding the graph. This definition allows us to make use of a powerful theorem from topology.

Euler's Formula is one of the bedrock results in topology, but it is straightforward to prove for the case of a planar graph. Simply stated, Euler's Formula says that for any planar graph

$$|V| - |E| + |F| = 2 \tag{6.8}$$

where $|F|$ is the number of faces, including the infinite face outside the graph (or finite if the graph is drawn on the sphere). This is clearly true for our triangle graph, since $3 - 3 + 2 = 2$, where we include the triangular face and the infinite face surrounding the graph. If we embed this instead on the sphere, we can see that we have two finite faces, each bounded by three edges. We can move the three vertices to the equator, and then the faces are the two hemispheres. We can prove that the relation holds for any planar graph by induction on the number of faces f in the graph.

The inductive hypothese will be $P(f) := v - e + f = 2$, where $v = |V|$ and $e = |E|$. The base case will be a tree, which by definition has no cycles (one infinite face), so $P(1) := v - (v - 1) + 1 = 2$ is true. Now suppose that we take our original graph and remove edges. We will remove an edge if removing it removes a face. We keep removing edges until there are no more faces, which is our base case. Now we will add edges back in, one at a time. Each edge creates a face. So suppose $P(f)$ is true (with v and e), and

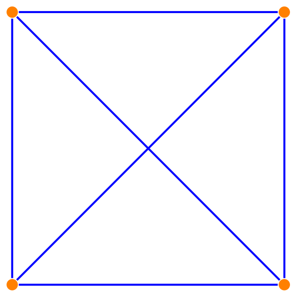


Figure 6.3: The K_4 complete graph

we add one of our removed edges which also creates a face, so that

$$
\begin{aligned}
P(f+1) := v-(e+1)+(f+1) = 2 \quad &(6.9)\\
v-e+f=2 \quad &(6.10)\\
P(f) \quad &(6.11)\\
\mathrm{T}, \quad &(6.12)
\end{aligned}
$$

and our result is proved.

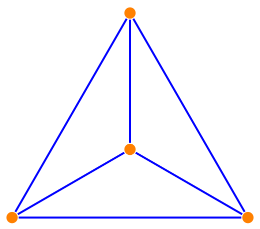


Figure 6.4: K_4 drawn as a planar graph.

We would like to be able to tell which graphs are planar and which are not. We can, of course, try drawing them. In Fig. 6.3 we draw K_4 which has edge crossings, but in Fig. 6.4 we see that an alternate drawing reveals it is in fact planar. How would we prove that a graph is planar? We will look at K_5, as it turns out to be an important part of the story. We will prove that K_5 is not planar by contradiction.

We first assume the negation, namely that K_5 is planar. We know that K_5 has 5 vertices, and 10 edges, but we do not know how many faces. By Euler's Formula,

$$
\begin{aligned}
v-e+f &= 2 && (6.13)\\
5-10+f &= 2 && (6.14)\\
f &= 7. && (6.15)
\end{aligned}
$$

However, each edge separates two faces (boundary edges have the infinite face on the other side), so that if we count all edges around each face we get

$$2e = \sum_f e_f \qquad (6.16)$$

where e_f is the number of edges around each face. In two dimensions, we have $e_f \geq 3$, so that $2e \geq 3f$. Using the result from Euler's Formula, we have

$$
\begin{aligned}
2e &\geq 3f && (6.17)\\
20 &\geq 21 && (6.18)\\
&\mathrm{F}, && (6.19)
\end{aligned}
$$

and we have derived a contradiction. Thus our original assumption is false, and K_5 is nonplanar. In fact, by Wagner's Theorem, a finite graph is planar if and only if it does not have K_5 or $K_{3,3}$ as a minor. A graph *minor* is a subgraph obtained from the original by deleting or contracting edges.

6.5 Sprouts

According to Martin Gardner, the game of Sprouts was invented by John Horton Conway and Michael Paterson at the University of Cambridge in 1967. The

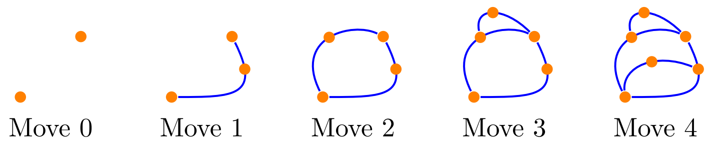


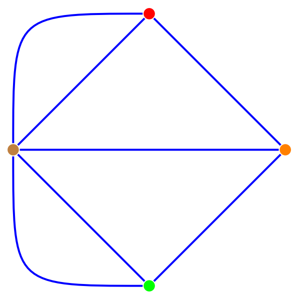
Figure 6.5: A game of Sprouts in which Player 2 wins.

game begins with any number of vertices placed on the plane. The first play draws a edge starting and ending on a vertex (the starting and ending vertex can be the same), and adds a vertex somewhere on the edge. The edge may not intersect another edge or vertex, and at most three edges may meet at a vertex. Play alternates in this fashion between players until it is impossible for one player to move, at which point the other player wins. An example game is shown in Fig. 6.5. At the start of a game, there are $3n$ places to attach edges, but each edge destroys two spots and adds one. Thus a game cannot be longer than $3n - 1$ moves.

6.6 Problems

Problem VI.1 Suppose we have two sets V and W, both of cardinality n. Thus there is a bijection ϕ between the sets. I can map a graph on V into a graph on W using ϕ as shown in Section 6.1. Prove that the graph we get after mapping the complete graph K_n on V into W is complete.

Problem VI.2 Consider again Königsberg, only now we will distinguish the parts of the city. The northern bank of the river is occupied by the School of Mathematics (red), the southern by the School of Physics (green), the west bank is home to the Administration (brown), and on the small island in the east is the Pub (orange),



1. It being customary for students after some hours at the Pub to try to walk the bridges, and many have returned for more refreshment claiming success. However, none have been able to repeat the feat by the light of day. A clever mathematics student, having analyzed the town's bridge

system by means of graph theory, concludes that the bridges cannot be walked. He contrives a cunning plan to build a makeshift eighth bridge so that he can begin in the evening at School, walk the bridges, and end at the Pub to brag of his victory. Of course, he wants the physicists to be unable to duplicate the feat from their School. Where does the mathematician build the eighth bridge?

2. A clever physics student, infuriated by the mathematician's solution to the problem, wants to construct another bridge, enabling her to begin at her School, walk the bridges, and end at the Pub to demonstrate the superiority of physical reasoning. As an extra bit of revenge, mathematicians should no longer be able to walk the bridges starting at the School of Mathematics and ending at the Pub as before. Where does she build her bridge?

3. The harried Provost has watched this furious bridge-building with dismay. It upsets the spirit of the University and, worse, contributes to absenteeism. He wants to build a final bridge with university funds that allows all the students to walk the bridges and return to their own dorms. Where does the Provost build the final bridge?

Problem VI.3 Does K_5 have a Hamiltonian path? If so, exhibit one.

Problem VI.4 An ancient King in Westeros had a large kingdom and five able sons. On his deathbed, he decreed that in order to preserve harmony among his heirs, each should receive a portion of the kingdom with a sizeable border to all the others. A grand road should be built between each pair of palaces, so that the princes could visit each other often. How was the kingdom divided? If the kingdom descended into war because a division could not be agreed, explain why.

Problem VI.5 Prove that $K_{3,3}$ is not planar

References

Hierholzer, Carl (1873). "Ueber die Möglichkeit, einen Linienzug ohne Wiederholung und ohne Unterbrechung zu umfahren". In: *Mathematische Annalen* 6, pp. 30–32. DOI: 10.1007/bf01442866.

Chapter 7

Problem Solutions

Proof Index

Concept Index