Introduction to Computational Logic

Lecture Notes SS 2013

July 26, 2013

Gert Smolka and Chad E. Brown
Department of Computer Science
Saarland University

Contents

Introduction

This course is an introduction to basic logic principles, constructive type theory, and interactive theorem proving with the proof assistant Coq. At Saarland University the course is taught in this format since 2010. Students are expected to be familiar with basic functional programming and the structure of mathematical definitions and proofs. Talented students at Saarland University often take the course in the second semester of their Bachelor's studies.

Constructive type theory provides a programming language for developing mathematical and computational theories. Theories consist of definitions and theorems, where theorems state logical consequences of definitions. Every theorem comes with a proof justifying it. If the proof of a theorem is correct, the theorem is correct. Constructive type theory is designed such that the correctness of definitions and proofs can be checked automatically.

Coq is an implementation of a constructive type theory known as the calculus of inductive definitions. Coq is designed as an interactive system that assists the user in developing theories. The most interesting part of the interaction is the construction of proofs. The idea is that the user points the direction while Coq takes care of the details of the proof. In the course we use Coq from day one.

Coq is a mature system whose development started in the 1980's. In recent years Coq has become a popular tool for research and education in formal theory development and program verification. Landmarks are a proof of the four color theorem, a proof of the Feit-Thompson theorem, and the verification of a compiler for the programming language C (COMPCERT).

Coq is the applied side of this course. On the theoretical side we explore the basic principles of constructive type theory, which are essential for programming languages, logical languages, proof systems, and the foundation of mathematics.

An important part of the course is the theory of classical and intuitionistic propositional logic. We study various proof systems (Hilbert, ND, tableaux), decidability of proof systems, and the semantic analysis of proof systems based on models. The study of propositional logic is carried out in Coq and serves as a case study of a substantial formal theory development.

Dedication

This text is dedicated to the many people who have designed and implemented Coq since 1985.

Contents

1 Types and Functions

In this chapter, we take a first look at Coq and its mathematical programming language. We define basic data types such as booleans, natural numbers, and lists and functions operating on them. For the defined functions we prove equational theorems, constructing the proofs in interaction with the Coq interpreter. The definitions we study are often recursive and the proofs we construct are often inductive.

In the following it is absolutely essential that you have a Coq interpreter running and that you experiment with the definitions and proofs we discuss. In Coq, proofs are constructed with scripts and the resulting proof process can only be understood in interaction with a Coq interpreter.

1.1 Booleans

We start with the definition of a type *bool* with two elements *true* and *false*.

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

The words *Inductive* and *Type* are keywords of Coq and the identifiers *bool*, *true*, and *false* are the names we have chosen for the type and its elements. The identifiers *bool*, *true*, and *false* serve as **constructors**, where *bool* is a **type constructor** and *true* and *false* are the **value constructors** of *bool*. The above definition overwrites the definition of *bool* in Coq's standard library, but this does not matter for our first encounter with Coq.

We define a negation function *negb*.

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

The *match* term represents a case analysis for the boolean argument x. There is a rule for each value constructor of *bool*. We can check the type of terms with the command *Check*:

```
Check negb.
% negb : bool → bool
Check negb (negb true).
% negb (negb true) : bool
```

We can evaluate terms with the command *Compute*.

```
Compute negb (negb true).
% true : bool
```

We are now ready for our first proof with Coq.

```
Lemma L1 :
  negb true = false.
Proof. simpl. reflexivity. Qed.
```

The command starting with the keyword *Lemma* states the equation we want to prove and gives the lemma the name *L1*. The sequence of commands starting with *Proof* and ending with *Qed* constructs the proof of Lemma *L1*. It is now essential that you step through the commands with the Coq interpreter one by one. Once the lemma command is accepted, Coq switches from **top level mode** to **proof editing mode**. The commands between *Proof* and *Qed* are called **tactics**. The tactic *simpl* simplifies both sides of the equation to be shown by applying the definition of *negb*. This leaves us with the trivial equation $false = false$, which we prove with the tactic *reflexivity*. The command *Qed* finishes the proof.

Our second proof shows that double negation is identity.

```
Lemma negb_negb (x : bool) :
  negb (negb x) = x.
Proof.
  destruct x.
  - reflexivity.
  - reflexivity.
Qed.
```

This time the claim involves a boolean variable x and the proof proceeds by case analysis on x. Since *reflexivity* performs simplification automatically, we have omitted the tactic *simpl*.

It is important that with Coq you step back and forth in the proof script and observe what happens. This way you can see how the proof advances. At each point in the proof process you are confronted with a **proof goal** comprised of a list of **assumptions** (possibly empty) and a **claim**. Here are the proof goals you will see when you step through the above proof script.

$$\frac{x : bool}{negb\,(negb\,x) = x} \qquad \frac{}{negb\,(negb\,true) = true}$$

$$\frac{}{negb\,(negb\,false) = false}$$

In each goal, the assumptions appear above and the claim appears below the rule. The tactic *destruct x* does the case analysis and replaces the initial goal with two subgoals, one for $x = true$ and one for $x = false$. The proof is finished if both subgoals are solved (i.e., proved).

Since the proof finishes with *reflexivity* in both cases, we can shorten the proof script by combining the tactics *destruct x* and *reflexivity* with the **semicolon operator**.

```
Proof. destruct x ; reflexivity. Qed.
```

We define a boolean conjunction function *andb*.

```
Definition andb (x y : bool) : bool :=
  match x with
    | true ⇒ y
    | false ⇒ false
  end.
```

We prove that boolean conjunction is commutative.

```
Lemma andb_com x y :
  andb x y = andb y x.

Proof.
  destruct x.
  − destruct y ; reflexivity.
  − destruct y ; reflexivity.
Qed.
```

The proof can be written more succinctly as

```
Proof. destruct x, y ; reflexivity. Qed.
```

The short proof script has the drawback that you don't see much when you step through it. For that reason we will often give proof scripts that are longer than necessary.

Note that we have stated the lemma *andb_com* without giving types for the variables x and y. This leaves it to Coq to infer the missing types. When you look at the initial goal of the proof, you will see that x and y have both received the type *bool*. Automatic **type inference** is an important feature of Coq.

A word on terminology. In mathematics, theorems are usually classified into propositions, lemmas, theorems, and corollaries. This distinction is a matter of style and does not matter logically. When we state a theorem in Coq, we will mostly use the keyword *Lemma*. Coq also accepts the keywords *Proposition*, *Theorem*, and *Corollary*, which are treated as synonyms.

Exercise 1.1.1 A boolean disjunction $x \vee y$ yields *false* if and only if both x and y are *false*.

a) Define disjunction as a function $orb : bool \to bool \to bool$ in Coq.

b) Prove that disjunction is commutative.

c) Formulate and prove the De Morgan law $\neg(x \vee y) = \neg x \wedge \neg y$ in Coq.

1.2 Cascaded Functions

When we look at the type of *andb*

```
Check andb.
% andb : bool → bool → bool
```

we note that Coq realizes *andb* as a **cascaded function** taking a boolean argument and returning a function $bool \to bool$. This means that an application *andb x y* first applies *andb* to just x. The resulting function is then applied to y. Cascaded functions are standard in functional programming languages where they are called curried functions.

To say more about cascaded functions, we consider **lambda abstractions**. A lambda abstraction is a term $\lambda x{:}s.t$ describing a function taking an argument x of type s and yielding the value described by the term t. For instance, the term $\lambda x{:}bool.x$ describes an identity function on *bool*. In Coq, lambda abstractions are written with the keyword *fun*:

```
Check fun x : bool ⇒ x.
% fun x : bool ⇒ x : bool → bool
```

Given an application of a lambda abstraction to a term, we can perform an evaluation step known as **beta reduction**:

$$(\lambda x{:}s.t)u \rightsquigarrow t^x_u$$

The notation t^x_u represents the term obtained from t by replacing the variable x with the term u. Beta reduction captures the intuitive notion of function application. Beta reduction is a basic computation rule in Coq.

```
Compute (fun x : bool ⇒ x) true.
% true : bool
```

Given the above explanations, the term

```
Check andb true.
% andb true : bool → bool
```

should describe an identity function $bool \to bool$. We confirm this hypothesis by evaluating the term with Coq.

```
Compute andb true.
% fun y : bool ⇒ y : bool → bool
```

To evaluate a term, Coq rewrites the term with symbolic reduction rules. The evaluation of *andb true* involves three **reduction steps**.

```
    andb true
                  unfolding of the definition of andb
=  (fun x : bool ⇒ fun y : bool ⇒ match x with true ⇒ y | false ⇒ false end) true
                  beta reduction
=  fun y : bool ⇒ match true with true ⇒ y | false ⇒ false end
                  match reduction
=  fun y : bool ⇒ y
```

The unfolding step done first suggests that we wrote the definition of *andb* using notational sugar. Using plain notation, we can define *andb* as follows.

```
Definition andb : bool → bool → bool :=
  fun x : bool ⇒
    fun y : bool ⇒
      match x with
        | true ⇒ y
        | false ⇒ false
      end.
```

Internally, Coq represents definitions and terms always in plain syntax. You can check this with the command *Print*.

```
Print negb.
negb = fun x : bool ⇒ match x with
                       | true ⇒ false
                       | false ⇒ true
                      end
     : bool → bool
```

Coq prints the definition of *andb* with a notational convenience to ease reading.

```
Print andb.
andb = fun x y : bool ⇒ match x with
                        | true ⇒ y
                        | false ⇒ false
                       end
     : bool → bool → bool
```

The additional argument variable y in the lambda abstraction for x represents a nested lambda abstraction for y (see the definition of *andb* above).

There are two basic notational rules for function types and function applications making many parentheses superfluous:

$s \to t \to u$	$\rightsquigarrow$	$s \to (t \to u)$	function arrow groups to the right
$s\ t\ u$	$\rightsquigarrow$	$(s\ t)\ u$	function application groups to the left

We have made use of these rules already. Without the rules, the application *andb x y* would have to be written as (*andb x*) *y*, and the type of *andb* would have to be written as *bool* → (*bool* → *bool*).

When using the commands *Print* and *Check*, you may see the keyword *Set* in places where you would expect the keyword *Type*. Types of sort *Set* are types at the lowest level of a type hierarchy. For now this hierarchy does not matter.

1.3 Natural Numbers

The natural numbers can be obtained with two constructors O and S:

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

Expressed with O and S, the natural numbers 0, 1, 2, 3, ... look as follows:

$$O,\ \ S\,O,\ \ S(S\,O),\ \ S(S(S\,O)),\ \ \ldots$$

We say that the natural numbers are obtained by iterating the **successor function** S on the initial number O. This is a form of recursion. The recursion makes it possible to obtain infinitely many values with finitely many constructors. The constructor representation of the natural numbers goes back to Dedekind and Peano.

Here is a function that yields the **predecessor** of a number.

```
Definition pred (x : nat) : nat :=
  match x with
    | O ⇒ O
    | S x' ⇒ x'
  end.

Compute  pred (S(S O)).
% S O : nat
```

We now define an addition function for the natural numbers. We base the definition on two equations:

$$O + y = y$$
$$Sx + y = S(x + y)$$

The equations are valid for all numbers x and y if we read Sx as $x + 1$. Read from left to right, they constitute a recursive algorithm for computing the sum of two numbers. The left-hand sides of the two equations amount to an exhaustive case analysis. The second equation is recursive in that it reduces an addition

$Sx + y$ to an addition $x + y$ with a smaller argument. Here is a computation applying the equations for $+$:

$$S(S(S\,O)) + y = S(S(S\,O) + y) = S(S(S\,O + y)) = S(S(S\,y))$$

In Coq, we express the recursive algorithm described by the equations with a recursive function *plus*.

```
Fixpoint plus (x y : nat) : nat :=
  match x with
    | O ⇒ y
    | S x' ⇒ S (plus x' y)
  end.

Compute plus (S O) ( S O).
% S(S O)) : nat
```

The keyword *Fixpoint* indicates that a recursive function is being defined. In Coq, functional recursion is always **structural recursion**. Structural recursion means that the recursion acts on the values of an inductive type and that each recursion step takes off at least one constructor. Structural recursion always terminates.

Here is the definition of a comparison function $leb : nat \to nat \to bool$ that tests whether its first argument is less or equal than its second argument.

```
Fixpoint leb (x y: nat) : bool :=
  match x with
    | O ⇒ true
    | S x' ⇒ match y with
                | O ⇒ false
                | S y' ⇒ leb x' y'
              end
  end.
```

A shorter, more readable definition of *leb* looks as follows:

```
Fixpoint leb' (x y: nat) : bool :=
  match x, y with
    | O, _ ⇒ true
    | _, O ⇒ false
    | S x', S y' ⇒ leb' x' y'
  end.
```

Coq translates the short form automatically into the long form (you can check this with the command *Print leb'*). The underline character used in the short form serves as *wildcard pattern* that matches everything. The order of the rules in sugared matches is significant. The second rule in the sugared match is only correct if the order of the rules is taken into account.

Exercise 1.3.1 Define a multiplication function $mult : nat \to nat \to nat$. Base your definition on the equations

$$O \cdot y = O$$
$$Sx \cdot y = y + x \cdot y$$

and use the addition function *plus*.

Exercise 1.3.2 Define functions as follows. In each case, first write down the equations your function is based on.

a) A function $power : nat \to nat \to nat$ that yields x^n for x and n.

b) A function $fac : nat \to nat$ that yields $n!$ for n.

c) A function $evenb : nat \to bool$ that tests whether its argument is even.

d) A function $mod2 : nat \to nat$ that yields the remainder of x on division by 2.

e) A function $minus : nat \to nat \to nat$ that yields $x - y$ for $x \geq y$.

f) A function $gtb : nat \to nat \to bool$ that tests $x > y$.

g) A function $eqb : nat \to nat \to bool$ that tests $x = y$. Do not use *leb* or *gtb*.

1.4 Structural Induction and Rewriting

The inductive type *nat* comes with two basic principles: structural recursion for defining functions and structural induction for proving lemmas. Suppose we have a proof goal

$$\frac{x : nat}{p\,x}$$

where $p\,x$ is a claim that depends on a variable x of type *nat*. Then **structural induction on** x will reduce the goal to two subgoals:

$$\frac{}{p\,O} \qquad \frac{\begin{array}{c} x : nat \\ IHx : p\,x \end{array}}{p\,(S\,x)}$$

This reduction is a case analysis on the structure of x, but has the additional feature that the second subgoal comes with an extra assumption *IHx* known as **inductive hypothesis**. We think of *IHx* as a proof of $p\,x$. If we can prove both subgoals, we have established the initial claim $p\,x$ for all $x : nat$. The correctness of the proof rule for structural induction can be argued as follows.

1. The first subgoal gives us a proof of $p\,O$.
2. The second subgoal gives us a proof of $p\,(S\,O)$ from the proof of $p\,O$.

3. The second subgoal gives us a proof of $p(S(S\,O))$ from the proof of $p(S\,O)$.
4. After finitely many steps we arrive at a proof of $p\,x$.

It makes sense to see the proof of the second subgoal as a function that for a proof of $p\,x$ yields a proof of $p(S\,x)$. We can now obtain a proof of $p\,x$ by structural recursion: If $x = O$, we take the proof provided by the first subgoal. If $x = S\,x'$, we first obtain a proof of $p\,x'$ by recursion and then obtain a proof of $p\,x = p(S\,x')$ by applying the function provided by the second subgoal.

We will explore the logical correctness of structural recursion in more detail once we have laid out more foundations. For now we are interested in applying the rule when we construct proofs with Coq, and this will turn out to be straightforward.

Our first case study of structural induction will be a proof that addition is commutative, that is, $plus\,x\,y = plus\,y\,x$. Formally, this fact is not completely obvious, since the definition of *plus* is by recursion on the first argument and thus asymmetric. We will first show that the symmetric variants

$$x + O = x$$
$$x + S\,y = S(x + y)$$

of the equations underlying the definition of *plus* hold. Here is our first inductive proof in Coq.

```
Lemma plus_O x :
  plus x O = x.
Proof.
  induction x ; simpl.
  - reflexivity.
  - rewrite IHx. reflexivity.
Qed.
```

If you step through the proof script with Coq, you will see the following proof goals.

$$\frac{x : nat}{plus\ x\ O = x}\ \text{induction x ; simpl} \qquad \frac{}{O = O}\ \text{reflexivity}$$

$$\frac{x : nat \quad IHx : plus\ x\ O = x}{S(plus\ x\ O) = Sx}\ \text{rewrite IHx} \qquad \frac{x : nat \quad IHx : plus\ x\ O = x}{Sx = Sx}\ \text{reflexivity}$$

Of particular interest is the application of the inductive hypothesis with the tactic *rewrite IHx*. The tactic rewrites a subterm of the claim with the equation *IHx*.

Doing inductive proofs with Coq is fun since Coq takes care of the bureaucratic aspects of the proof process. Here is our next example.

```
Lemma plus_S x y :
  plus x (S y) = S (plus x y).
Proof.
  induction x ; simpl.
  - reflexivity.
  - rewrite IHx. reflexivity.
Qed.
```

Note that the proof scripts for the lemmas *plus_S* and *plus_O* are identical. When you run the script for each of the two lemmas, you see that they generate different proofs. Using the lemmas, we can prove that addition is commutative.

```
Lemma plus_com x y :
  plus x y = plus y x.
Proof.
  induction x ; simpl.
  - rewrite plus_O. reflexivity.
  - rewrite plus_S. rewrite IHx. reflexivity.
Qed.
```

Note that the lemmas are applied with the rewrite tactic.

Next we prove that addition is associative.

```
Lemma plus_asso x y z :
  plus (plus x y) z = plus x (plus y z).
Proof.
  induction x ; simpl.
  - reflexivity.
  - rewrite IHx. reflexivity.
Qed.
```

Exercise 1.4.1 Prove the commutativity of *plus* by induction on y.

1.5 More on Rewriting

When we rewrite with an equational lemma like *plus_com*, it may happen that the lemma applies to several subterms of the claim. In such a situation it may be necessary to tell Coq which subterm it should rewrite. To do such controlled rewriting, we have to load the module *Omega* of the standard library and use the tactic *setoid_rewrite*. Here is an example deserving careful exploration with Coq.

```
Require Import Omega.
Lemma plus_AC x y z :
  plus y (plus x z) = plus (plus z y) x.
```

```
Proof.
  setoid_rewrite plus_com at 3.
  setoid_rewrite plus_com at 1.
  apply plus_asso.
Qed.
```

Note the use of the tactic *apply* to finish the proof by application of the lemma *plus_asso*. Here is a more involved example.

```
Lemma plus_AC' x y z :
  plus (plus (mult x y) (mult x z)) (plus y z) = plus (plus (mult x y) y) (plus (mult x z) z).
Proof.
  rewrite plus_asso. rewrite plus_asso. f_equal.
  setoid_rewrite plus_com at 1. rewrite plus_asso. f_equal.
  apply plus_com.
Qed.
```

Run the proof script to understand the effect of the tactic *f_equal*.

Both rewrite tactics can apply equations from right to left. This is requested by writing an arrow "<-" before the name of the equation. Here is an example (one can use the keyword *Example* as a synonym for *Lemma*).

```
Example Ex1 x y z :
  S (plus x (plus y z)) = S (plus (plus x y) z).
Proof. rewrite ← plus_asso. reflexivity. Qed.
```

Exercise 1.5.1 Prove the following lemma without using the tactic *reflexivity* for the inductive step (i.e., the second subgoal of the induction). Use the tactics *f_equal* and *apply* to substitute for *reflexivity*.

```
Lemma mult_S' x y :
  mult x (S y) = plus (mult x y) x.
```

Exercise 1.5.2 Prove the following lemmas.

```
Lemma mult_O  (x : nat) :
  mult x O = O.

Lemma mult_S (x y : nat) :
  mult x (S y) = plus (mult x y) x.

Lemma mult_com (x y : nat) :
  mult x y = mult y x.

Lemma mult_dist (x y z: nat) :
  mult (plus x y) z = plus (mult x z) (mult y z).

Lemma mult_asso (x y z: nat) :
  mult (mult x y) z = mult x (mult y z).
```

1.6 Recursive Abstractions

The plain notation for recursive functions uses **recursive abstractions** written with the keyword *fix*.

```
Print plus.
```

```
plus = fix f (x y : nat) {struct x} : nat :=
         match x with
           | O ⇒ y
           | S x' ⇒ S (f x' y)
         end
     : nat → nat → nat
```

The variable f appearing after the keyword *fix* is local to the abstraction and represents the recursive function described. As with argument variables, the local name of a recursive function does not matter. You may use g or *plus* in place of f, for instance. The annotation $\{struct\ x\}$ says that the structural recursion is on x. If you write a recursive abstraction by hand you may omit the annotation and Coq will infer it automatically. In fully plain notation the recursive abstraction for *plus* takes only one argument:

```
fix  f  (x : nat) : nat → nat :=
  fun y : nat ⇒ match x with
                  | O ⇒ y
                  | S x' ⇒ S (f x' y)
                end.
```

The reduction rule for recursive abstractions only applies if the argument of the recursive abstraction exhibits at least one constructor. When a recursive abstraction is reduced, the local name of the recursive function is replaced with the recursive abstraction. Experiment with Coq to get a feel for this. The following interactions will get you started.

```
Compute plus O.
% fun y : nat ⇒ y
```

```
Compute plus (S (S O)).
% fun y : nat ⇒ S(Sy)
```

```
Compute fun x ⇒ plus (S x).
```

```
fun x : nat ⇒
  fun y : nat ⇒
    S ( (fix  f  (x : nat) : nat → nat :=
           fun y : nat ⇒ match x with
                           | O ⇒ y
                           | S x' ⇒ S (f x' y)
                           end) x y )
```

At first, the many notational variants Coq supports for a term can be confusing. Even in printing-all mode identical terms may be displayed with different names for the local variables. You can find out more by stating equational lemmas and using the tactics *compute* and reflexivity. Here are examples.

```
Goal plus O = fun x ⇒ x.
Proof. compute. reflexivity. Qed.
Goal (fun x ⇒ plus (S x)) = fun x y ⇒ S (plus x y).
Proof. compute. reflexivity. Qed.
```

The command *Goal* states a lemma without giving it a name. The tactic *compute* computes the normal form of the claim. We have inserted the compute tactic so that we can see the normal forms of the terms being equated. The **normal form** of a term s is the term obtained by fully evaluating s. Every term has exactly one normal form. The reflexivity tactic proves every equation where both sides evaluate to the same normal form.

1.7 Defined Notations

Coq comes with commands for defining notations. For instance, we can define infix notations for *plus* and *mult*.

```
Notation "x + y" := (plus x y)  (at level 50, left associativity).
Notation "x * y" := (mult x y) (at level 40, left associativity).
```

We can now write the distributivity law for multiplication and addition in familiar form:

```
Lemma mult_dist' x y z :
  x * (y + z) = x*y + x*z.
Proof.
  induction x ; simpl.
  - reflexivity.
  - rewrite IHx. rewrite plus_asso. rewrite plus_asso. f_equal.
    setoid_rewrite ← plus_asso at 2.
    setoid_rewrite plus_com at 4.
    symmetry. apply plus_asso.
Qed.
```

Note the use of the tactic *symmetry* to turn around the equation to be shown.

You can tell Coq to not use defined notations when it prints terms.[1]

```
Set Printing All.
```

[1] When working with CoqIde, use the view menu to switch printing-all mode on and off (display all basic low-level contents).

```
Check O + O * S O.
% plus O (mult O (S O)) : nat
Unset Printing All.
```

It is very important to distinguish between notation and abstract syntax when working with Coq. Notations are used when reading input from and writing output to the user. Internally, all notational sugar is removed and terms are represented in **abstract syntax**. The abstract syntax is basically what you see in printing-all mode. All logical reasoning is defined on the abstract syntax. As it comes to semantic issues, it is irrelevant in which notation a syntactic object is described. So if for some term written with notational sugar it is not clear to you how it translates to abstract syntax, switching to printing-all mode is always a good idea.

Exercise 1.7.1 Prove the lemmas from Exercise 1.5.2 once more using infix notations for *plus* and *mult*. Note that the proof scripts remain unchanged.

Exercise 1.7.2 Prove associativity of multiplication using the distributivity lemma *mult_dist'* from this section. This proof requires more applications of the commutativity law for multiplication than a proof using the lemma *mult_dist* from Exercise 1.5.2.

Exercise 1.7.3 Prove $(x + x) + x = x + (x + x)$ by induction on x using Lemma *plus_S*. Note that the direct proof of this instance of the associativity law is more complicated than the proof of the general associativity law. In fact, it seems impossible to prove $(x + x) + x = x + (x + x)$ without using a lemma.

1.8 Standard Library

Coq comes with an extensive standard library providing definitions, notations, lemmas, and tactics. When it starts, the Coq interpreter loads part of the standard library. You can load additional modules using the command *Require*. (We have already used *Require* to load the module *Omega* so that we can use the smart rewriting tactic *setoid_rewrite*.)

The definitions the Coq interpreter starts with include the types *bool* and *nat*. So there is no need to define these types when we want to use them. The standard library equips *nat* with many notations familiar from Mathematics. For instance, we may write $2 + 3 * 2$ for *plus* $(S(S\ O))$ (*mult* $(S(S(S\ O)))$ $(S(S\ O))$). The following interaction illustrates the predefined notational sugar.

```
Set Printing All.
```

```
Check 2+3*2.
% plus (S(S O)) (mult (S(S(S O))) (S(S O))) : nat
Unset Printing All.
```

The above interaction took place in a context where the library definitions of *nat*, *plus*, and *mult* were not overwritten. If you execute the above commands in a context where you have defined your own versions of *nat*, *plus*, and *times*, you will see that the notations 2, 3, +, and ∗ still refer to the predefined objects from the library. If you want to know more about predefined identifiers, you may use the commands *Check* and *Print* or consult the Coq library pages in the Web (at coq.inria.fr). If you want to know more about a notation, you may use the command *Locate*.

```
Locate "*".
```

When you run the above command, you will see that "*" is used with more than one definition (so-called overloading).

For boolean matches, Coq's library provides the if-then-else notation. For instance:

```
Set Printing All.
Check if false then 0 else 1.
% match false return nat with true ⇒ O | false ⇒ S O end
Unset Printing All.
```

Note that the match is shown with a **return type annotation**. The return type annotation is part of the abstract syntax of a match. The annotation is usually added by Coq but can also be stated explicitly.

The standard module *Omega* comes with an **automation tactic** *omega* that knows about the arithmetic primitives of the library. For instance, *omega* can prove that addition is associative:

```
Goal ∀ x y z, (x + y) + z = x + (y + z).
Proof. intros x y z. omega. Qed.
```

Note the explicit quantification of the variables x, y, and z with the universal quantifier ∀. The symbol ∀ can written as the string *forall* in Coq. Also note the use of the tactic *intros* to introduce the quantified variables as assumptions.

The tactic *omega* works well for goals that involve addition and subtraction. It knows little about multiplication but can deal well with products where one side is a constant.

```
Goal ∀ x y, 2 * (x + y) = (y + x) * 2.
Proof. intros x y. omega. Qed.
```

1.9 Pairs and Implicit Arguments

Given two values x and y, we can form the ordered pair (x, y). Given two types X and Y, we can form the product type $X \times Y$ containing all pairs whose first component is an element of X and whose second component is an element of Y. This leads to the Coq definition

```
Inductive prod (X Y : Type) : Type :=
| pair : X → Y → prod X Y.
```

which fixes two constructors

$$\begin{aligned} \mathit{prod} &: \mathit{Type} \to \mathit{Type} \to \mathit{Type} \\ \mathit{pair} &: \forall X\ Y : \mathit{Type}.\ X \to Y \to \mathit{prod}\ X\ Y \end{aligned}$$

for obtaining products and pairs. The pairing constructor takes four arguments, where the first two arguments are the types of the components of the pair to be constructed. Here are typings explaining the type of the pairing constructor.

$$\begin{aligned} \mathit{pair\ nat} &: \forall Y : \mathit{Type}.\ \mathit{nat} \to Y \to \mathit{prod\ nat}\ Y \\ \mathit{pair\ nat\ bool} &: \mathit{nat} \to \mathit{bool} \to \mathit{prod\ nat\ bool} \\ \mathit{pair\ nat\ bool\ O} &: \mathit{bool} \to \mathit{prod\ nat\ bool} \\ \mathit{pair\ nat\ bool\ O\ true} &: \mathit{prod\ nat\ bool} \end{aligned}$$

One says that *pair* is a **polymorphic constructor**. This addresses the fact that the types of the third and fourth argument are given as first and second argument. While the logical analysis is conclusive, the resulting notation for pairs is tedious. As is, we have to write *pair nat bool 0 true* for the pair $(0, \mathit{true})$. Fortunately, Coq comes with a **type inference feature** making it possible to just write *pair 0 true* and leave it to the interpreter to insert the missing arguments. One speaks of **implicit arguments**. With the command

```
Arguments pair {X} {Y} _ _.
```

we tell Coq to treat the arguments X and Y of *pair* as implicit arguments. Now we can obtain pairs without specifying the types of the components.

```
Check pair 0 true.
% pair 0 true : prod nat bool
```

The implicit arguments of a function can still be given explicitly if we prefix the name of the function with the character @:

```
Check @pair nat.
% @pair nat : ∀ Y : Type, nat → Y → prod nat Y
Check @pair nat bool 0.
% @pair nat bool 0 : bool → prod nat bool
```

We can see which terms Coq inserts for the implicit arguments by switching to printing-all mode.

```
Set Printing All.

Check pair 0 true.
% @pair nat bool 0 true : prod nat bool

Unset Printing All.
```

You can use the command *About* to find out which arguments of a function name are implicit.

```
About pair.

  pair : ∀ X Y : Type, X → Y → prod X Y
  Arguments X, Y are implicit
```

Coq actually prints more information about the arguments, but the extra information is not relevant for now.

We can switch Coq into **implicit arguments mode**, which has the effect that some arguments are automatically declared implicit when a function name is defined. With implicit arguments mode on, the inductive definition of pairs would automatically equip the constructor *pair* with the two implicit arguments declared above. We now switch to implicit arguments mode

```
Set Implicit Arguments.
Unset Strict Implicit.
```

and define functions yielding the first and the second component of a pair (so-called projections).

```
Definition fst (X Y : Type) (p : prod X Y) : X :=
  match p with pair x _ ⇒ x end.

Definition snd (X Y : Type) (p : prod X Y) : Y :=
  match p with pair _ y ⇒ y end.

Compute fst (pair O true).
% O : nat

Compute snd (pair O true).
% true : bool
```

Note that the first two arguments of *fst* and *snd* are implicit. We prove the so-called **eta law for pairs**.

```
Lemma pair_eta (X Y : Type) (p : prod X Y) :
  pair (fst p) (snd p) = p.

Proof. destruct p as [x y]. simpl. reflexivity. Qed.
```

Note the use of the tactic *destruct*. It replaces the variable p with the pair *pair x y* where x and y are fresh variables. This is justified since *pair* is the only constructor with which a value of type *prod X Y* can be obtained. Destructuring of a variable of a single constructor type is similar to matching on a variable of a single constructor type (see the definitions of *fst* and *snd*).

The standard library defines products and pairs as shown above and equips them with familiar notations. Using the definitions and notations of the standard library, we can state and prove the eta law as follows.

```
Lemma pair_eta (X Y : Type) (p : X * Y) :
  (fst p, snd p) = p.
Proof. destruct p as [x y]. simpl. reflexivity. Qed.
```

Here is a function swapping the components of a pair:

```
Definition swap (X Y : Type) (p : X * Y) : Y * X :=
  (snd p, fst p).
Compute swap (0, true).
% (true, 0) : prod bool nat
Lemma swap_swap (X Y : Type) (p : X * Y) :
  swap (swap p) = p.
Proof. destruct p as [x y]. unfold swap. simpl. reflexivity. Qed.
```

Note the use of the tactic *unfold*. We use it since *simpl* does not simplify applications of functions not involving a match. Since *reflexivity* does all the required unfolding and simplification automatically, we may omit the unfold and simplification tactics in the above script.

The notations for pairs and products are defined such that nesting to the left may be written without parentheses. For instance, we may write $(1,2,3)$ for $((1,2),3)$ and $nat * nat * nat$ for $(nat * nat) * nat$. So the command

```
Check (fun x : nat * nat * nat ⇒ fst x) (1,2,3)
```

will succeed with the type $nat * nat$.

Exercise 1.9.1 An operation taking two arguments can be represented either as a function taking its arguments one by one (**cascaded representation**) or as a function taking both arguments bundled in one pair (**cartesian representation**). While the cascaded representation is natural in Coq, the cartesian representation is common in mathematics. Define polymorphic functions

$$car : \forall X\ Y\ Z : Type,\ (X \to Y \to Z) \to (X * Y \to Z)$$
$$cas : \forall X\ Y\ Z : Type,\ (X * Y \to Z) \to (X \to Y \to Z)$$

that translate between the cascaded and cartesian representation and prove the correctness of your functions with the following lemmas.

```
Lemma car_spec X Y Z (f : X → Y → Z) x y :
  car f (x,y) = f x y.
Lemma cas_spec X Y Z (f : X * Y → Z) x y :
  cas f x y = f (x,y).
```

Note that the arguments X, Y, and Z of *car* and *cas* are implicit.

1.10 Lists

Lists represent finite sequences $[x_1; \ldots; x_n]$ with two constructors *nil* and *cons*.

$$\begin{aligned} [] &\mapsto nil \\ [x] &\mapsto cons\ x\ nil \\ [x; y] &\mapsto cons\ x\ (cons\ y\ nil) \\ [x; y; z] &\mapsto cons\ x\ (cons\ y\ (cons\ z\ nil)) \end{aligned}$$

The constructor *nil* represents the empty sequence. Nonempty sequences are obtained with the constructor *cons*. All elements of a list must be taken from the same type. This design is realized by the following inductive definition.

```
Inductive list (X : Type) : Type :=
| nil : list X
| cons : X → list X → list X.
```

The definition provides three constructors:

$$\begin{aligned} list &: Type \to Type \\ nil &: \forall X : Type.\ list\ X \\ cons &: \forall X : Type.\ X \to list\ X \to list\ X \end{aligned}$$

With implicit arguments mode switched on, the type argument of *cons* is declared implicit. This is not the case for the type argument of *nil* since there is no other argument where the argument can be obtained from. So we use the arguments command to declare the argument of *nil* as implicit.

```
Arguments nil {X}.
```

Now Coq will try to derive the argument of *nil* from the context surrounding an occurrence of *nil*. For instance:

```
Set Printing All.
Check cons 1 nil.
% @cons nat (S O) (@nil nat) : list nat
Unset Printing All.
```

We define an infix notation for *cons*.

```
Notation "x :: y" := (cons x y) (at level 60, right associativity).

Set Printing All.
Check 1::2::nil.
% @cons nat (S O) (@cons nat (S (S O)) (@nil nat)) : list nat
Unset Printing All.
```

We also define the bracket notation for lists.

```
Notation "[]" := nil.
Notation "[ x ]" := (cons x nil).
Notation "[ x ; .. ; y ]" := (cons x .. (cons y nil) ..).

Set Printing All.
Check [1;2].
% @cons nat (S O) (@cons nat (S (S O)) (@nil nat)) : list nat
Unset Printing All.
```

Using informal notation, we define functions yielding the **length**, the **concatenation**, and the **reversal** of lists.

$$
\begin{aligned}
|[x_1;\ldots;x_n]| &:= n \\
[x_1;\ldots;x_m] +\!\!+ [y_1;\ldots;y_n] &:= [x_1;\ldots;x_m;y_1;\ldots;y_n] \\
rev\,[x_1;\ldots;x_n] &:= [x_n;\ldots;x_1]
\end{aligned}
$$

The formal definitions of these functions replace the dot-dot-dot notation with structural recursion on the constructor representation of lists. The idea is expressed with the following equations.

$$
\begin{aligned}
|nil| &= 0 \\
|x :: A| &= 1 + |A| \\
\\
nil +\!\!+ B &= B \\
x :: A +\!\!+ B &= x :: (A +\!\!+ B) \\
\\
rev\ nil &= nil \\
rev\ (x :: A) &= rev\ A +\!\!+ [x]
\end{aligned}
$$

The Coq definitions are now straightforward.

```
Fixpoint length (X : Type) (A : list X) : nat :=
  match A with
    | nil ⇒ O
    | _ :: A' ⇒ S (length A')
  end.

Notation "| A |" := (length A) (at level 70).
```

```
Fixpoint app (X : Type) (A B : list X) : list X :=
  match A with
    | nil ⇒ B
    | x :: A' ⇒ x :: app A' B
  end.

Notation "x ++ y" := (app x y) (at level 60, right associativity).

Fixpoint rev  (X : Type) (A : list X) : list  X :=
  match A with
    | nil ⇒ nil
    | x :: A' ⇒ rev A' ++ [x]
  end.

Compute rev [1;2;3].
% [3;2;1] : list nat
```

Properties of the list operations can be shown by structural induction on lists, which has much in common with structural induction on numbers.

```
Lemma app_nil (X : Type) (A : list X) :
  A ++ nil = A.

Proof.
  induction A as [|x A] ; simpl.
  - reflexivity.
  - rewrite IHA. reflexivity.
Qed.
```

Note that the script applies the induction tactic with an annotation specifying the variable names to be used in the inductive step. Try out what happens if you replace x with b and A with B. The vertical bar in the annotation separates the base case of the induction from the inductive step.

Lists are provided through the standard module *List*. The following commands load the module and the notations coming with it.

```
Require Import List.
Import ListNotations.
Notation "| A |" := (length A) (at level 70).
```

This gives you everything we have defined so far. The notation command defines the notation for *length*, which is not defined in the standard library.

Exercise 1.10.1 Prove the following lemmas.

```
Lemma app_assoc (X : Type) (A B C : list X) :
  (A ++ B) ++ C = A ++ (B ++ C).

Lemma length_app (X : Type) (A B : list X) :
  |A ++ B| = |A| + |B|.
```

```
Lemma rev_app (X : Type) (A B : list X) :
  rev (A ++ B) = rev B ++ rev A.
Lemma rev_rev (X : Type) (A : list X) :
  rev (rev A) = A.
```

1.11 Quantified Inductive Hypotheses

So far the inductive hypotheses of our inductive proofs were plain equations. We will now see inductive proofs where the inductive hypothesis is a universally quantified equation, and where the quantification is needed for the proof to go through. As examples we consider correctness proofs for tail-recursive variants of recursive functions.

If you are familiar with functional programming, you will know that the function *rev* defined in the previous section takes quadratic time to reverse a list. This is due to the fact that each recursion step involves an application of the function *app*. One can write a tail-recursive function that reverses lists in linear time. The trick is to move the elements of the main list to a second list passed as an additional argument.

```
Fixpoint revi (X : Type) (A B : list X) : list X :=
  match A with
    | nil ⇒ B
    | x :: A' ⇒ revi A' (x :: B)
  end.
```

The following lemma gives us a non-recursive characterization of *revi*.

```
Lemma revi_rev (X : Type) (A B : list X) :
  revi A B = rev A ++ B.
```

We prove this lemma by induction on A. For the induction to go through, the inductive hypothesis must hold for all lists B. To get this property, we move the universal quantification for B from the assumptions to the claim before we start the induction. We use the tactic *revert* to move the quantification.

```
Proof.
  revert B. induction A as [|x A] ; simpl.
  - reflexivity.
  - intros B. rewrite IHA. rewrite app_assoc. simpl. reflexivity.
Qed.
```

Step through the script to see how the proof works. The tactic *intros B* moves the universal quantification of B from the claim back to the assumptions.

Exercise 1.11.1 Prove the following lemma.

```
Lemma rev_revi (X : Type) (A : list X) :
  rev A = revi A nil.
```

Note that the lemma tells us how we can reverse lists with *revi*.

Exercise 1.11.2 Here is a tail-recursive function obtaining the length of a list with an additional argument.

```
Fixpoint lengthi (X : Type) (A : list X) (n : nat) :=
  match A with
    | nil ⇒ n
    | _ :: A' ⇒ lengthi  A' (S n)
  end.
```

Proof the following lemmas. The tactic *omega* will be helpful.

```
Lemma lengthi_length  X (A : list X) n :
  lengthi A n = |A| + n.

Lemma length_lengthi X (A : list X) :
  |A| = lengthi A 0.
```

Exercise 1.11.3 Define a factorial function *fact* and a tail-recursive function *facti* that computes factorials using an additional argument. Prove $fact\ n = facti\ n\ 1$ for all n. Use the tactic *omega* and the lemmas *mult_plus_distr_l*, *mult_plus_distr_r*, *mult_assoc*, and *mult_comm* from the standard library.

1.12 Iteration as Polymorphic Higher-Order Function

We now define a function *iter* that yields $f^n x$ given n, f, and x. Speaking procedurally, $f^n x$ is obtained from x by applying n-times the function f. We base the definition of *iter* on two equations:

$$\begin{aligned} iter\ 0\ f\ x &= x \\ iter\ (S\ n)\ f\ x &= f\,(iter\ n\ f\ x) \end{aligned}$$

For the definition of *iter* we need the type of x. Since this can be any type, we take the type of x as argument.

```
Fixpoint iter (n : nat) (X : Type) (f : X → X) (x : X) : X :=
  match n with
    | 0 ⇒ x
    | S n' ⇒ f (iter n' f x)
  end.
```

Since we are working in implicit arguments mode, the type argument X of *iter* is implicit.

The function *iter* formulates a recursion principle known as iteration or primitive recursion. It also serves us as an example of a polymorphic higher-order function. A function is **polymorphic** if it takes a type as argument, and **higher-order** if it takes a function as argument.

Many operations can be expressed with *iter*. We consider addition.

```
Lemma iter_plus x y :
  x + y = iter x S y.
Proof. induction x ; simpl ; congruence. Qed.
```

Note the use of the automation tactic *congruence*. This tactic can finish off proofs if rewriting with unquantified equations and reflexivity suffice.

Subtraction is another operation that can be expressed with *iter*.

```
Lemma iter_minus x y :
  x−y = iter y pred x.
Proof. induction y ; simpl ; omega. Qed.
```

The minus notation and the predecessor function *pred* are from the standard library. Use the commands *locate* and *Print* to find out more.

The standard library provides *iter* under the name *nat_iter*.

Exercise 1.12.1 Prove the following lemma:

```
Lemma iter_mult x y :
  x * y = iter x (plus y) 0.
```

Exercise 1.12.2 Prove the following lemma:

```
Lemma iter_shift X (f : X → X) x n :
  iter  (S n) f  x = iter n f (f x)
```

Exercise 1.12.3 Define a function *power* computing powers x^n and prove the following lemma.

```
Lemma iter_power x n :
  power x n = iter n (mult x) 1.
```

Exercise 1.12.4 *iter* can compute factorials by iterating on pairs.

$$(0,0!) \to (1,1!) \to (2,2!) \to \cdots \to (n,n!)$$

Write a factorial function *fact* and a step function *step* such that you can prove the following lemmas.

```
Lemma iter_fact_step n :
  step (n, fact n) = (S n, fact (S n)).
Lemma iter_fact' n :
  iter n step (O,1) = (n, fact n).
Lemma iter_fact n :
  fact n = snd (iter n step (O,1)).
```

Exercise 1.12.5 We can see *iter n* as a functional representation of the number n carrying with it the structural recursion coming with n. The type of the functional representations is as follows.

```
Definition Nat := ∀ X : Type, (X → X) → X → X.
```

Write conversion functions *encode* : *nat* → *Nat* and *decode* : *Nat* → *nat* and prove *decode* (*encode n*) = n for every number n.

1.13 Options and Finite Types

We will define a function that for a number n yields a type with n elements. The function will start from an empty type and n-times apply a function that for a given type yields a type with one additional element.

Coq's standard library defines an empty type *Empty_set* as an inductive type without constructors:

```
Inductive Empty_set : Type := .
```

Since *Empty_set* is empty, it is inconsistent to assume that it has an element. In fact, if we assume that *Empty_set* has an element, we can prove everything. For instance:

```
Lemma vacuous_truth (x : Empty_set) :
  1 = 2.
Proof. destruct x. Qed.
```

The proof is by case analysis over the assumed element x of *Empty_set*. Since *Empty_set* has no constructor, we can prove the claim $1 = 2$ for every constructors of *Empty_set*. One says that the claim follows vacuously. Vacuous reasoning is a basic logical principle.[2]

The type constructor *option* from the standard library can be applied to any type and yields a type with one additional element.

[2] From Wikipedia: A vacuous truth is a truth that is devoid of content because it asserts something about all members of a class that is empty or because it says "If A then B" when in fact A is inherently false. For example, the statement "all cell phones in the room are turned off" may be true simply because there are no cell phones in the room. In this case, the statement "all cell phones in the room are turned on" would also be considered true, and vacuously so.

```
Inductive option (X : Type) : Type  :=
| Some : X → option X
| None : option X.
```

The constructor *Some* yields the elements of X and the constructor *None* yields the new element (none of the old elements). The elements of an option type are called *options*. The standard library declares the type argument X of both *Some* and *None* as implicit argument (check with *Print*).

We can now define a function $fin : nat \to Type$ such that $fin\ n$ is a type with n elements.

```
Definition fin (n : nat) : Type :=
  nat_iter n option Empty_set.
```

Here are definitions naming the elements of the types *fin 1*, *fin 2*, and *fin 3*.

```
Definition a11 : fin 1 := @None Empty_set.
Definition a21: fin 2  := Some a11.
Definition a22 : fin 2 := @None (fin 1).
Definition a31: fin 3 := Some a21.
Definition a32 : fin 3 := Some a22.
Definition a33 : fin 3 := @None (fin 2).
```

For clarity we have specified the implicit argument of *None*. You may omit the implicit arguments and leave it to Coq to insert them. Next we establish three simple facts about finite types.

```
Goal ∀ n, fin (2+n) = option (fin (S n)).
Proof. intros n. reflexivity. Qed.
Goal ∀ m n, fin (m+n) = fin (n+m).
Proof.
  intros m n. f_equal. omega.
Qed.
Lemma fin1 (x : fin 1) :
  x = None.
Proof.
  destruct x as [x|].
  - simpl in x. destruct x.
  - reflexivity.
Qed.
```

Exercise 1.13.1 One can define a bijection between *bool* and *fin 2*. Show this fact by completing the definitions and proving the lemmas shown below.

```
Definition fromBool (b : bool) : fin 2 :=
Definition toBool (x : fin 2) : bool :=
Lemma bool_fin b :  toBool (fromBool b) = b.
Lemma fin_bool x :  fromBool (toBool x) = x.
```

Exercise 1.13.2 One can define a bijection between *nat* and *option nat*. Show this fact by defining functions *fromNat* and *toNat* and by proving that they commute.

Exercise 1.13.3 In Coq every function is total. Option types can be used to express partial functions as total functions.

a) Define a function $find : \forall X : Type,\ (X \to bool) \to list\ X \to option\ X$ that given a test p and a list A yields an element of A satisfying p if there is one.

b) Define a function $minus\_opt : nat \to nat \to option\ nat$ that yields $x - y$ if $x \geq y$ and *None* otherwise.

1.14 More about Functions

Functions are objects of our imagination. A function relates arguments with results, where an argument is related with at most one result. One says that functions map arguments to results.

Functions in Coq are very general in that they can take functions and types as arguments and yield functions and types as results. For instance:

- The type constructor *list* is a function that maps types to types.
- The value constructor *nil* is a function that maps types to lists.
- The function *plus* maps numbers to functions that map numbers to numbers.
- The function *fin* maps numbers to types.

Coq describes functions, arguments, and results with syntactic objects called terms. There are four canonical forms for terms describing functions:

1. A lambda abstraction $\lambda x : s.t$.
2. A recursive abstraction $fix\ f\ (x : s) : t := u$.
3. A constructor c.
4. An application $c\ t_1 \cdots t_n$ of a constructor c to $n \geq 1$ terms $t_1, \ldots, t_n$.

The general form of a function type is $\forall x : s.t$. A function of type $\forall x : s.t$ relates every element x of type s with exactly one element of type t. One speaks of a **dependent function type** if the argument variable x appears in t. If x does not appear in t, there is no dependency and $\forall x : s.t$ is written as $s \to t$.

```
Check ∀ x : nat, nat.
% nat → nat : Type
```

In Coq, every function has a unique type. Here are examples of functions and

their types:

$$\begin{aligned} \mathit{andb} &: \mathit{bool} \to \mathit{bool} \to \mathit{bool} \\ \mathit{cons} &: \forall X : \mathit{Type},\ X \to \mathit{list}\ X \to \mathit{list}\ X \\ \mathit{iter} &: \mathit{nat} \to \forall X : \mathit{Type},\ (X \to X) \to X \to X \\ \mathit{fin} &: \mathit{nat} \to \mathit{Type} \end{aligned}$$

The functions *andb*, *cons*, and *iter* are cascaded, which means that they yield a function when applied to an argument. One says that a cascaded function takes more than one argument. The function *andb* takes 2 arguments, and *cons* takes 3 arguments. The function *iter* is more interesting. It takes at least 4 arguments, but it may take additional arguments if the second argument is a function type:

$$\begin{aligned} \mathit{iter}\ 2\ \mathit{nat} &: (\mathit{nat} \to \mathit{nat}) \to \mathit{nat} \to \mathit{nat} \\ \mathit{iter}\ 2\ (\mathit{nat} \to \mathit{nat}) &: ((\mathit{nat} \to \mathit{nat}) \to \mathit{nat} \to \mathit{nat}) \to (\mathit{nat} \to \mathit{nat}) \to \mathit{nat} \to \mathit{nat} \end{aligned}$$

One says that a function of type $\forall x : s.t$ is *polymorphic* if x ranges over types. The constructors *nil* and *cons* are typical examples of polymorphic functions. The function *iter* yields a polymorphic function for every argument.

Coq comes with reduction rules for terms. A reduction rule describes a computation step. Coq is designed such that the application of reduction rules to terms always terminates with a unique normal form. We say that a term evaluates to its normal form. We have seen four reduction rules so far:

- The application of a lambda abstraction to a term can always be reduced (beta reduction).
- A match on a constructor or the application of a constructor can always be reduced.
- A defined name can always be reduced to the term the name is bound to (unfolding).
- The application of a recursive abstraction to a constructor or the application of a constructor can always be reduced.

Coq differs from functional programming languages in that its type discipline is more general and in that it restricts recursion to structural recursion. In Coq, types are first-class values and polymorphic types are first-class types, which is not the case in functional programming languages. On the other hand, recursion in Coq is always tied to an inductive type and every recursion step must take off at least one constructor.

1.15 Discussion and Remarks

A basic feature of Coq's language are inductive types. We have introduced inductive types for booleans, natural numbers, pairs, and lists. The elements of inductive types are obtained with so-called constructors. Inductive types generalize the structure underlying the Peano axioms for the natural numbers. Inductive types are a basic feature of modern functional programming languages (e.g., ML and Haskell). The first functional programming language with inductive types was Hope, developed in the 1970's in Edinburgh by Rod Burstall and others.

Inductive types are accompanied by structural case analysis, structural recursion, and structural induction. Typical examples of recursive functions are addition and multiplication of numbers and concatenation and reversal of lists. We have also seen a polymorphic higher-order function *iter* formulating a recursion scheme known as iteration.

Coq is designed such that evaluation always terminates. For this reason Coq restricts recursion to structural recursion on inductive types. Every recursion step must take off at least one constructor of a given argument.

The idea of cascaded functions appeared 1924 in a paper by Moses Schönfinkel and was fully developed in the 1930's by Alonzo Church and Haskell Curry. Lambda abstractions and beta reduction were first studied in the 1930's by Alonzo Church and his students in an untyped syntactic system called lambda calculus. The idea of dependent function types evolved in the 1970's in the works of Nicolaas de Bruijn, Jean-Yves Girard, and Per Martin-Löf.

Coq comes with many notational devices including user-defined infix notations and implicit arguments. It is very important to distinguish between notational conveniences and abstract syntax. Notational conveniences are familiar from mathematics and make it possible for humans to work with complex terms. However, all semantic issues and all logical reasoning are defined on the abstract syntax where all conveniences are removed and all details are filled in.

Coq supports the formulation and the proof of theorems. So far we have just seen the tip of the iceberg. We have formulated equational theorems and used case analysis, induction, and rewriting to prove them. In Coq, Proofs are constructed by scripts, which are obtained with commands called tactics. A tactic either resolves a proof goal or reduces a proof goal to one or several subgoals. Proof scripts are constructed in interaction with Coq, where Coq applies the proof rules and maintains and displays the open subgoals.

Proof scripts are programs that construct proofs. To understand a proof, one steps with the Coq interpreter through the script constructing the proof and looks at the proof goals obtained with the tactics. Eventually, we will learn that Coq represents proofs as terms. If you are curious, you may use the command

Print L to see the term serving as the proof of a lemma *L*.

Coq Summary

Type and Value Constructors from the Standard Library

bool, *true*, *false*, *nat*, *O*, *S*, *prod*, *pair*, *list*, *nil*, *cons*, *Empty_set*, *option*, *Some*, *None*.

Defined Functions from the Standard Library

negb, *andb*, *pred*, *plus*, *mult*, *minus*, *nat_iter*, *length*, *app*, *rev*.

Term Variants

Names, applications, matches (*match*), lambda abstractions (*fun*), recursive abstractions (*fix*).

Definitional Commands

Inductive, *Definition*, *Fixpoint*, *Lemma*, *Example*, *Goal*, *Proof*, *Qed*.

Tactics

destruct, *induction*, *simpl*, *unfold*, *reflexivity*, *symmetry*, *f_equal*, *rewrite*, *setoid_rewrite*, *apply*, *intros*, *revert*, *congruence*, *omega*.

Notational Commands

Notation, *Arguments*, *Set Implicit Arguments*, *Unset Strict Implicit*.

Module Commands

Require Import, *Import*.

Query Commands

Check, *Compute*, *Print*, *About*, *Locate*, *Set*/*Unset Printing All*.

Make sure that for each of the above constructs you can point to examples in the text of this chapter. To know more, consult the Coq online documentation at coq.inria.fr.

2 Propositions and Proofs

Logical statements are called propositions in Coq. So far we have only seen equational propositions. We now extend our repertoire to propositions involving connectives and quantifiers.

2.1 Logical Connectives and Quantifiers

When we argue logically, we often combine primitive propositions into compound propositions using logical operations. The logical operations include connectives like implication and quantifiers like "for all". Here is an overview of the logical operations we will consider.

Operation	Notation	Reading
conjunction	$A \wedge B$	A and B
disjunction	$A \vee B$	A or B
implication	$A \to B$	if A, then B
equivalence	$A \leftrightarrow B$	A if and only if B
negation	$\neg A$	not A
universal quantification	$\forall x : T. A$	for all x in T, A
existential quantification	$\exists x : T. A$	for some x in T, A

There are two different ways of assigning meaning to logical operations and propositions. The **classical approach** commonly used in mathematics postulates that every proposition has a truth value that is either true or false. The more recent **constructive approach** defines the meaning of propositions in terms of their proofs and does not rely on truth values. Coq and our presentation of logic follow the constructive approach. The cornerstone of the constructive approach is the **BHK interpretation**,[1] which relates proofs and logical operations as follows.

- A proof of $A \wedge B$ consists of a proof of A and a proof of B.
- A proof of $A \vee B$ is either a proof of A or a proof of B.
- A proof of $A \to B$ is a function that for every proof of A yields a proof of B.

[1] The name BHK interpretation reflects the origin of the scheme in the work of the mathematicians Luitzen Brouwer, Arend Heyting, and Andrey Kolmogorov in the 1930's.

- A proof of $\forall x : T.A$ is a function that for every $x : T$ yields a proof of A.
- A proof of $\exists x : T.A$ consists of a term $s : T$ and a proof of A_s^x.

The notation A_s^x stands for the proposition obtained from the proposition A by replacing the variable x with the term s. One speaks of a **substitution** and says that s is **substituted** for x. Equivalence and negation are missing in the above list since they are definable with other connectives:

$$A \leftrightarrow B \;:=\; (A \to B) \wedge (B \to A)$$
$$\neg A \;:=\; A \to \bot.$$

The symbol $\bot$ represents the primitive proposition **false** that has no proof. To give a proof of $\neg A$ we thus have to give a function that yields for every proof of A a proof of $\bot$. If such a function exists, no proof of A can exist since no proof of false exists.

In this chapter we will learn how Coq accommodates the logical operations and the concomitant proof rules. We start with implication and universal quantification.

2.2 Implication and Universal Quantification

Example: Symmetry of Equality

We begin with the proof of a proposition saying that equality is symmetric.

```
Goal ∀ (X : Type) (x y : X), x=y → y=x.
Proof. intros X x y A. rewrite A. reflexivity. Qed.
```

The command *Goal* is like the command *Lemma* but leaves it to Coq to choose a name for the lemma. The tactic *intros* takes away the universal quantifications and the implication of the claim by representing the respective assumptions as explicit assumptions of the proof goal.

$$\frac{\begin{array}{l} X : \mathit{Type} \\ x : X \\ y : X \\ A : x = y \end{array}}{y = x}$$

The rest of the proof is straightforward since we have the assumption $A : x = y$ saying that A is a proof of the equation $x = y$. The proof A can be used to rewrite the claim $y = x$ into the trivial equation $y = y$.

Recall the *revert* tactic and note that *revert* can undo the effect of *intros*.

Exercise 2.2.1 Prove the following goal.

Goal ∀ x y, andb x y = true → x = true.

Example: Modus Ponens

Our second example is a proposition stating a basic law for implication known as *modus ponens*.

Goal ∀ X Y : Prop, X → (X → Y) → Y.

Proof. intros X Y x A. exact (A x). **Qed**.

The proposition quantifies over all propositions X and Y since *Prop* is the type of all propositions. The proof first takes away the universal quantifications and the outer implications[2] leaving us with the goal

$$\frac{\begin{array}{l} X : \mathit{Prop} \\ Y : \mathit{Prop} \\ x : X \\ A : X \to Y \end{array}}{Y}$$

Given that we have a proof A of $X \to Y$ and a proof x of X, we obtain a proof of the claim Y by applying the function A to the proof x.[3] Coq accommodates this reasoning with the tactic *exact*.

Example: Transitivity of Implication

Goal ∀ X Y Z : Prop, (X → Y) → (Y → Z) → X → Z.

Proof. intros X Y Z A B x. exact (B (A x)). **Qed**.

Exercise 2.2.2 Prove that equality is transitive.

2.3 Predicates

Functions that eventually yield a proposition are called **predicates**. With predicates we can express properties and relations. Here is a theorem involving two predicates p and q and a nested universal quantification.

Goal ∀ p q : nat → Prop, p 7 → (∀ x, p x → q x) → q 7.

[2] Like the arrow for function types the arrow for implication adds missing parentheses to the right, that is, $X \to (X \to Y) \to Y$ elaborates to $X \to ((X \to Y) \to Y)$.

[3] Recall from Section 2.1 that proofs of implications are functions.

Proof. intros p q A B. exact (B 7 A). **Qed**.

Think of p and q as properties of numbers. After the intros we have the goal

$$\frac{\begin{array}{l} p : nat \to Prop \\ q : nat \to Prop \\ A : p\,7 \\ B : \forall x,\ p\,x \to q\,x \end{array}}{q\,7}$$

The proof now exploits the fact that B is a function that yields a proof of $q\,7$ when applied to 7 and a proof of $p\,7$.

2.4 The Apply Tactic

The tactic *apply* applies proofs of implications in a backward manner.

Goal ∀ X Y Z : Prop, (X → Y) → (Y → Z) → X → Z.

Proof. intros X Y Z A B x. apply B. apply A. exact x. **Qed**.

The tactic *apply* also works for universally quantified implications.

Goal ∀ p q : nat → Prop, p 7 → (∀ x, p x → q x) → q 7.

Proof. intros p q A B. apply B. exact A. **Qed**.

Step through the proofs with Coq to understand.

Exercise 2.4.1 Prove the following goals.

Goal ∀ X Y, (∀ Z, (X → Y → Z) → Z) → X.

Goal ∀ X Y, (∀ Z, (X → Y → Z) → Z) → Y.

Exercise 2.4.2 Prove the following goals, which express essential properties of booleans, numbers, and lists.

Goal ∀ (p : bool → Prop) (x : bool),
p true → p false → p x.

Goal ∀ (p : nat → Prop) (x : nat),
p O → (∀ n, p n → p (S n)) → p x.

Goal ∀ (X : Type) (p : list X → Prop) (xs : list X),
p nil → (∀ x xs, p xs → p (cons x xs)) → p xs.

Hint: Use case analysis and induction.

2.5 Leibniz Characterization of Equality

What does it mean that two objects are equal? The mathematician and philosopher Leibniz answered this question in an interesting way: Two objects are equal if they have the same properties. We know enough to prove in Coq that Leibniz was right.

```
Goal ∀ (X : Type) (x y : X),
(∀ p : X → Prop, p x → p y) → x=y.
Proof. intros X x y A. apply (A (fun z ⇒ x=z)). reflexivity. Qed.
```

Run the proof with Coq to understand. After the intros we have the goal

$$\frac{\begin{array}{l} X : \mathit{Type} \\ x : X \\ y : X \\ A : \forall p : X \to \mathit{Prop}.\ \ px \to py \end{array}}{x = y}$$

Applying the proof A to the predicate $\lambda z.x{=}z$ gives us a proof of the implication $x{=}x \to x{=}y$.[4] Backward application of this proof reduces the claim to the trivial claim $x{=}x$, which can be established with reflexivity.

Exercise 2.5.1 Prove the following goals.

```
Goal ∀ (X : Type) (x y : X),
x=y → ∀ p : X → Prop, p x → p y.
```

```
Goal ∀ (X : Type) (x y : X),
(∀ p : X → Prop, p x → p y) →
forall p : X → Prop, p y → p x.
```

2.6 Propositions are Types

You may have noticed that Coq's notations for implications and universal quantifications are the same as the notations for function types. This goes well with our assumption that the proofs of implications and universal quantifications are functions (see Section 2.1). The notational coincidence is profound and reflects the *propositions as types principle*, which accommodates propositions as types taking the proofs of the propositions as members. The propositions as types principle is also known as *Curry-Howard correspondence* after two of its inventors.

[4] $\lambda z.\, x{=}z$ is the mathematical notation for the function *fun z => x=z*, which for z yields the equation $x{=}z$.

There is a special universe *Prop* that takes exactly the propositions as members. Universes are types that take types as members. *Prop* is a subuniverse of the universe *Type*. Consequently, every member of *Prop* is a member of *Type*.

A function type $s \to t$ is actually a function type $\forall x : s.\, t$ where the variable x does not occur in t. Thus an implication $s \to t$ is actually a quantification $\forall x : s.\, t$ saying that for every proof of s there is a proof of t. Note that the reduction of implications to quantifications rests on the ability to quantify over proofs. Constructive type theory has this ability since proofs are first-class citizens that appear as members of types in the universe *Prop*.

The fact that implications are universal quantifications explains why the tactics *intros* and *apply* are used for both implications and universal quantifications.

Given a function type $\forall x : s.\, t$, we call x a *bound variable*. What concrete name is chosen for a bound variable does not matter. Thus the notations $\forall X : Type.\, X$ and $\forall Y : Type.\, Y$ denote the same type. Moreover, if we have a type $\forall x : s.\, t$ where x does not occur in t, we can omit x and just write $s \to t$ without losing information. That the concrete names of bound variables do not matter is a basic logic principle.

Exercise 2.6.1 Prove the following goals in Coq. Explain what you see.

```
Goal ∀ X : Type,
(fun x : X ⇒ x) = (fun y : X ⇒ y)

Goal ∀ X Y : Prop,
(X → Y) → ∀ x : X, Y.

Goal ∀ X Y : Prop,
(∀ x : X, Y) → X → Y.

Goal ∀ X Y : Prop,
(X → Y) = (∀ x : X, Y).
```

2.7 Falsity and Negation

Coq comes with a proposition *False* that by itself has no proof. Given certain assumptions, a proof of *False* may however become possible. We speak of **inconsistent assumptions** if they make a proof of *False* possible. There is a basic logic principle called **explosion** saying that from a proof of *False* one can obtain a proof of every proposition. Coq provides the explosion principle through the tactic *contradiction*.

```
Goal False → 2=3.

Proof. intros A. contradiction A. Qed.
```

We also refer to the proposition *False* as **falsity**. The logical notation for *False* is $\bot$. With falsity Coq defines **negation** as $\neg s := s \to \bot$. So we can prove $\neg s$ by assuming a proof of s and constructing a proof of $\bot$.

```
Goal ∀ X : Prop, X → ¬¬ X.
```

```
Proof. intros X x A. exact (A x). Qed.
```

The proof script works since Coq automatically unfolds the definition of negation. The double negation $\neg\neg X$ unfolds into $(X \to \bot) \to \bot$. Here is another example.

```
Goal ∀ X : Prop,
(X → ¬ X) → (¬ X → X) → False.
```

```
Proof.
  intros X A B. apply A.
  - apply B. intros x. exact (A x x).
  - apply B. intros x. exact (A x x).
Qed.
```

Sometimes the tactic *exfalso* is helpful. It replaces the claim with $\bot$, which is justified by the explosion principle.

```
Goal ∀ X : Prop,
¬¬ X → (X → ¬ X) → X.
```

```
Proof. intros X A B. exfalso. apply A. intros x. exact (B x x). Qed.
```

Exercise 2.7.1 Prove the following goals.

```
Goal ∀ X : Prop, ¬¬¬ X → ¬ X.
```

```
Goal ∀ X Y : Prop, (X → Y) → ¬ Y → ¬ X.
```

Exercise 2.7.2 Prove the following goals.

```
Goal ∀ X : Prop, ¬¬ (¬¬ X → X).
```

```
Goal ∀ X Y : Prop, ¬¬ (((X → Y) → X) → X).
```

Exercise 2.7.3 Prove the following proposition in Coq without using only the tactic *exact*.

```
Goal ∀ X:Prop,
(X → False) → (¬ X → False) → False.
```

2.8 Conjunction and Disjunction

The tactics for conjunctions are *destruct* and *split*.

```
Goal ∀ X Y : Prop, X ∧ Y → Y ∧ X.

Proof.
  intros X Y A. destruct A as [x y]. split.
  – exact y.
  – exact x.
Qed.
```

The tactics for disjunctions are *destruct*, *left*, and *right*.

```
Goal ∀ X Y : Prop, X ∨ Y → Y ∨ X.

Proof.
  intros X Y A. destruct A as [x|y].
  – right. exact x.
  – left. exact y.
Qed.
```

Run the proof scripts with Coq to understand. Note that we can prove a conjunction $s \wedge t$ if and only if we can prove both s and t, and that we can prove a disjunction $s \vee t$ if and only if we can prove either s or t.

The *intros* tactic destructures proofs when given a destructuring pattern. This leads to shorter proof scripts.

```
Goal ∀ X Y : Prop, X ∧ Y → Y ∧ X.

Proof.
  intros X Y [x y]. split.
  – exact y.
  – exact x.
Qed.

Goal ∀ X Y : Prop, X ∨ Y → Y ∨ X.

Proof.
  intros X Y [x|y].
  – right. exact x.
  – left. exact y.
Qed.
```

Nesting of destructuring patterns is possible:

```
Goal ∀ X Y Z : Prop,
X ∨ (Y ∧ Z) → (X ∨ Y) ∧ (X ∨ Z).

Proof.
  intros X Y Z [x|[y z]].
  – split; left; exact x.
```

```
  - split; right.
    + exact y.
    + exact z.
Qed.
```

Note that the bullet + is used to indicate proofs of subgoals of the last main subgoal. One can use three levels of bullets, − for top level subgoals, + for second level subgoals, and ∗ for third level subgoals. One can also separate part of a proof using curly braces {· · · } inside which one can restart using the bullets −, +, and ∗. In this way Coq supports an arbitrary number of subgoal levels.

Exercise 2.8.1 Prove the following goals.

```
Goal ∀ X : Prop,
¬ (X ∨ ¬ X) → X ∨ ¬ X.

Goal ∀ X : Prop,
(X ∨ ¬ X → ¬ (X ∨ ¬ X)) → X ∨ ¬ X.

Goal ∀ X Y Z W : Prop,
(X → Y) ∨ (X → Z) → (Y → W) ∧ (Z → W) → X → W.
```

Exercise 2.8.2 Prove the following goals.

```
Goal ∀ X : Prop, ¬¬ (X ∨ ¬ X).

Goal ∀ X Y : Prop, ¬¬ ((X → Y) → ¬ X ∨ Y).
```

2.9 Equivalence and Rewriting

Coq defines equivalence as $s \leftrightarrow t := (s \to t) \wedge (t \to s)$. Thus an equivalence $s \leftrightarrow t$ is provable if and only if the implications $s \to t$ and $t \to s$ are both provable. Coq automatically unfolds equivalences.

```
Lemma and_com : ∀ X Y : Prop, X ∧ Y ↔ Y ∧ X.

Proof.
  intros X Y. split.
  - intros [x y]. split.
    + exact y.
    + exact x.
  - intros [y x]. split.
    + exact x.
    + exact y.
Qed.

Lemma deMorgan : ∀ X Y : Prop, ¬ (X ∨ Y) ↔ ¬ X ∧ ¬ Y.
```

```
Proof.
  intros X Y. split.
  - intros A. split.
    + intros x. apply A. left. exact x.
    + intros y. apply A. right. exact y.
  - intros [A B] [x|y].
    + exact (A x).
    + exact (B y).
Qed.
```

One can use the tactic *apply* with equivalences. Since an equivalence is a conjunction of implications, the *apply* tactic will choose one of the two implications to use. The user can choose which of the two implications to use by using the tactic *apply* with one of the arrows → and ← (similar to the tactic *rewrite*).

One can often reason with equivalences in the same ways as with equations. Part of the justification for this is the fact that logical equivalence is an equivalence relation (i.e., it is reflexive, symmetric and transitive). A number of lemmas can justify rewriting with equivalences in many (but not all) contexts. For example, the following result allows us rewrite with equivalences below conjunctions.

```
Goal ∀ X Y Z W : Prop, (X ↔ Y) → (Z ↔ W) → (X ∧ Z ↔ Y ∧ W).
```

We leave the proof of this goal as an exercise.

In contexts where rewriting with equivalences is allowed, we may use the tactic *setoid_rewrite*.[5]

```
Goal ∀ X Y Z : Prop, ¬ (X ∨ Y) ∧ Z ↔ Z ∧ ¬ X ∧ ¬ Y.
Proof.
 intros X Y Z.
 setoid_rewrite deMorgan.
 apply and_com.
Qed.
Goal ∀ X : Type, ∀ p q : X → Prop, (∀ x, ¬ (p x ∨ q x)) → ∀ x, ¬ p x ∧ ¬ q x.
Proof.
 intros X p q A.
 setoid_rewrite ← deMorgan.
 exact A.
Qed.
```

One can also use the tactics *reflexivity*, *symmetry* and *transitivity* to reason about equivalences.

```
Goal ∀ X : Prop, X ↔ X.
Proof. reflexivity. Qed.
```

[5] Recall that the tactic *setoid_rewrite* is provided by the standard library module *Omega*.

```
Goal ∀ X Y : Prop, (X ↔ Y) → (Y ↔ X).
Proof. intros X Y A. symmetry. exact A. Qed.
Goal ∀ X Y Z : Prop, (X ↔ Y) → (Y ↔ Z) → (X ↔ Z).
Proof.
 intros X Y Z A B. transitivity Y.
 – exact A.
 – exact B.
Qed.
```

Proof scripts done using the tactics *setoid_rewrite*, *reflexivity*, *symmetry*, and *transitivity* to reason with equivalences can always be replaced by proof scripts that do not use these tactics. Some of the exercises below should give the reader an idea how such a replacement could be accomplished.

Exercise 2.9.1 Prove equivalence is an equivalence relation without using the tactics *setoid_rewrite*, *reflexivity*, *symmetry* and *transitivity*.

```
Goal ∀ X : Prop, X ↔ X.
Goal ∀ X Y : Prop, (X ↔ Y) → (Y ↔ X).
Goal ∀ X Y Z : Prop, (X ↔ Y) → (Y ↔ Z) → (X ↔ Z).
```

Exercise 2.9.2 Prove the following facts which justify rewriting with equivalences in certain contexts. Do not use the tactics *setoid_rewrite*, *reflexivity*, *symmetry* and *transitivity*.

```
Goal ∀ (X Y Z W : Prop), (X ↔ Y) → (Z ↔ W) → (X ∧ Z ↔ Y ∧ W).
Goal ∀ (X:Type) (p q:X → Prop), (∀ x:X, p x ↔ q x) → ((∀ x:X, p x) ↔ ∀ x:X, q x).
```

Exercise 2.9.3 Prove the following facts using *setoid_rewrite*, *reflexivity*, *symmetry* and *transitivity*. You may use the lemmas *deMorgan* and *and_com*.

```
Goal ∀ X Y Z : Prop, X ∧ ¬ (Y ∨ Z) ↔ (¬ Y ∧ ¬ Z) ∧ X.
Goal ∀ X : Type, ∀ p q : X → Prop, (∀ x, ¬ (p x ∨ q x)) ↔ ∀ x, ¬ p x ∧ ¬ q x.
```

Exercise 2.9.4 Prove the following goals.

```
Goal ∀ X Y : Prop, X ∧ (X ∨ Y) ↔ X.
Goal ∀ X Y : Prop, X ∨ (X ∧ Y) ↔ X.
Goal ∀ X:Prop, (X → ¬ X) → X ↔ ¬¬ X.
```

Exercise 2.9.5 (Impredicative Characterizations) It turns out that falsity, negations, conjunctions, disjunctions, and even equations are all equivalent to propositions obtained with just implication and universal quantification. Prove the following goals to get familiar with this so-called impredicative characterizations.

Goal False ↔ ∀ Z : Prop, Z.

Goal ∀ X : Prop,
¬ X ↔ ∀ Z : Prop, X → Z.

Goal ∀ X Y : Prop, X ∧ Y ↔ ∀ Z : Prop, (X → Y → Z) → Z.

Goal ∀ X Y : Prop, X ∨ Y ↔ ∀ Z : Prop, (X → Z) → (Y → Z) → Z.

Goal ∀ (X : Type) (x y : X), x=y ↔ ∀ p : X → Prop, p x → p y.

2.10 Automation Tactics

Coq provides various automation tactics that help in the construction of proofs. In a proof script, an automation tactic can always be replaced by a sequence of basic tactics.

A simple automation tactic is *assumption*. This tactic solves goals whose claim appears as an assumption.

Goal ∀ X Y : Prop, X ∧ Y → Y ∧ X.

Proof. intros X Y [x y]. split ; assumption. **Qed**.

The automation tactic *auto* is more powerful. It uses the tactics *intros*, *apply*, *assumption*, *reflexivity* and a few others to construct a proof. We may use *auto* to finish up proofs once the goal has become obvious.

Goal ∀ (X : Type) (p : list X → Prop) (xs : list X),
p nil → (∀ x xs, p xs → p (cons x xs)) → p xs.

Proof. induction xs ; auto. **Qed**.

The automation tactic *tauto* solves every goal that can be solved with the tactics *intros* and *reflexivity*, the basic tactics for falsity, implication, conjunction, and disjunction, and the definitions of negation and equivalence.

Goal ∀ X : Prop, ¬ (X ↔ ¬ X).

Proof. tauto. **Qed**.

2.11 Existential Quantification

The tactics for existential quantifications are *destruct* and *exists*.[6]

Goal ∀ (X : Type) (p q : X → Prop),
(∃ x, p x ∧ q x) → ∃ x, p x.

[6] The existential quantifier ∃ can be written as the keyword *exists* in Coq code. When we display Coq code, we always replace the string *exists* with the symbol ∃. For this reason the tactic *exists* appears as the symbol ∃ in Coq code.

```
Proof.
  intros X p q A. destruct A as [x B]. destruct B as [C _].
  ∃ x. exact C.
Qed.
```

Run the proof scripts with Coq to understand.

The **diagonal law** is a simple fact about nonexistence that has amazing consequences. One such consequence is the undecidability of the halting problem. We state the diagonal law as follows:

```
Definition diagonal : Prop := ∀ (X : Type) (p : X → X → Prop),
¬ ∃ x, ∀ y, p x y ↔ ¬ p y y.
```

If X is the type of all Turing machines and pxy says that x halts on the string representation of y, the diagonal law says that there is no Turing machine x such that x halts on a Turing machine y if and only if y does not halt on its string representation.

The proof of the diagonal law is not difficult.

```
Lemma circuit (X : Prop) : ¬ (X ↔ ¬ X).
Proof. tauto. Qed.
Goal diagonal.
Proof. intros X p [x A]. apply (@circuit (p x x)). exact (A x). Qed.
```

We can prove the diagonal law without a lemma if we use the tactic *specialize*.

```
Goal diagonal.
Proof. intros X p [x A]. specialize (A x). tauto. Qed.
```

A **disequation** $s \neq t$ is a negated equation $\neg(s=t)$. We prove the correctness of a characterization of disequality that employs existential quantification.

```
Goal ∀ (X : Type) (x y : X),
x ≠ y ↔ ∃ p : X → Prop, p x ∧ ¬ p y.
Proof.
  split.
  – intros A. ∃ (fun z ⇒ x = z). auto.
  – intros [p [A B]] C. apply B. rewrite ← C. apply A.
Qed.
```

Note that *split* tacitly introduces X, x, and y.

Exercise 2.11.1 Prove the De Morgan law for existential quantification.

```
Goal ∀ (X : Type) (p : X → Prop),
¬ (∃ x, p x) ↔ ∀ x, ¬ p x.
```

Exercise 2.11.2 Prove the exchange rule for existential quantifications.

Goal ∀ (X Y : Type) (p : X → Y → Prop),
(∃ x, ∃ y, p x y) ↔ ∃ y, ∃ x, p x y.

Exercise 2.11.3 (Impredicative Characterization) Prove the following goal. It shows that existential quantification can be expressed with implication and universal quantification.

Goal ∀ (X : Type) (p : X → Prop),
(∃ x, p x) ↔ ∀ Z : Prop, (∀ x, p x → Z) → Z.

Exercise 2.11.4 Below are characterizations of equality and disequality based on reflexive relations. Prove the correctness of the characterizations.

Goal ∀ (X : Type) (x y : X),
x = y ↔ ∀ r : X → X → Prop, (∀ z : X, r z z) → r x y.

Goal ∀ (X : Type) (x y : X),
x ≠ y ↔ ∃ r : X → X → Prop, (∀ z : X, r z z) ∧ ¬ r x y.

Hint for first goal: Use the tactic *specialize* and simplify the resulting assumption with *simpl in A* where *A* is the name of the assumption.

Exercise 2.11.5 Prove the following goal.

Goal ∀ (X: Type) (x : X) (p : X → Prop), ∃ q : X → Prop,
q x ∧ (∀ y, p y → q y) ∧ ∀ y, q y → p y ∨ x = y.

Exercise 2.11.6

a) Prove the following goal.

Goal ∀ (X : Type) (Y : Prop) ,
X → Y ↔ (∃ x : X, True) → Y.

b) Explain why $s \to t$ is a proposition if s is a type and t is a proposition.

2.12 Basic Proof Rules

By now we have conducted many proofs in Coq. In this chapter we mostly proved general properties of the logical connectives and quantifiers. The proofs were constructed with a small set of tactics, where every tactic performs a basic proof step. The proof steps performed by the tactics can be described by the proof rules appearing in Figure 2.1. We may say that the rules describe basic logic principles and that the tactics implement these principles.

Each proof rule says that a proof of the conclusion (the proposition appearing below the line) can be obtained from proofs of the premises (the items appearing

$$\frac{s \Rightarrow t}{s \to t} \qquad \frac{s \to t \quad s}{t}$$

$$\frac{x : s \Rightarrow t}{\forall x : s.\, t} \qquad \frac{\forall x : s.\, t \quad u : s}{t_u^x}$$

$$\frac{\bot}{u}$$

$$\frac{s \quad t}{s \wedge t} \qquad \frac{s \wedge t \quad s, t \Rightarrow u}{u}$$

$$\frac{s}{s \vee t} \qquad \frac{t}{s \vee t} \qquad \frac{s \vee t \quad s \Rightarrow u \quad t \Rightarrow u}{u}$$

$$\frac{u : s \quad t_u^x}{\exists x : s.\, t} \qquad \frac{\exists x : s.\, t \quad x : s,\, t \Rightarrow u}{u}$$

Figure 2.1: Basic proof rules

above the line). The notation $s \Rightarrow t$ used in some premises says that there is a proof of t under the assumption that there is a proof of s. The notation $u : s$ says that the term u has type s, and the notation s_t^x stands for the proposition obtained from s by replacing x with t.

We explain one of the proof rules for disjunctions in detail.

$$\frac{s \vee t \quad s \Rightarrow u \quad t \Rightarrow u}{u}$$

The rule says that we can obtain a proof of a proposition u if we are given a proof of a disjunction $s \vee t$, a proof of u assuming a proof of s, and a proof of u assuming a proof of t. The rule is justified since a proof of the disjunction $s \vee t$ gives us a proof of either s or t. Speaking more generally, the rule tells us that we can do a case analysis if we have a proof of a disjunction. Coq implements the rule in a backward fashion with the tactic *destruct*.

$$\frac{A : s \vee t}{u} \qquad \texttt{destruct A as [B|C]} \qquad \frac{B : s}{u} \qquad \frac{C : t}{u}$$

Each row in Figure 2.1 describes the rules for one particular family of propositions. The rules on the left are called **introduction rules**, and the rules on the

right are called **elimination rules**. The introduction rule for a logical operation O tells us how we can directly prove propositions obtained with O, and the elimination rule tells us how we can make use of a proof of a proposition obtained with O. For most families of propositions there is exactly one introduction and exactly one elimination rule. The exceptions are falsity (no introduction rule) and disjunctions (two introduction rules). Coq realizes the rules in Figure 2.1 with the following tactics.

	introduction	elimination
$\rightarrow$	intros	apply, exact
$\forall$	intros	apply, exact
$\bot$		contradiction, exfalso
$\wedge$	split	destruct
$\vee$	left, right	destruct
$\exists$	exists	destruct

There are no proof rules for negation and equivalence since these logical connectives are defined on top of the basic logical connectives.

$$\neg s := s \rightarrow \bot$$
$$s \leftrightarrow t := (s \rightarrow t) \wedge (t \rightarrow s)$$

The proof rules in Figure 2.1 were first formulated and studied by Gerhard Gentzen in 1935. They are known as *intuitionistic natural deduction rules*.

Exercise 2.12.1 Above we describe the elimination rule for disjunction in detail and relate it to a Coq tactic. Make sure that you can discuss each rule in Figure 2.1 in this fashion.

2.13 Proof Rules as Lemmas

Coq can express proof rules as lemmas. Here are the lemmas for the introduction and the elimination rule for conjunctions.

```
Lemma AndI (X Y : Prop) :
X → Y → X ∧ Y.
Proof. tauto. Qed.
Lemma AndE (X Y U : Prop) :
X ∧ Y → (X → Y → U) → U.
Proof. tauto. Qed.
```

To apply the proof rules, we can now apply the lemmas.

```
Goal ∀ X Y : Prop, X ∧ Y → Y ∧ X.
```

```
Proof.
  intros X Y A. apply (AndE A).
  intros x y. apply AndI.
  - exact y.
  - exact x.
Qed.
```

If you look at the applications of the lemmas in the proof above, it becomes clear that in Coq the name of a lemma is actually the name of the proof of the lemma. Since the statement of a lemma is typically universally quantified, the proof of a lemma is typically a proof generating function. Thus lemmas can be applied as you see it in the above proof scripts. When we represent a proof rule as a lemma, the proposition of the lemma formulates the rule as we see it, and the proof of the lemma is a function constructing a proof of the conclusion of the rule from the proofs required by the premises of the rule.

Next we represent the proof rules for existential quantifications as lemmas. Given a proposition $\exists x : s.t$, we face a bound variable x that may occur in the term t. To preserve the binding, we represent the proposition t as the predicate $\lambda x : s.t$.

```
Lemma ExI (X : Type) (p : X → Prop) :
forall x : X, p x → ∃ x, p x.
Proof. intros x A. ∃ x. exact A. Qed.
Lemma ExE (X : Type) (p : X → Prop) (U : Prop) :
(∃ x, p x) → (∀ x, p x → U) → U.
Proof. intros [x A] B. exact (B x A). Qed.
```

We can now prove propositions involving existential quantifications without using the tactics *exists* and *destruct*.

```
Goal ∀ (X : Type) (p q : X → Prop),
(∃ x, p x ∧ q x) → ∃ x, p x.
Proof.
  intros X p q A. apply (ExE A).
  intros x B. apply (AndE B). intros C _.
  exact (ExI C).
Qed.
```

Exercise 2.13.1 Formulate the introduction and elimination rules for disjunctions as lemmas and use the lemmas to prove the commutativity of disjunction.

2.14 Inductive Propositions

Recall that Coq provides for the definition of inductive types. So far we have used this facility to populate the universe *Type* with types providing booleans, natural

numbers, lists, and a few other families of values. It is also possible to populate the universe *Prop* with inductive types. We will speak of **inductive propositions** following the convention that types in *Prop* are called propositions. Here are the definitions of two inductive propositions from Coq's standard library.[7]

```
Inductive True : Prop :=
| I : True.

Inductive False : Prop := .
```

Recall that the proofs of a proposition A are the members of the type A. Thus the proposition *True* has exactly one proof (i.e., the **proof constructor** *I*), and the proposition *False* has no proof (since we defined *False* with no proof constructor).

By case analysis over the constructors of *True* we can prove that *True* has exactly one proof.

```
Goal ∀ x y : True, x=y.
Proof. intros x y. destruct x. destruct y. reflexivity. Qed.
```

By case analysis over the constructors of *False* we can prove that from a proof of *False* we can obtain a proof of every proposition.

```
Goal ∀ X : Prop, False → X.
Proof. intros X A. destruct A. Qed.
```

The case analysis over the proofs of *False* immediately succeeds since *False* has no constructor. We have discussed this form of reasoning in Section 1.13 where we considered the type *void*.

Coq defines conjunction and disjunction as inductive predicates (i.e., inductive type constructors into *Prop*).[8]

```
Inductive and (X Y : Prop) : Prop :=
| conj : X → Y → and X Y.

Inductive or (X Y : Prop) : Prop :=
| or_introl : X → or X Y
| or_intror : Y → or X Y.
```

Note that the inductive definitions of conjunction and disjunction follow exactly the BHK interpretation: A proof of $X \wedge Y$ consists of a proof of X and a proof of Y, and a proof of $X \vee Y$ consists of either a proof of X or a proof of Y. Also note that the definition of conjunction mirrors the definition of the product operator *prod* in Section 1.9.

Coq defines existential quantification as an inductive predicate that takes a type and a predicate as arguments:

[7] Use the command *Print* to look up the definitions

[8] Use the commands *Set Printing All* and *Print* to find out the definitions of the infix notations "∧" and "∨".

```
Inductive ex (X : Type) (p : X → Prop) : Prop :=
| ex_intro : ∀ x : X, p x → ex p.
```

With this definition an existential quantification $\exists x : s.t$ is represented as the application $ex\,(\lambda x : s.t)$. This way the binding of the local variable x is delegated to the predicate $\lambda x : s.t$. We have used this technique before to formulate the introduction and elimination rules for existential quantifications as lemmas (see Section 2.13).

Negation and equivalence are defined with plain definitions in Coq's standard library:

```
Definition not (X : Prop) : Prop := X → False.
Definition iff (X Y : Prop) : Prop := (X → Y) ∧ (Y → X).
```

Exercise 2.14.1 Prove the commutativity of disjunction without using the tactics *left* and *right*.

Exercise 2.14.2 Define your own versions of the logical operations and prove that they agree with Coq's predefined operations. Choose names different from Coq's predefined names to avoid conflicts.

Exercise 2.14.3 One can characterize negation with the following introduction and elimination rules not using falsity.

$$\frac{x : \mathit{Prop},\ s \Rightarrow x}{\neg s} \qquad\qquad \frac{\neg s \qquad s}{u}$$

The introduction rule requires a proof of an arbitrary proposition x under the assumption that a proof of s is given.

a) Formulate the rules as lemmas and prove the lemmas.

b) Give an inductive definition of negation based on the introduction rule.

c) Prove the elimination lemma for your inductive definition of negation.

2.15 An Observation

Look at the introduction rules for conjunction, disjunction, and existential quantification. If we formulate these rules as lemmas, we get exactly the types of the proof constructors of the inductive definitions of the respective logical operations.

Given the inductive definition of a logical operation, we can prove the elimination lemma for the operation. Since the inductive definition is only based

on the introduction rule of the operation, we can see the elimination rule as a consequence of the introduction rule.

We can also go from the elimination rules to the introduction rules. Look at the impredicative characterization of the logical operations in terms of implication and universal quantification appearing in Exercises 2.9.5 and 2.11.3. These characterizations reformulate the elimination rules of the logical operations. If we define a logical operation based on its impredicative characterization, we can prove the corresponding introduction and elimination lemmas. For conjunction we get the following development.

```
Definition AND (X Y : Prop) : Prop :=
forall Z : Prop, (X → Y → Z) → Z.

Lemma ANDI (X Y : Prop) :
X → Y → AND X Y.

Proof. intros x y Z. auto. Qed.

Lemma ANDE (X Y Z: Prop) :
AND X Y → (X → Y → Z) → Z.

Proof. intros A. exact (A Z). Qed.

Lemma AND_agree (X Y : Prop) :
AND X Y ↔ X ∧ Y.

Proof.
  split.
  - intros A. apply A. auto.
  - intros [x y] Z A. apply A ; assumption.
Qed.
```

Exercise 2.15.1 Define disjunction with a plain definition based on the impredicative characterization in Exercise 2.9.5. Prove an introduction, an elimination, and an agreement lemma for your disjunction. Carry out the same program for the existential quantifier.

2.16 Excluded Middle

In Mathematics, one assumes that every proposition is either false or true. Consequently, if X is a proposition, the proposition $X \vee \neg X$ must be true. The assumption that $X \vee \neg X$ is true for every proposition X is known as *principle of excluded middle*, XM for short. Here is a definition of XM in Coq.

```
Definition XM : Prop := ∀ X : Prop, X ∨ ¬ X.
```

Coq can neither prove *XM* nor ¬*XM*. This means that we can consistently assume *XM* in Coq. The philosophy here is that *XM* is a basic mathematical assumption but not a basic proof rule. By not building in *XM*, we can make explicit which proofs rely on *XM*. Logical systems that build in *XM* are called **classical**, and systems not building in *XM* are called **constructive** or **intuitionistic**.

Exercise 2.16.1 Prove the following goals. They state consequences of the De Morgan laws for conjunction and universal quantification whose proofs require the use of excluded middle.

```
Goal ∀ X Y : Prop,
XM → ¬ (X ∧ Y) → ¬ X ∨ ¬ Y.

Goal ∀ (X : Type) (p : X → Prop),
XM → ¬ (∀ x, p x) → ∃ x, ¬ p x.
```

Exercise 2.16.2 Prove that the following propositions are equivalent. There are short proofs if you use *tauto*.

```
Definition XM : Prop := ∀ X : Prop, X ∨ ¬ X.              (* excluded middle *)
Definition DN : Prop := ∀ X : Prop, ¬¬ X → X.             (* double negation *)
Definition CP : Prop := ∀ X Y : Prop, (¬ Y → ¬ X) → X → Y.    (* contraposition *)
Definition Peirce : Prop := ∀ X Y : Prop, ((X → Y) → X) → X.  (* Peirce's Law *)
```

Exercise 2.16.3 (Drinker's Paradox) Consider a bar populated by at least one person. Using excluded middle, one can prove that one can pick some person in the bar such that everyone in the bar drinks Whiskey if this person drinks Whiskey. Do the proof in Coq.

```
Lemma drinker (X : Type) (d : X → Prop) :
XM → (∃ x : X, True) → ∃ x, d x → ∀ x, d x.
```

Exercise 2.16.4 (Glivenko's Theorem) A proposition is **pure** if it is either a variable, falsity, or an implication, negation, conjunction, or disjunction of pure propositions. Valery Glivenko showed in 1929 that a pure proposition is provable classically if and only if its double negation is provable intuitionistically. That is, if *s* is a pure proposition, then *XM* → *s* is provable in Coq if and only if ¬¬*s* is provable in Coq. This tells us that *tauto* can prove the following goals.

```
Goal ∀ X : Prop,
¬¬ (X ∨ ¬ X).

Goal ∀ X Y : Prop,
¬¬ (((X → Y) → X) → X).

Goal ∀ X Y : Prop,
¬¬ (¬ (X ∧ Y) ↔ ¬ X ∨ ¬ Y).
```

Goal ∀ X Y : Prop,
¬¬ ((X → Y) ↔ (¬ Y → ¬ X)).

Do the proofs without using *tauto* and try to find out why the outer double negation can replace excluded middle.

Exercise 2.16.5 A proposition s is **propositionally decidable** if the proposition $s \vee \neg s$ is provable. Prove that the following propositions are propositionally decidable.

a) ∀ X : Prop, ¬ (X ∨ ¬ X)

b) ∃ X : Prop, ¬ (X ∨ ¬ X)

c) ∀ P : Prop, ∃ f : Prop → Prop, ∀ X Y : Prop,
(X ∧ P → Y) ↔ (X → f Y)

d) ∀ P : Prop, ∃ f : Prop → Prop, ∀ X Y : Prop,
(X → Y ∧ P) ↔ (f X → Y)

2.17 Discussion and Remarks

Our treatment of propositions and proofs is based on the constructive approach, which sees proofs as first-class objects and defines the meaning of propositions by relating them to their proofs. In contrast to the classical approach, no notion of truth value is needed. Our starting point is the BHK interpretation, which identifies the proofs of implications and quantifications as functions. The BHK interpretation is refined by the propositions as types principle, which models implications and universal quantification as function types such that the proofs of a proposition appear as the members of the type representing the proposition. As it turns out, universal quantification alone suffices to express all logical operations (impredicative characterizations).

The ideas of the constructive approach developed around 1930 and led to the BHK interpretation (Brouwer, Heyting, Kolmogorov). A complementary achievement is the system of natural deduction (i.e., basic proof rules) formulated in 1935 by Gerhard Gentzen. While the BHK interpretation starts with proofs as first-class objects, Gentzen's approach takes the proof rules as starting point and sees proofs as derivations obtained with the rules. Given the BHK interpretation, the correctness of the proof rules can be argued. Given the proof rules, the correctness of the BHK interpretation can be argued.

A formal model providing functions as assumed by the BHK interpretation was developed in the 1930's by Alonzo Church under the name lambda calculus. The notion of types was first formulated by Bertrand Russell around 1900. A typed lambda calculus was published by Alonzo Church in 1940. Typed lambda

calculus later developed into constructive type theory, which became the foundation for Coq.

The correspondence between propositions and types was recognized by Curry and Howard for pure propositional logic and first reported about in a paper from 1969. The challenge then was to formulate a type theory strong enough to model quantifications as propositions. For such a type theory dependent function types are needed. Dependently typed type theories were developed by Nicolaas de Bruijn, Per Martin-Löf, and Jean-Yves Girard around 1970. Coq's type theory originated in 1985 (Coquand and Huet) and has been refined repeatedly.

2.18 Tactics Summary

intros $x_1 \ldots x_n$	introduces implications and universal quantifications
apply t	reduces claim by backward application of proof function t
exact t	Solves goal with proof t
contradiction t	Soves goal by explosion if t is proof of False
exfalso	Changes claim to False (explosion)
split	splits conjunctive claim
left	reduces disjunctive claim to left constituent
right	reduces disjunctive claim to right constituent
exists t	instantiates existential claim with witness t
specialize (x t)	instantiates assumption x with t
assumption	solves goals whose claim appears as assumption
auto	tries to solve goal with *intros*, *apply*, *assumption*, *reflexivity*, ...
tauto	solves goals solvable by pure propositional reasoning

3 Conversion and Equality

In this chapter we study equality in Coq. Equality in Coq rests on conversion, an equivalence relation on terms coming with the type theory underlying Coq. There is the basic assumption that convertible terms represent the same object. Moreover, evaluation steps respects conversion in that they rewrite terms to convertible terms.

We will see many basic proofs involving equality. For instance, we will prove that the number 1 is different from 2, and that constructors like S or *cons* are injective. We will also prove that the type *nat* is different from the type *bool*. We will study these proofs at the level of the underlying type theory.

A second topic of this chapter are inductive proofs. Here we learn that inductive proofs are obtained as recursive functions in the underlying type theory.

3.1 Conversion Principle

The type theory underlying Coq comes with an equivalence relation on terms called **convertibility**. The type theory assumes that convertible terms have the same meaning. This assumption is expressed in the **conversion principle**, which says that convertible types have the same elements. Applied to propositions, the conversion principle says that a proof s of a proposition t is also a proof of every proposition t' that is convertible with t. Thus if we search for a proof of a proposition t, we can switch to a convertible proposition t' and search for a proof of t'.

The convertibility relation is defined as the least equivalence relation on terms that is compatible with the term structure and certain conversion rules. Conversion rules can be applied in both directions (i.e., from left to right and from right to left). For the terms introduced so far we have the following conversion rules.

- **Alpha conversion**. Consistent renaming of local variables. For instance, $\lambda x:s.x$ and $\lambda y:s.y$ are alpha convertible.
- **Beta conversion**. The terms $(\lambda x:s.t)u$ and t_u^x are beta convertible. Beta conversion is the undirected version of beta reduction. The direction from t_u^x to $(\lambda x:s.t)u$ is called **beta expansion**. Terms of the form $(\lambda x:s.t)u$ are called **beta redexes**.

- **Eta conversion**. The terms $\lambda x : s.tx$ and t are eta convertible if x does not occur in t and both terms have the same type. The direction from $\lambda x : s.tx$ to t is called **eta reduction**, and the reverse direction is called **eta expansion**. Eta reduction eliminates unnecessary lambda abstractions.
- **Delta conversion.** A defined name x and the term t it is bound to are convertible. The direction from the name to the term is called **unfolding**, the other direction is called **folding**.[1]
- **Match conversion**. The undirected version of match reduction.
- **Fix conversion**. The undirected version of fix reduction.

Since the computation rules are directed versions of the conversion rules for lambda abstractions (beta), matches, fixes, and defined names (delta), every evaluation step is a conversion step. Thus a term is always convertible to its normal form.

Coq comes with various **conversion tactics** making it possible to convert the claim and the assumptions of proof goals. Such conversions are logically justified by the conversion principle. We will see the conversion tactics *change*, *pattern*, *hnf*, *cbv*, *simpl*, *unfold*, and *fold*. The following examples do not prove interesting lemmas but illustrate the conversion rules and the conversion tactics.

```
Goal ¬¬True.
```

Proof.	$\neg\neg True$
change (¬True → False).	$\neg True \to False$
change (¬(True → False)).	$\neg(True \to False)$
change (¬¬True).	$\neg\neg True$
hnf.	$\neg True \to False$
change (¬¬True).	$\neg\neg True$
cbv.	$(True \to False) \to False$
change (¬¬True).	$\neg\neg True$
simpl.	$\neg\neg True$
pattern True.	$(\lambda p : Prop.\neg\neg p)\ True$
pattern not at 2.	$(\lambda f : Prop \to Prop.(\lambda p : Prop.\neg f\ p)\ True)\ not$
hnf.	$\neg True \to False$
exact (fun f ⇒ f I).	
Show Proof.	
Qed.	

The tactic *change t* changes the current claim to t provided the current claim and t are convertible. The tactic *change* gives us a means to check with Coq whether two terms are convertible. The tactic *hnf* (head normal form) applies computation rules to the top of a term until the top of the term cannot be reduced further. The tactic *cbv* (call by value) fully evaluates a term (similar to the

[1] The names of lemmas established with *Qed* cannot be unfolded.

command *Compute*). The tactic *pattern t* abstracts out a subterm t of a claim by converting the claim to a beta redex $(\lambda x{:}s.u)t$ that reduces to the claim by a beta reduction step. Note that *pattern* performs a beta expansion. The second use of *pattern* in the above script abstracts out only the second occurrence of the subterm *not*.

Note that all terms shown at the right of the above proof are convertible propositions. By the conversion principle we know that all of these propositions have the same proofs.

The above script also contains an occurrence of the tactic *simpl* so that we can compare it with the tactics *hnf* and *cbv*. Note that the occurrence of *simpl* has no effect in the above script. In fact, *simpl* will change a term only if the conversion involves a match reduction. If you study the examples in Chapter 1, you will learn that *simpl* applies computation rules but also performs folding steps for recursive definitions (backward application of definition unfolding).

Note the command *Show Proof* at the end of the script. It shows the proof term the script will have constructed at this point. The conversion tactics do not show in the proof term, except for the fact that the missing types in the description of the proof term appearing as argument of *exact* will be derived based on the goal visible at this point.

All conversion tactics can be applied to assumptions. For instance, the command "*simpl in A*" will simplify the assumption A.

For the following conversion examples we define an inductive predicate *demo*.

```
Inductive demo (X : Type) (x : X) : Prop :=
| demoI : demo x.
```

First we demo delta conversion with the tactics *unfold* and *fold*.

```
Goal demo plus.
```

```
Proof.                  demo plus
  unfold plus.          demo (fix plus (x y : nat) : nat := match x with ··· end)
  unfold plus.          demo (fix plus (x y : nat) : nat := match x with ··· end)
  fold plus.            demo plus
  apply demoI.
Qed.
```

Note that the second occurrence of the unfold tactic has no effect since the claim does not contain a defined name *plus*.

Next we demo alpha conversion.

```
Goal demo (fun x : nat ⇒ x).

Proof.
  change (demo (fun y : nat ⇒ y)).
  change (demo (fun myname : nat ⇒ myname)).
```

```
  apply demoI.
Qed.
```

For the remaining demos we use Coq's section facility to conveniently declare variables.[2]

```
Section Demo.
  Variable n : nat.
```

Here is a conversion demo involving match and fix conversions.

```
  Goal demo (5+n+n).
```

`Proof.`	$demo\ (5+n+n)$
`  change (demo (2+3+n+n)).`	$demo\ (2+3+n+n)$
`  simpl.`	$demo\ (S\,(S\,(S\,(S\,(S\,(n+n))))))$
`  change (demo (10+n-5+n)).`	$demo\ (10+n-5+n)$
`  pattern n at 1.`	$(\lambda x{:}nat.\ demo\ (10+x-5+n))\ n$
`  hnf.`	$demo\ (10+n-5+n)$
`  simpl.`	$demo\ (S\,(S\,(S\,(S\,(S\,(n+n))))))$
`  apply demoI.`	
`Qed.`	

Finally, we demonstrate eta conversion.

```
  Variable X : Type.
  Variable f : X → X → X.
  Goal demo f.
```

`Proof.`	$demo\ f$
`  change (demo (fun x ⇒ f x)).`	$demo\ (\lambda x{:}X.\ f\,x)$
`  cbv.`	$demo\ (\lambda x{:}X.\ f\,x)$
`  change (demo (fun x y ⇒ f x y)).`	$demo\ (\lambda x{:}X.\ \lambda y{:}X.\ f\,x\,y)$
`  cbv.`	$demo\ (\lambda x{:}X.\ \lambda y{:}X.\ f\,x\,y)$
`  apply demoI.`	
`Qed.`	

```
End Demo.
```

You may wonder why Coq does not employ eta reduction as computation rule. The reason is that naive eta reduction is not always type preserving. For instance, the term

$$\lambda x{:}Prop.\ (\lambda y{:}Type.\ y)\ x$$

has type $Prop \to Type$. The application of the inner lambda abstraction to x type checks since every proposition is a type. A naive eta reduction would yield the term $\lambda y{:}Type.\ y$, which has type $Type \to Type$. This violates type preservation since the types $Prop \to Type$ and $Type \to Type$ are incomparable in Coq.

[2] This is the first time we use Coq's section facility.

Exercise 3.1.1 The tactic *reflexivity* can prove an equation $s = t$ if and only if the terms s and t are convertible. Argue for each of the following goals whether or not it can be shown by reflexivity and check your answer with Coq.

(a) **Goal** plus 1 = S.
(b) **Goal** (fun y ⇒ 3+y) = plus (4−1).
(c) **Goal** S = fun x ⇒ x + 1.
(d) **Goal** S = fun x ⇒ 1 + x.
(e) **Goal** S = fun x ⇒ 2+x+1−2.
(f) **Goal** plus 3 = fun x ⇒ 5+x−2.
(g) **Goal** mult 2 = fun x ⇒ x + (x + 0).
(h) **Goal** S = fun x ⇒ S (pred (S x)).
(i) **Goal** minus = fun x y ⇒ x−y.

3.2 Disjointness and Injectivity of Constructors

Different constructors of an inductive type always yield different values. We start by proving that the constructors *true* and *false* of *bool* are different.

```
Goal false ≠ true.
Proof.
  intros A.
  change (if false then True else False).
  rewrite A.
  exact I.
Qed.
```

The proof follows a simple path. We first introduce the equation *false* = *true*. Then we convert the resulting claim into a conditional with the condition *false*. Using the assumed equation *false* = *true*, we rewrite the condition of the conditional to *true*. By conversion we obtain the claim *True* and finish the proof. What makes the proof go through is the conversion rule for matches and the conversion principle.

The idea of the proof of *false* ≠ *true* carries over to *nat*. We prove that the constructors *O* and *S* yield different values.

```
Lemma disjoint_O_S n :
  0 ≠ S n.
Proof.
  intros A.
  change (match 0 with 0 ⇒ False | _ ⇒ True end).
  rewrite A.
  exact I.
Qed.
```

With a similar idea we can prove that the constructor *S* is injective.

```
Lemma injective_S x y :
  S x = S y → x = y.
Proof.
  intros A.
  change (pred (S x) = pred (S y)).
  rewrite A.
  reflexivity.
Qed.
```

Coq's tactics *discriminate*, *injection*, and *congruence* can do this sort of proofs automatically (that is, construct suitable proof terms).

```
Goal ∀ x, S x ≠ 0.
Proof. intros x A. discriminate A. Qed.
Goal ∀ x y, S x = S y → x = y.
Proof. intros x y A. injection A. auto. Qed.
```

The tactic *congruence* can prove both of the above goals in one go.

Exercise 3.2.1 Give three proofs for each of the following goals: with *congruence*, with *discriminate*, and with *change*.

(a)
```
Goal ∀ (X : Type) (x : X),
       Some x ≠ None.
```

(b)
```
Goal ∀ (X : Type) (x : X) (A : list X),
       x::A ≠ nil.
```

Exercise 3.2.2 Give three proofs for each of the following goals: with *congruence*, with *injection*, and with *change*.

(a)
```
Goal ∀ (X Y: Type) (x x' : X)  (y y' : Y),
       (x,y) = (x',y') → x=x' ∧ y = y'.
```

(b)
```
Goal ∀ (X : Type) (x x' : X) (A A' : list X),
       x::A = x'::A' → x=x' ∧ A = A'.
```

Exercise 3.2.3 Prove the following goals.

(a) `Goal ∀ x, negb x ≠ x.`

(b) `Goal ∀ x, S x ≠ x.`

(c) `Goal ∀ x y z, x + y = x + z → y = z.`

(d) `Goal ∀ x y : nat, x = y ∨ x ≠ y.`

Hint: Recall that you can simplify an assumption *A* with the command *simpl in A*.

Exercise 3.2.4 Prove the following goal.

```
Goal ∃ (X : Type) (f : list X → X), ∀ A B, f A = f B → A = B.
```

Before you prove the goal, you may define an inductive type.

Exercise 3.2.5 Prove $False \neq True$.

Exercise 3.2.6 A term $\lambda x : s.\, t\,x$ can only be eta reduced if x does not occur in t. If this restriction was removed, we could obtain a proof of *False*. Show this by proving the following goal.

```
Goal ∃ (f : nat → nat → nat) x,
       (fun x ⇒ f x x) ≠ f x.
```

3.3 Leibniz Equality

There is a straightforward characterization of equality that can be expressed in every logical system that can quantify over predicates. The characterization is due to the philosopher and mathematician Gottfried Wilhelm Leibniz and says that two objects x and y are equal if they have the same properties. Formally, Leibniz' characterization can be expressed with the equivalence

$$x = y \;\leftrightarrow\; \forall p : X \to Prop.\; p\,x \leftrightarrow p\,y$$

We can use the equivalence to define equality. If equality is obtained in some other way, we still expect it to satisfy the equivalence. This means that equality is determined up to logical equivalence in any logical system that can quantify over predicates. The Leibniz characterization of equality suffices to justify the tactics *reflexivity* and *rewrite*.

1. Assume that s and t are convertible terms such that the equation $s = t$ is well typed. We prove the proposition $s = t$. First we observe that the propositions $s = t$ and $s = s$ are convertible since s and t are convertible (recall that propositions are terms). Thus we know by the conversion principle that $s = t$ is provable if $s = s$ is provable. By the Leibniz characterization of equality we know that $s = s$ is provable if $\forall p : X \to Prop.\; p\,s \leftrightarrow p\,s$ is provable, which is the case. So we have a proof of $s = t$ and a justification of the tactic *reflexivity*.
2. Assume we have a proof of an equation $s = t$ and two propositions $u\,s$ and $u\,t$. Then we know by the Leibniz characterization of $s = t$ that $u\,s$ is provable if and only if $u\,t$ is provable. So if we have a claim or an assumption $u\,s$, we can rewrite it to $u\,t$. This justifies the rewriting tactic for the case where s and t appear as the right constituent of a top level application. Since we have beta

conversion, the restriction to top level applications is not significant. Given a term v containing a subterm s, beta expansion will give us a term u such that the terms v and us are convertible. Taken together, we have arrived at a justification of the tactic *rewrite*.

We now define an equality predicate we call **Leibniz equality**.

```
Definition leibniz_eq (X : Type) (x y : X) : Prop :=
  ∀ p : X → Prop, p x → p y.
```

The definition deviates from Leibniz' characterization in that it uses an implication rather than an equivalence. As it turns out, the asymmetric version we use is logically equivalent to the symmetric version with the equivalence. We have chosen the asymmetric version since it is simpler than the symmetric version. We can read the asymmetric version as follows: A proof of $x = y$ is a function that for every predicate p maps a proof of px to a proof of py.

We define a convenient notation for Leibniz equality and prove that it is reflexive and symmetric.

```
Notation "x == y" := (leibniz_eq x y) (at level 70, no associativity).

Lemma leibniz_refl X (x : X) :
  x == x.

Proof. hnf. auto. Qed.

Lemma leibniz_sym X (x y : X) :
  x == y → y == x.

Proof.
  unfold leibniz_eq. intros A p.
  apply (A (fun z ⇒ p z → p x)).
  auto.
Qed.
```

Next we show that Leibniz equality agrees with Coq's predefined equality.

```
Lemma leibniz_agrees X (x y : X) :
  x == y ↔ x = y.

Proof.
  split ; intros A.
  – apply (A (fun z ⇒ x=z)). reflexivity.
  – rewrite A. apply leibniz_refl.
Qed.
```

Since we can turn Leibniz equations into Coq equations, we can rewrite with Leibniz equations. However, we can also rewrite without going through Coq's predefined equality. All we need is the following lemma.

```
Lemma leibniz_rewrite X (x y : X) (p : X → Prop) :
  x == y → p y → p x.
```

```
Proof. intros A. apply (leibniz_sym A). Qed.
```

We now prove that addition is associative with respect to Leibniz equality without using anything connected with Coq's predefined equality.

```
Lemma leibniz_plus_assoc x y z :
  (x + y) + z == x + (y + z).
Proof.
  induction x ; simpl.
  – apply leibniz_refl.
  – pattern (x+y+z). apply (leibniz_rewrite IHx). apply leibniz_refl.
Qed.
```

The proof deserves careful study. One interesting point is the use of *pattern* to abstract out the term we want to rewrite. With *pattern* we can convert a term s containing a subterm u to a beta redex $(\lambda x.t)u$ such that $\lambda x.t$ is the predicate p we need to rewrite with a Leibniz equation $u == v$. So beta conversion makes it possible to reduce general rewriting to top level rewriting $pu \rightsquigarrow pv$. A proof of the proposition $\forall p.\, pv \to pu$ is a function that makes it possible to rewrite a claim with the equation $u = v$.

Coq's library defines equality as an inductive predicate. This is in harmony with the definitions of the logical connectives and of existential quantification. We will discuss Coq's inductive definition of equality in a later chapter on inductive predicates.

Exercise 3.3.1 Prove that addition is commutative for Leibniz equality without using Coq's predefined equality. You will need two lemmas.

Exercise 3.3.2 Prove the following rewrite lemmas for Leibniz equality without using other lemmas.

(a)
```
Lemma leibniz_rewrite_lr X (x y : X) (p : X → Prop) :
    x == y → p y → p x.
```

(b)
```
Lemma leibniz_rewrite_rl X (x y : X) (p : X → Prop) :
    x == y → p x → p y.
```

Exercise 3.3.3 Suppose we want to rewrite a subterm u in a proposition t using the lemma *leibniz_rewrite*. Then we need a predicate $\lambda x.s$ such that t and $(\lambda x.s)u$ are convertible and s is obtained from t by replacing the occurrence of u we want to rewrite with the variable x. Let t be the proposition $x + y + x = y$.

a) Give a predicate for rewriting the first occurrence of x in t.

b) Give a predicate for rewriting the second occurrence of y in t.

c) Give a predicate for rewriting all occurrences of y in t.

d) Give a predicate for rewriting the term $x + y$ in t.

e) Explain why the term $y + x$ cannot be rewritten in t.

3.4 By Name Specification of Implicit Arguments

We take the opportunity to discuss an engineering detail of Coq's term language. In implicit arguments mode, Coq derives for some constants (i.e., defined names) an **expanded type** providing for additional implicit arguments. The real type and the expanded type are always convertible, so the difference does not matter for type checking. We can use the command *About* to find out whether Coq has determined an expanded type for a constant. For instance, this is the case for the constant *leibniz_sym* defined in the previous section.

```
About leibniz_sym.
```

```
  leibniz_sym : ∀ (X : Type) (x y : X), x == y → y == x
  Expanded type for implicit arguments
  leibniz_sym : ∀ (X : Type) (x y : X), x == y → ∀ p : X → Prop, p y → p x
  Arguments X, x, y, p are implicit
```

If you print the lemma *leibniz_rewrite* from the previous section, you will see the following proof term:

```
fun (X : Type) (x y : X) (p : X → Prop) (A : x == y) =>
  leibniz_sym (x:=x) (y:=y) A (p:=p)
```

Note that the implicit arguments x, y, and p of *leibniz_sym* are explicitly specified by name. By-name specification of implicit and explicit arguments can also be used when you give terms to Coq. Step through the following script to understand the many notational possibilities Coq has in offer.

```
Goal ∀ X (x y : X) (p : X → Prop),
       x == y → p y → p x.
Proof.
  intros X x y p A.
  Check leibniz_sym A.
  Check leibniz_sym A (p:=p).
  Check @leibniz_sym X x y A p.
  Check @leibniz_sym _ _ _ A p.
  exact (leibniz_sym A (p:=p)).
  Show Proof.
 Qed.
```

3.5 Local Definitions

Coq's term language has a construct for local definitions taking the form

let $x : t := s$ in u

where x is the local name, t is the type declared for x, s is value of x, and u is the term in which the local definition is visible. Coq will check that the term s has the declared type t. In case the declared type is omitted, Coq will try to infer it. Local definitions come with a reduction rule called **zeta reduction** that replaces the defined name with its value:

$$\text{let } x : t := s \text{ in } u \quad \rightsquigarrow \quad u^x_s$$

Here are examples.

```
Compute let x := 2 in x + x.
% 4 : nat

Compute let x := 2 in let x := x + x in x.
% 4 : nat

Compute let f := plus 3 in f 7.
% 10 : nat
```

The undirected version of zeta reduction serves as a conversion rule (**zeta conversion**). Note that zeta reduction looks very much like beta reduction. There is however an important difference between a local definition *let* $x : t := s$ *in* u and the corresponding beta redex $(\lambda x : t.\ u)\ s$: The continuation u of a local definition is type checked with delta conversion enabled between the local name x and the defining term s. Thus the local definition

```
Check let X := nat in (fun x : X ⇒ x) 2.
```

will type check while the corresponding beta redex will not.

```
Check (fun X ⇒ (fun x : X ⇒ x) 2) nat.
% Error : The term 2 is expected to have type X.
```

Besides for local definitions, Coq uses the let notation also as a syntactic convenience for one-constructor matches. For instance:

```
let (x,y) := (2,7) in x + y  ⇝  match (2,7) with pair x y ⇒ x + y end
```

3.6 Proof of nat ≠ bool

We will now prove that the types *bool* and *nat* are different. The proof will employ a predicate p on types that holds for *bool* but does not hold for *nat*. For p we choose the property that a type has at most two elements. The proof script uses two important tactics we have not seen before.

```
Goal bool ≠ nat.
```

```
Proof.
  pose (p X := ∀ x y z : X, x=y ∨ x=z ∨ y=z).
  assert (H: ¬p nat).
  { intros B. specialize (B 0 1 2). destruct B as [B|[B|B]] ; discriminate B. }
  intros A. apply H. rewrite ← A.
  intros [|] [|] [|] ; auto.
Qed.
```

The tactic *pose* defines the discriminating predicate p.[3] The tactic *assert* states the intermediate claim $\neg p\,nat$. For the proof of the intermediate claim Coq introduces a subgoal. The script proving the subgoal is enclosed in curly braces. The tactic *specialize* is used to instantiate the universally quantified assumption $p\,nat$ with the numbers 0, 1, and 2. With case analysis and *discriminate* we show that the instantiated assumption is contradictory. After the intermediate claim is established, we can use it as an additional assumption H. We now introduce the assumption $A : bool = nat$ and apply the intermediate claim H. The claim is now $p\,nat$. We rewrite with the assumption A and obtain the claim $p\,bool$. This claim follows by case analysis over the universally quantified boolean variables. As always, step carefully through the proof script to understand.

Exercise 3.6.1 Prove the following goals.

(a) `Goal bool ≠ option bool.`
(b) `Goal option bool ≠ prod bool bool.`
(c) `Goal bool ≠ False.`

Exercise 3.6.2 Step through the proof of $bool \neq nat$ and insert the command *Show Proof* immediately after the assert. You will see that the local definition of p is realized with a let and that the assumption H is realized with a beta redex.

```
let p := fun X : Type ⇒ forall x y z : X, x = y ∨ x = z ∨ y = z in
  (fun H : ¬ p nat ⇒ ?2) ?1
```

The two **existential variables** ?2 and ?1 represent the claims of the two subgoals that have to be solved at this point (1 represents the claim of the subgoal for the assert and ?2 represents the claim of the remaining subgoal).

3.7 Cantor's Theorem

Cantor's theorem says that there is no surjective function from a set to its power set. This means that the power set of a set X is strictly larger than X. For his proof Cantor used a technique commonly called *diagonalisation*. It turns out

[3] The tactic *pose* constructs a proof term with a let expression accommodating the local definition.

that Cantor's proof carries over to type theory. Here we can show that there is no surjective function from a Type X to the type $X \to Prop$. Speaking informally, this means that there are strictly more predicates on X than there are elements of X.

```
Definition surjective (X Y : Type) (f : X → Y) : Prop :=  ∀ y, ∃ x, f x = y.

Lemma Cantor X :
  ¬ ∃ f : X → X → Prop, surjective f.

Proof.
  intros [f  A].
  pose (g x := ¬ f x x).
  specialize (A g).
  destruct A as [x A].
  assert (H: ¬ (g x ↔ ¬ g x)) by tauto.
  apply H. unfold g at 1. rewrite A. tauto.
Qed.
```

The proof assumes a type X and a surjective function f from X to $X \to Prop$ and constructs a proof of *False*. We first define a *spoiler function* $gx := \neg fxx$ in $X \to Prop$. Since f is surjective, there is an x such that $fx = g$. Thus $gx = \neg fxx = \neg gx$, which is contradictory.

Exercise 3.7.1 Prove the following goals.

(a) `Goal ¬ ∃ f : nat → nat → nat, surjective f.`

(b) `Goal ¬ ∃ f : bool → bool → bool, surjective f.`

Exercise 3.7.2 Prove the following generalization of Cantor's Theorem.

```
Lemma Cantor_generalized X Y :
  (∃ N : Y → Y, ∀ y, N y ≠ y) →
  ¬ ∃ f : X → X → Y,  surjective f.
```

Exercise 3.7.3 Prove the following variant of Cantor's Theorem.

```
Lemma Cantor_neq X Y (f : X → X → Y) (N : Y → Y) :
 (∀ y, N y ≠ y) → ∃ h, ∀ x, f x ≠ h.
```

Exercise 3.7.4 Prove the following goals. They establish sufficient conditions for the surjectivity and injectivity of functions based on inverse functions.

```
Definition injective (X Y : Type) (f : X → Y) : Prop :=  ∀ x x' : X, f x = f x' → x = x'.

Goal ∀ X Y : Type, ∀ f : X → Y, (∃ g : Y → X, ∀ y, f (g y) = y) → surjective f.

Goal ∀ X Y : Type, ∀ f : X → Y, (∃ g : Y → X, ∀ x, g (f x) = x) → injective f.
```

Exercise 3.7.5 One can also show that no type X admits an injective function f from $X \to \mathit{Prop}$ to X. Given X and f, the proof defines a predicate $p : X \to \mathit{Prop}$ such that both $\neg p(f\,p)$ and $p(f\,p)$ are provable. Given the definition of p, the proof is routine. Complete the following proof script.

```
Goal ∀ X, ¬ ∃ f : (X → Prop) → X, injective f.
Proof.
  intros X [f A].
  pose (p x := ∃ h, f h = x ∧ ¬ h x).
  . . .
Qed.
```

3.8 Kaminski's Equation

Kaminski's equation[4] takes the form $f(f(f\,x)) = f\,x$ and holds for every function $f : \mathit{bool} \to \mathit{bool}$ and every boolean x. The proof proceeds by repeated boolean case analysis: First on x and then on $f\,\mathit{true}$ and $f\,\mathit{false}$. For the proof to work, the boolean case analysis on $f\,\mathit{true}$ must provide the equations $f\,\mathit{true} = \mathit{true}$ and $f\,\mathit{true} = \mathit{false}$ coming with the case analysis. The equations are also needed for the case analysis on $f\,\mathit{false}$. We use the annotation *eqn* to tell the tactic *destruct* that we need the equations.

```
Goal ∀ (f : bool → bool) (x : bool),  f (f (f x)) = f x.
Proof. intros f x. destruct x, (f true) eqn:A, (f false) eqn:B ; congruence. Qed.
```

To understand, replace the semicolon before *congruence* with a period and solve the 8 subgoals by hand.

For boolean case analyses, the annotated use of *destruct* can be simulated with the following lemma.

```
Lemma destruct_eqn_bool (p : bool → Prop) (x : bool) :
  (x = true → p true) → (x = false → p false) → p x.
Proof. destruct x ; auto. Qed.
```

To apply the lemma, we use the tactic *pattern* to identify the predicate p.

```
Goal ∀ (f : bool → bool) (x : bool),  f (f (f x)) = f x.
Proof.
  destruct x ;
  pattern (f true) ; apply destruct_eqn_bool ;
  pattern (f false) ; apply destruct_eqn_bool ;
  congruence.
Qed.
```

[4] The equation was brought up as a proof challenge by Mark Kaminski in 2005 when he wrote his Bachelor's thesis on classical higher-order logic.

Replace the semicolons with periods and solve the subgoals by hand to understand.

Exercise 3.8.1 Prove the following variant of Kaminski's equation.

```
Goal ∀ (f g : bool → bool) (x : bool), f (f (f (g x))) = f (g (g (g x))).
```

3.9 Boolean Equality Tests

It is not difficult to write a boolean equality test for *nat*.

```
Fixpoint nat_eqb (x y : nat) : bool :=
  match x, y with
    | O, O ⇒ true
    | S x', S y' ⇒ nat_eqb x' y'
    | _, _ ⇒ false
  end.
```

We prove that the boolean equality test agrees with Coq's equality.

```
Lemma nat_eqb_agrees x y :
  nat_eqb x y = true ↔ x = y.
Proof.
  revert y.
  induction x ; intros [|y] ; split ; simpl ; intros A ; try congruence.
  − f_equal. apply IHx, A.
  − apply IHx. congruence.
Qed.
```

Note that the proof uses the **tactical** *try*. Try is needed since *congruence* can only solve 6 of the 8 subgoals produced by the induction on x, the case analysis on y, and the split of the equivalence. A command *try t* behaves like the tactic t if t succeeds but leaves the goal unchanged if t fails. Also note the command *apply IHx, A*. It first applies the inductive hypothesis from left to right and then applies the assumption A. So we learn that *apply* can apply equivalences in either direction and that succeeding applications can be condensed in one apply with commas. Without these conveniences, we may write *apply IHx, A* as

```
destruct (IHx y) as [C _]. apply C. apply A.
```

Exercise 3.9.1 Write a boolean equality test for *bool* and prove that it agrees with Coq's equality.

Exercise 3.9.2 Write a boolean equality test for *lists* and prove that it agrees with Coq's equality. The equality test for lists should take a boolean equality test for the element type of the lists as arguments. Prove the correctness of your equality test with the following lemma.

```
Lemma list_eqb_agrees X (X_eqb : X → X → bool) (A B : list X) :
  (∀ x y, X_eqb x y = true ↔ x = y) →
  (list_eqb X_eqb A B = true ↔ A = B).
```

Coq Summary

Conversion Tactics

change, *pattern*, *hnf*, *cbv*, *simpl*, *unfold*, *fold*.

Constructor Tactics

discriminate, *injection*, *congruence*.

Other Tactics

pose, *assert*.

Tacticals

try

New Features of the Tactics *apply* and *destruct*

- *apply* can apply equivalences in either direction. See Section 3.9, proof of *nat_eqb_agrees*.
- A sequence of applies can be written as a single apply using commas. For instance, we may write "*apply A*, *B*, *C*." for "*apply A*. *apply B*. *apply C*.". See Section 3.9, proof of *nat_eqb_agrees*.
- *destruct* can be used with an *eqn*-annotation to provide the equations governing the case analysis as assumptions. The *eqn*-annotation goes after the *as*-annotation.

New Features of the Term Language

- Implicit arguments can be specified by name rather than by position. See Section 3.4, application of *leibniz_sym*.
- Local definitions with the let notation. See Section 3.5.
- Let notation for one-constructor matches. See Section 3.5.

Sections

See Section 3.1.

4 Induction and Recursion

So far we have done all inductive proofs with the tactic *induction*. We will continue to do so, but it is time to explain how inductive proofs are obtained in Coq's type theory. Recall that tactics are not part of Coq's type theory, that propositions are represented as types, and that proofs are represented as terms describing elements of propositions. So there must be some way to represent inductive proofs as terms of the type theory. Since inductive proofs in Coq are always based on inductive types (e.g., *nat* or *list X*), the fact that Coq obtains structural induction as structural recursion should not come as a surprise.

4.1 Induction Lemmas

When we define an inductive type, Coq automatically establishes an induction lemma for this type. For *nat* the induction lemma has the following type.[1]

```
Check nat_ind.
```

$nat\_ind\ :\ \forall p : nat \to Prop,\ \ p\,0\ \to\ (\forall n : nat,\ p\,n \to p\,(Sn))\ \to\ \forall n : nat,\ p\,n$

The type tells us that *nat_ind* is a function that takes a predicate p and yields a proof of $\forall n : nat,\ p\,n$, provided it is given a proof of $p\,0$ and a function that for every n and every proof of $p\,n$ yields a proof of $p\,(S\,n)$. The second and the third argument of *nat_ind* represent what in mathematical speak is called the *basis step* and the *inductive step*.

Coq's tactic *induction* is applied to a variable of an inductive type and applies the induction lemma of this type. In the case of *nat_ind* this will produce two subgoals, one for the basis step and one for the inductive step. Here is a proof that obtains the necessary induction by applying *nat_ind* directly.

```
Goal ∀ n, n + 0 = n.

Proof.
  apply (nat_ind (fun n ⇒ n + 0 = n)).
  − reflexivity.
  − intros n IHn. simpl. f_equal. exact IHn.
Qed.
```

[1] Coq uses the capital letter P for the argument p. We follow our own conventions and use the letter p. The difference will not matter in the following.

The proof applies Coq's induction lemma *nat_ind* with the right predicate p. This yields two subgoals, one for the basis step and one for the inductive step. Note the introduction of the *inductive hypothesis IHn* in the script for the inductive step.

Here is a second example for the use of the induction lemma *nat_ind*.

```
Goal ∀ n m, n + S m = S (n + m).

Proof.
  intros n m. revert n.
  apply (nat_ind (fun n ⇒ n + S m = S (n + m))) ; simpl.
  − reflexivity.
  − intros n IHn. f_equal. exact IHn.
Qed.
```

The proof would also go through with a more general inductive predicate p quantifying over m. In this case the first line of the proof script would be deleted. See Exercise 4.1.1.

We now know how to construct inductive proofs with the induction lemma *nat_ind*. Next we explain how the lemma *nat_ind* is defined. Speaking type theoretically, we have to define a function that has the type of *nat_ind*. We do this with the definition command using a recursive abstraction.

```
Definition nat_ind (p : nat → Prop) (basis : p 0) (step : ∀ n, p n → p (S n))
 : ∀ n, p n := fix f n := match n return p n with
                            | 0 ⇒ basis
                            | S n' ⇒ step n' (f n')
                          end.
```

Note that the match specifies a **return type function** $\lambda n.pn$. This is necessary since the two rules of the match have different return types. The return type of the first rule is $p\,0$, and the return type of the second rule is $p(S\,n')$. The return types of the rules are obtained by applying the return type function to the left hand sides of the rules.

Exercise 4.1.1 Prove the following goal by applying the induction lemma *nat_ind* immediately (i.e., don't introduce n and m).

```
Goal ∀ n m, n + S m = S (n + m).
```

Exercise 4.1.2 We consider an induction lemma for list types.

a) Complete the following definition of an induction lemma for list types.

```
Definition list_ind (X : Type) (p : list X → Prop)
           (basis : p nil)
           (step : ∀ (x : X) (A : list X), p A → p (x::A))
 : ∀ A : list X, p A :=
```

b) Prove that list concatenation is associative using the induction lemma *list_ind*.

c) Use the command *Check* to find out the type of the induction lemma Coq provides for list types. Since Coq's lemma is also bound to the name *list_ind*, you will have to undo your definition to see the type.

4.2 Primitive Recursion

Primitive recursion is a basic computational idea for natural numbers first studied in the 1930's. We saw a formulation of primitive recursion called iteration in Section 1.12. The basic idea is to apply a step function n-times to a start value. We formalized the idea with a function *nat_iter* taking the number n, the step function, and the start value as arguments.[2] For the application of *nat_iter* the type of *nat_iter* is crucial. The more general the type of *nat_iter*, the more recursive functions can be expressed with *nat_iter*.

We will now formulate primitive recursion as a function *prec* that can express both the computational function *nat_iter* and the induction lemma *nat_ind*. We base the definition of *prec* on two equations.

$$\begin{aligned} prec\ x\ f\ 0 &= x \\ prec\ x\ f\ (S\,n) &= f\ n\ (prec\ x\ f\ n) \end{aligned}$$

Compared to *nat_iter*, we have reordered the arguments and now work with a step function that takes the number of iterations so far as an additional first argument. For instance, *prec* x f 3 = f 2 (f 1 (f 0 x)). From the equations it is clear that *prec* can express *nat_iter*.

We now come to the type of *prec*. We take the type of the induction lemma *nat_ind* where the type of p is generalized to $nat \to Type$ (recall that propositions are types).

$$prec\ :\ \forall p{:}nat \to Type,\ \ p\,0\ \to\ (\forall n{:}nat,\ p\,n \to p\,(Sn))\ \to\ \forall n{:}nat,\ p\,n$$

Given the type and the equations, the definition of *prec* is straightforward.[3]

[2] In Section 1.12 we used the short name *iter* for *nat_iter*. The function *nat_iter* is defined in Coq's standard library.

[3] Due to implicit argument mode, p is accommodated as implicit argument of *prec*.

```
Definition prec (p : nat → Type) (x : p 0) (f : ∀ n, p n → p (S n))
  : ∀ n, p n := fix F n := match n return p n with
                             | 0 ⇒ x
                             | S n' ⇒ f n' (F n')
                           end.
```

Note that the definition of *prec* is identical with the definition of the induction lemma *nat_ind* except for the more general type of p. Since $nat \to Prop$ is a subtype of $nat \to Type$, we can instantiate the type of *prec* to the type of *nat_ind*.

```
Check fun p : nat → Prop ⇒ prec (p:= p).
```

$$\forall p : nat \to Prop,\ \ p\,0 \ \to\ (\forall n : nat,\ p\,n \to p\,(Sn)) \ \to\ \forall n : nat,\ p\,n$$

Thus we can use *prec* to obtain the induction lemma *nat_ind*.

```
Lemma nat_ind (p : nat → Prop) :
  p 0 → (∀ n, p n → p (S n)) → ∀ n, p n.
Proof. exact (prec (p:=p)). Qed.
```

We can also define arithmetic functions like addition with *prec*.

```
Definition add := prec (fun y ⇒ y) (fun _ r y ⇒ S (r y)).
Compute add 3 7.
% 10
```

We prove that *add* agrees with the addition provided by Coq's library.

```
Goal ∀ x y, add x y = x + y.
Proof. intros x y. induction x ; simpl ; congruence. Qed.
```

As announced before, we can obtain the function *nat_iter* from *prec*.

```
Goal ∀ X f x n ,
       nat_iter n f x = prec (p:= fun _ ⇒ X) x (fun _ ⇒ f) n.
Proof. induction n ; simpl ; congruence. Qed.
```

If we were allowed only a single use of *fix* for *nat*, we could define *prec* and then express all further recursions with *prec*. In fact, since *prec* can also express matches on *nat*, we can work without *fix* and *match* for *nat* as long as we have *prec*.

Coq automatically synthesizes a primitive recursion function *X_rect* for every inductive type X. Print *nat_rect* to see the primitive recursion function for *nat*.

Exercise 4.2.1 Prove *prec* = *nat_rect*.

Exercise 4.2.2 Prove that *prec* satisfies the two characteristic equations stated at the beginning of this section.

Exercise 4.2.3 Show that *prec* can express multiplication and factorial.

Exercise 4.2.4 Show that *prec* can express the predecessor function *pred*.

Exercise 4.2.5 Show that *prec* can express matches for *nat*. Do this by completing and proving the following goal.

```
Goal ∀ X x f n ,
       match n with O ⇒ x | S n' ⇒ f n' end = prec   ...   .
```

4.3 Size Induction

Given a predicate $p : X \to Prop$, **size induction** says that we can prove px using the assumption that we have a proof of py for every y whose size is smaller than the size of x. The sizes of the elements of X are given by a size function $X \to nat$. We formulate size induction as a proposition and prove it with natural induction (i.e., structural induction on *nat*).

```
Lemma size_induction X (f : X → nat) (p : X → Prop) :
  (∀ x, (∀ y, f y < f x → p y) → p x) →
  ∀ x, p x.
Proof.
  intros step x. apply step.
  assert (G: ∀ n y, f y < n → p y).
  { intros n. induction n.
    - intros y B. exfalso. omega.
    - intros y B. apply step. intros z C. apply IHn. omega. }
  apply G.
Qed.
```

The proof is clever. It introduces the step function *step* of the size induction and x, leaving us with the claim px. By applying *step* we obtain the claim $\forall y{:}X.\ fy < fx \to py$. The trick is now to generalize this claim to the more general claim $\forall n\, \forall y{:}X.\ fy < n \to py$, which can be shown by natural induction on n.

Note that we have not seen a definition of Coq's order predicate "<" for *nat*. The details of the definition do not matter since we are using the automation tactic *omega* to solve goals involving the order predicate.

Exercise 4.3.1 The principle of *complete induction* can be formulated as follows.

```
Lemma complete_induction (p : nat → Prop) :
  (∀ x, (∀ y, y < x → p y) → p x) → ∀ x, p x.
```

a) Prove the lemma using the lemma *size_induction*.

b) Prove the lemma using natural induction.

Exercise 4.3.2 Define your own order predicate *lt* : *nat* → *nat* → *Prop* and prove the size induction lemma for your order predicate. Hint: Define *lt* with the boolean order test *leb* from Section 1.3 and prove the following lemma by induction on x. No other lemma will be needed.

```
Lemma lt_tran x y z : lt x y → lt y (S z) → lt x z.
```

4.4 Equational Specification of Functions

It is often instructive to specify a recursive function by a system of equations. We have seen such equational specifications for the functions *plus*, *nat_iter*, and *prec*. For arithmetic functions like addition and multiplication equational specifications where already used by Dedekind. In Coq, we can express equational specifications as predicates. Given a specification, we may prove that there is a function satisfying the specification (satisfiability) and that any two function satisfying the specification agree on all arguments (uniqueness). We start with a somewhat unusual specification of addition.

```
Definition addition (f : nat → nat → nat) : Prop :=
  ∀ x y,
    f x 0 = x ∧
    f x (S y) = f (S x) y.
Lemma addition_existence :
  addition plus.
Proof. intros x y. omega. Qed.
Lemma addition_uniqueness f g :
  addition f → addition g → ∀ x y, f x y = g x y.
Proof.
  intros A B x y. revert x. induction y ; intros x.
  - destruct (A x 0) as [A' _]. destruct (B x 0) as [B' _]. congruence.
  - destruct (A x y) as [_ A']. destruct (B x y) as [_ B']. specialize (IHy (S x)). congruence.
Qed.
```

From the example we learn that an equational specification is abstract in that is does not say how the specified function is realized. The specification *addition* suggests a tail recursive function matching on the second argument. The function *plus* from the library recurses on the first argument and is not tail recursive. Nevertheless, *plus* satisfies the specification *addition*.

Our second example specifies a function known as Ackermann's function.[4]

[4] Ackermann's function grows rapidly. For example, for 4 and 2 it yields a number of 19,729 decimal digits. It was designed as a terminating recursive function that cannot be computed with first-order primitive recursion. In Exercise 4.4.2 you will show that Ackermann's function can be computed with higher-order primitive recursion.

```
Definition ackermann (f : nat → nat → nat) : Prop :=
  ∀ m n,
    f O n = S n  ∧
    f (S m) O = f m 1  ∧
    f (S m) (S n) = f m (f (S m) n).
```

The satisfiability and uniqueness of this specification can be argued as follows. Since for any two arguments exactly one of the three equations applies, f exists and is unique if the application of the equations terminates. This is the case since either the first argument is decreased, or the first argument stays the same and the second argument is decreased.

The above termination argument is outside the scope of Coq's termination checker. Coq insists that every fix comes with an argument that is structurally decreased by every recursive application. The problem can be solved by formulating Ackermann's function with two nested recursions.

```
Definition ack : nat → nat → nat :=
  fix f m := match m with
               | O ⇒ S
               | S m' ⇒ fix g n := match n with
                                    | O ⇒ f m' 1
                                    | S n' ⇒ f m' (g n')
                                  end
             end.
```

Note that *ack* is defined as a recursive function that yields a recursive function when give an argument greater than 0. Each of the two recursions is structural on its argument. The correctness proof for *ack* is straightforward.

```
Goal ackermann ack.
```

```
Proof. unfold ackermann. auto. Qed.
```

We can also show that any two functions satisfying the specification *ackermann* agree on all arguments.

```
Goal ∀ f g x y,  ackermann f → ackermann g → f x y = g x y.
```

```
Proof.
  intros f g x y A B. revert y. induction x ; intros y.
  - destruct (A 0 y) as [C _]. destruct (B 0 y) as [D _]. congruence.
  - induction y.
    + destruct (A x 0) as [_ [C _]]. destruct (B x 0) as [_ [D _]]. congruence.
    + destruct (A x y) as [_ [_ C]]. destruct (B x y) as [_ [_ D]]. congruence.
Qed.
```

Exercise 4.4.1 Write an equational specification for multiplication and prove that Coq's multiplication satisfies the specification. Also prove that two functions agree on all arguments if they satisfy the specification. Do the same for subtraction.

Exercise 4.4.2 Write an Ackermann function using *prec* rather than fix and match. Prove that your function satisfies the specification *ackermann*.

Exercise 4.4.3 We specify primitive recursion as follows.

```
Definition primitive_recursion
(r : ∀ p : nat → Type, p 0 → (∀ n, p n → p (S n)) → ∀ n, p n)
: Prop :=
  ∀ p x f n,
    let r := r p x f in
    r 0 = x ∧
    r (S n) = f n (r n).
```

Note that a local declaration with let is used to write the specifying equations in compact form. Show that *prec* satisfies the specification. Also prove that two functions agree on all arguments if they satisfy the specification.

Exercise 4.4.4 Give an equational specification for *nat_iter*. Prove that *nat_iter* satisfies the specification and that the specification is unique.

Coq Summary

New Features of the Term Language

- Matches can be specified with a return type function. See Section 4.1, definition of *nat_ind*.

5 Truth Value Semantics and Elim Restriction

Coq's type theory is designed such that the truth value semantics of propositions commonly used in Mathematics can be consistently assumed. This design comes at the price of the elim restriction, which restricts matches on proofs such that proofs must be returned. The elim restriction severely restricts the computational use of proofs.

5.1 Truth Value Semantics

In Mathematics one assumes that a proposition is either true or false. More specifically, one assumes that every proposition denotes a truth value, which is either true or false. With the boolean definition of disjunction it then follows that for any proposition p the disjunction $p \vee \neg p$ is true.

Given the fact that propositions denote truth values, we could consider two propositions equal if they denote the same truth value. This assumption is made in boolean logic as well as in Church-Henkin simple type theory.

We formulate the mathematical assumption that a proposition is either true or false as a proposition in Coq:

```
Definition TVS : Prop := ∀ X : Prop, X=True ∨ X=False.
```

We can now ask whether Coq can prove *TVS* or $\neg$*TVS*. It turns out that Coq can prove neither of the two. That Coq cannot prove *TVS* seems intuitively clear since there is nothing in the basic proof rules that would give us a proof of *TVS*. On the other hand, that Coq cannot prove $\neg$*TVS* is not clear at all given the fact that propositions in Coq are obtained as types.

We call a proposition p **consistent** in Coq if Coq cannot prove $\neg p$. Moreover, we call a proposition p **independent** in Coq if Coq can prove neither p nor $\neg p$. Note that every independent proposition is consistent, but not vice versa (e.g. *True* is consistent but not independent).

Above we have stated that *TVS* is independent in Coq. The consistency of *TVS* in Coq does not come for free. In fact, the design of Coq's type theory has been carefully arranged so that *TVS* is consistent. To obtain the consistency of *TVS*,

Coq imposes a severe typing restriction known as the *elim restriction*. To prepare the discussion of the elim restriction, we first consider some conseqences of *TVS*.

First we show that *TVS* implies *XM* (excluded middle). This follows from the fact that *True* and *False* satisfy *XM* (i.e., $True \vee \neg True$ and $False \vee \neg False$ are provable).

```
Goal TVS → XM.
```

```
Proof. intros A X. destruct (A X) as [B|B] ; rewrite B ; auto. Qed.
```

Another important consequence of *TVS* is **proof irrelevance**.

```
Definition PI : Prop := ∀ (X : Prop) (A B : X), A=B.
```

Proof irrelevance says that a proposition has at most one proof. Here is a proof that truth value semantics implies proof irrelevance.

```
Goal TVS → PI.
```

```
Proof.
  intros A X B C. destruct (A X) ; subst X.
  – destruct B, C. reflexivity.
  – contradiction B.
Qed.
```

The proof exploits that the proposition *True* has exactly one proof, a fact following from the inductive definition of *True*. The script uses the tactic *subst*, which eliminates a variable x if there is an assumption $x = s$ such that x does not occur in s.

A third consequence of *TVS* is **propositional extensionality**.

```
Definition PE : Prop := ∀ X Y : Prop, (X ↔ Y) → X=Y.
```

Propositional extensionality says that two propositions are equal if they are equivalent.

```
Goal TVS → PE.
```

```
Proof. intros A X Y B. destruct (A X), (A Y) ; subst X Y ; tauto. Qed.
```

Finally, we show that excluded middle and propositional extensionality together imply truth value semantics.

```
Goal XM → PE → TVS.
```

```
Proof.
  intros xm pe X. destruct (xm X) as [A|A].
  – left. apply pe. tauto.
  – right. apply pe. tauto.
Qed.
```

We now know that *TVS* and $XM \wedge PE$ are equivalent.

Exercise 5.1.1 Make sure you can prove $TVS \leftrightarrow XM \wedge PE$

Exercise 5.1.2 Prove $TVS \rightarrow PE$ without using the tactic *subst*. Use the tactic *rewrite* instead.

5.2 Elim Restriction

If we have a surjective function $f : X \rightarrow bool$, then the type X must contain at least two elements.

```
Goal ∀ X (f : X → bool),  surjective f → ∃ x y : X, x ≠ y.
Proof.
  intros X f  A. destruct (A true) as [x B].  destruct (A false) as [y C].
  ∃ x, y. congruence.
Qed.
```

Now consider the inductive proposition

```
Inductive bp : Prop := P1 : bp | P2 : bp.
```

which by definition has two proofs *P1* and *P2*. Given the match for *bp*, it is easy to construct a surjective function $bp \rightarrow bool$, it seems. However, Coq rejects the following match on $x : bp$:

```
Check fun x : bp ⇒ match x with P1 ⇒ true | P2 ⇒ false end.
% Error : Incorrect elimination of "x" ...
```

The reason is the so-called **elim restriction**: A match on a proof (i.e., on an element of a proposition) is only allowed if the match returns a proof. The above match does not return a proof and hence it is rejected by the elim restriction. One important reason for the elim restriction is that it is needed so that truth value semantics is consistent in Coq. Without the elim restriction, we get a surjective function from *bp* to *bool*, which entails that the proposition *bp* has two different elements. This however contradicts proof irrelevance, which says that no proposition has more than one proof. Since *TVS* entails *PI*, not having the elim restriction would mean that truth value semantics is inconsistent in Coq.

There are a few exceptions to the elim restriction, all of them being consistent with proof irrelevance. For instance, there is no restriction on the matches for *True* and *False*. The remaining exceptions will be discussed in Section 7.5.

We give another example illustrating the elim restriction. Consider the following lemma, which establishes the existence of a so-called *Skolem function* for total predicates $p : X \rightarrow Y \rightarrow Prop$ where Y is a proposition.

```
Lemma Prop_Skolem (X : Type) (Y : Prop) (p : X → Y → Prop) :
  (∀ x, ∃ y, p x y) → ∃ f, ∀ x, p x (f x).
```

```
Proof.
 intros A.
 ∃ (fun x ⇒ let (y,_) := A x in y).
 intros x.
 destruct (A x) as [y B].
 exact B.
Qed.
```

The let notation in the definition of the Skolem function is notational sugar for the one-constructor match

```
match A x return Y with ex_intro y _ ⇒ y end
```

This match on the proof $A\ x$ is legal since it returns a proof y. If we generalize the lemma to $Y : \mathit{Type}$, the proof script fails since the match now violates the elim restriction.

We will speak of proper types and proper values. A **proper type** is a type that is not a proposition, and a **proper value** is an element of a proper type. Thus a type is not proper if and only if it is a proposition, and a value is not proper if and only if it is a proof. Examples of proper types are *nat*, *list nat*, and $\mathit{nat} \to \mathit{nat}$. Example of proper values are 5, *cons*, and $\lambda x : \mathit{nat}.x$.

Exercise 5.2.1 Prove that the proposition $\forall X : \mathit{Type}.\ X = \mathit{True} \vee X = \mathit{False}$ is inconsistent in Coq. Note that *TVS* is a weaker statement where X is restricted to propositional types. Find out where your proof breaks if you apply it to *TVS*.

5.3 Propositional Extensionality Entails Proof Irrelevance

It turns out that propositional extensionality entails proof irrelevance. This is a surprising result with a very interesting proof. The proof rests on a fixpoint theorem that given a surjective function $X \to X \to Y$ states that every function $Y \to Y$ has a fixpoint.

```
Lemma sur_fixpoint X Y (f : X → X → Y) (g : Y → Y) :
  surjective f → ∃ y, g y = y.
Proof.
  intros A.
  pose (h x := g (f x x)).
  destruct (A h) as [x B].
  ∃ (h x). unfold h at 2. rewrite ← B. reflexivity.
Qed.
```

The proof of the theorem should remind you of Cantor's theorem. In fact, we can obtain Cantor's theorem as a corollary of the surjective fixpoint theorem by specializing to $Y := \mathit{Prop}$ and $g := \mathit{not}$.

We now assume *PE* and prove *PI*. It suffices to prove $P1 = P2$ for the two constructors of *bp* since given two proofs x and y of a proposition X we can obtain a function f such that $f\ P1 = x$ and $f\ P2 = y$ using the match for *bp*. For $P1 = P2$ it suffices to show that the "negation" function mapping *P1* to *P2* and *P2* to *P1* has a fixpoint. This we obtain with the surjective fixpoint theorem. To do so, we need a surjective function $bp \to bp \to bp$. Such a function is easy to obtain if we have the equation $(bp \to bp) = bp$. This finishes the proof since the equation is a straightforward consequence of *PE*.

```
Goal PE → PI.

Proof.
  intros pe.
  cut (P1=P2).
  { intros A X B C.
    change (B = match P1 with P1 ⇒ C | P2 ⇒ B end).
    rewrite A. reflexivity. }
  pose (neg x := match x with P1 ⇒ P2 | P2 ⇒ P1 end).
  cut (∃ P, neg P = P).
  { unfold neg. intros [[|] C].
    - symmetry. exact C.
    - exact C. }
  cut (∃ f : bp → bp → bp, surjective f).
  { intros [f A]. apply (sur_fixpoint (f:=f)). exact A. }
  cut (bp = (bp → bp)).
  { intros A. rewrite ← A. ∃ (fun x ⇒ x). intros x. ∃ x. reflexivity. }
  apply pe. split ; auto using P1.
Qed.
```

Note the use of the tactic *cut* to realize the backwards reasoning of the proof outline. In the last line the automation tactic *auto* is used with a suffix telling it to use the proof constructor $P1 : bp$.

Exercise 5.3.1 Prove Cantor's theorem using the surjective fixpoint theorem *sur_fixpoint*.

```
Lemma Cantor X :
  ¬ ∃ f : X → X → Prop, surjective f.
```

Exercise 5.3.2 Two types are *isomorphic* if there are commuting functions back and forth.

```
Definition iso (X Y : Type) : Prop :=
  ∃ f : X → Y, ∃ g : Y → X, ∀ x y, g (f x) = x ∧ f (g y) = y.
```

Propositional univalence is the property that propositions are equal if they are isomorphic.

Definition PU : Prop := ∀ X Y : Prop, iso X Y → X = Y.

It turns out that propositional extensionality factors into propositional univalence and proof irrelevance, that is, $PE \leftrightarrow PU \wedge PI$. We have already shown $PE \rightarrow PI$. Prove $PE \rightarrow PU$ and $PU \rightarrow PI \rightarrow PE$ to establish the equivalence.

Coq Summary

New Tactics

subst, *cut*.

Tactic *auto* with *using*

The tactic *auto* can be enhanced with lemmas and constructors specified with a *using* suffix. See the proof of the goal $PE \rightarrow PI$ in Section 5.3.

6 Sum and Sigma Types

Sum types and sigma types are non-propositional variants of disjunctions and existential quantifications. Since they are proper types, sum and sigma types are not subject to the elim restriction. The elements of sum and sigma types are computational values carrying a proof. With sum and sigma types we can write certifying functions whose results contain correctness proofs.

6.1 Boolean Sums and Certifying Tests

Boolean sums are disjunctions placed in *Type* rather than *Prop*. Coq's standard library defines boolean sums as follows.

```
Inductive sumbool (X Y : Prop) : Type :=
| left : X → sumbool X Y
| right : Y → sumbool X Y.

Arguments left {X} {Y} _.
Arguments right {X} {Y} _.

Notation "{ X } + { Y }" := (sumbool X Y).
```

Boolean sums are like disjunctions except for the crucial difference that they are proper types rather than propositions. Thus boolean sums are not subject to the elim restriction. We call the elements of boolean sums **decisions**. We can think of a decision as a proof-carrying boolean value, or as a proof of a disjunction on which we can freely match.

A **certifying test** is a function that yields a decision. Coq's library provides many certifying tests. For instance, there is a certifying test for the order on natural numbers:

$$le\_dec : \forall x\, y : nat,\ \{x \le y\} + \{\neg(x \le y)\}$$

The type of *le_dec* tells us that *le_dec* is a function that takes two numbers x and y and returns a decision containing a proof of either $x \le y$ or $\neg x \le y$. With *le_dec* a minimum function can be written as follows:

```
Definition min (x y : nat) : nat :=
  if le_dec x y then x else y.
```

```
Compute min 7 3.
% 3 : nat
```

Note the use of the if-then-else notation in the definition of *min*. The if-then-else notation is available for all inductive types with two constructors and expands to a match. The definition of *min* expands as follows.

```
Set Printing All.
Print min.
  min = fun x y : nat ⇒ match le_dec x y with left _ ⇒ x | right _ ⇒ y end
Unset Printing All.
```

We prove the correctness of *min*.

```
Goal ∀ x y, (x ≤ y → min x y = x) ∧ (y ≤ x → min x y = y).
```

```
Proof.
  intros x y. split ; intros A.
  - unfold min. destruct (le_dec x y) as [B|B].
    + reflexivity.
    + omega.
  - unfold min. destruct (le_dec x y) as [B|B].
    + omega.
    + reflexivity.
Qed.
```

The proof can be shortened to a one-liner.

```
intros x y. split ; intros A ; unfold min ; destruct (le_dec x y) ; omega.
```

The Coq library *Compare_dec* offers many certifying tests for natural numbers. Here are a few.

$$\begin{aligned} le\_lt\_dec &: \forall x\,y : nat,\ \{x \le y\} + \{y < x\} \\ le\_ge\_dec &: \forall x\,y : nat,\ \{x \le y\} + \{x \ge y\} \\ le\_gt\_dec &: \forall x\,y : nat,\ \{x \le y\} + \{x > y\} \\ lt\_eq\_lt\_dec &: \forall x\,y : nat,\ \{x < y\} + \{x = y\} + \{y < x\} \end{aligned}$$

The type of *lt_eq_lt_dec* needs explanation. Since boolean sums are proper types taking propositions as arguments, they cannot be nested. Coq solves the problem with an additional sum type *sumor* and a concomitant notation.

```
Set Printing All.
Check {True} + {False} + {False}.
% sumor (sumbool True False) False : Type
Unset Printing All.
```

The type *sumor* and the accompanying notation are defined as follows.

```
Inductive sumor (X : Type) (Y : Prop) : Type :=
| inleft : X → sumor X Y
| inright : Y → sumor X Y.

Notation "X + { Y }" := (sumor X Y).
```

Exercise 6.1.1 Prove the following goal.

```
Goal ∀ X Y : Prop, {X} + {Y} → X ∨ Y.
```

Explain why you cannot prove the other direction $\forall X\ Y : \mathit{Prop},\ X \vee Y \to \{X\} + \{Y\}$.

Exercise 6.1.2 Prove the following goals.

```
Goal ∀ x y, if le_dec x y then x ≤ y else ¬ x ≤ y.
Goal ∀ x y, if le_dec x y then x ≤ y else x > y.
```

6.2 Inhabitation and Decidability

An **inhabitant** of a type is an element of a type. So saying that x is an inhabitant of a type X means the same as saying that x is a member of X, or that x is an element of X. We say that a type is **inhabited** if it has at least one inhabitant. So a type is inhabited if and only if it is nonempty. Coq's library comes with an inductive predicate for inhabitation.

```
Inductive inhabited (X : Type) : Prop :=
| inhabits : X → inhabited X.
```

A proposition is inhabited if and only if it is provable.

```
Goal ∀ X : Prop, inhabited X ↔ X.
Proof.
  split.
  – intros [A] ; exact A.
  – intros A. constructor. exact A.
Qed.
```

Note the use of the tactic *constructor*. Here it has the same effect as the command *apply inhabits*. In general, the tactic *constructor* tries to prove an inductive proposition by applying a constructor of the definition of the proposition. The tactic *constructor* is convenient since the name of the constructor needs not to be given.

We say that a proposition p is **decidable** if the sum $\{p\} + \{\neg p\}$ is inhabited. To have a concise notation for decidable propositions, we define the function

```
Definition dec (X : Prop) : Type := {X} + {¬ X}.
```

Note that *dec* is not a predicate. An element of *dec X* is a decision that gives us a proof of either X or $\neg X$. We call a member of *dec X* a **decision of** X.

The certifying test *le_dec* from the standard library tells us that all propositions of the form $x \leq y$ are decidable.

```
Check le_dec : ∀ x y : nat, dec (x ≤ y).
```

We define a function that converts a decision to a boolean by forgetting the proof coming with the decision.

```
Definition dec2bool (X : Prop) (d : dec X) : bool :=
  if d then true else false.
Compute dec2bool (le_dec 2 3).
% true : bool
```

We now establish the decidability of *True*. To do so, we construct a decision of type *dec True*. This is easy since the constructor *I* is a proof of *True*.

```
Definition True_dec : dec True := left I.
```

The decidability of *False* is also easy to establish.

```
Definition False_dec : dec False := right (fun A ⇒ A).
```

In the next section we will show that implications, conjunctions, and disjunctions of decidable propositions are decidable.

Exercise 6.2.1 Prove the following goal.

```
Goal ∀ X : Type, X → inhabited X.
```

Note that $X \to$ *inhabited* X is notation for the proposition $\forall x{:}X$, *inhabited* X. Explain why you cannot prove that the type $\forall X : Type$, *inhabited* $X \to X$ is inhabited.

Exercise 6.2.2 Prove $\forall X\, Y : Prop.\ X \vee Y \leftrightarrow inhabited\,(\{X\} + \{Y\})$.

Exercise 6.2.3 Prove $\forall X : Prop.\ dec\, X \to X \vee \neg X$.

6.3 Writing Certifying Tests

We show that implication preserves decidability of propositions.

```
Definition impl_dec (X Y : Prop) : dec X → dec Y → dec (X → Y).
  intros A [B|B].
  – left. auto.
  – destruct A as [A|A].
    + right. auto.
    + left. tauto.
Defined.
```

The definition of the function *impl_dec* should come as a surprise. This is the first time we construct a member of a type that is not a proposition with a script. Use the command *Print impl_dec* to see that the function constructed is in fact similar to what you would have written by hand. Note that the tactics *left* and *right* so far used for disjunctions also work for boolean sums. In fact, *left* and *right* will work for every inductive type with two constructors. We can compute with the certifying test *impl_dec*.

```
Check impl_dec (le_dec 3 2) False_dec.
% dec(3 ≤ 2 → False)

Compute (dec2bool (impl_dec (le_dec 3 2) False_dec)).
% true : bool
```

Here is a certifying equality test for *nat*.

```
Definition nat_eq_dec (x y : nat) : dec (x=y).

  revert y. induction x ; simpl ; intros [|y].
  - left. auto.
  - right. auto.
  - right. auto.
  - destruct (IHx y).
    + left. congruence.
    + right. congruence.
Defined.

Compute dec2bool (nat_eq_dec 3 3).
% true : bool
```

This is the first time we use the induction tactic to synthesize a function returning a proper value. When you print *nat_eq_dec*, you will see that the induction tactic realizes the necessary recursion with *nat_rect*, an automatically generated function providing primitive recursion for *nat*.

A more convenient way to obtain a certifying equality test for *nat* is using the automation tactic *decide equality*.

```
Goal ∀ x y : nat, dec (x=y).

Proof. unfold dec. decide equality. Qed.
```

The standard library offers a boolean test *leb* for the order on *nat* and a correctness lemma

$$leb\_iff : \forall x\, y : nat,\ leb\, x\, y = true \leftrightarrow x \leq y$$

We can use *leb* and *leb_iff* to write a certifying test for the order on *nat*.

```
Definition le_dec (x y : nat) : dec (x ≤ y).
  destruct (leb x y) eqn:A.
  – left. apply leb_iff. exact A.
  – right. intros B. apply leb_iff in B. congruence.
Defined.
```

Note that the script for the second subgoal applies the correctness lemma *leb_iff* to the assumption *B* using the tactic *apply*. This is the first time we apply a lemma to an assumption using the tactic *apply*. It is also possible to rewrite assumptions with the tactic *rewrite*. We have already mentioned that the conversion tactics can be applied to assumptions. To apply a tactic to an assumption *A*, one ends the command with "*in A*".

Decidability of propositions propagates through logical equivalences. That is, if X and Y are equivalent propositions, then X is decidable if and only if Y is decidable.

```
Definition dec_prop_iff (X Y : Prop) : (X ↔ Y) → dec X → dec Y.
  intros A [B|B].
  – left. tauto.
  – right. tauto.
Defined.
```

There are many undecidable propositions in Coq. A prominent example of an undecidable proposition is excluded middle (i.e., $XM := \forall X : Prop, X \vee \neg X$). In fact, a proposition is undecidable in Coq if and only if it is independent in Coq.

Exercise 6.3.1 Prove the following goals.

```
Goal ∀ X : Prop, inhabited X → dec X.
Goal ∀ X : Prop, dec X → dec (inhabited X).
Goal ∀ X : Prop, dec (inhabited X) → dec X.
```

Exercise 6.3.2 Complete the following definitions establishing the fact that decidable propositions are closed under conjunction and disjunction.

```
Definition and_dec (X Y : Prop) : dec X → dec Y → dec (X ∧ Y).
Definition or_dec (X Y : Prop) : dec X → dec Y → dec (X ∨ Y).
```

Exercise 6.3.3 Write a certifying test $\forall x\, y : nat.\ \{x < y\} + \{x = y\} + \{y < x\}$.

Exercise 6.3.4 Write a certifying equality test for *bool*.

a) Use the automation tactic *decide equality*.

b) Write the test without using *decide equality*.

Exercise 6.3.5 Compare the certifying equality test *nat_eq_dec* from this section with the boolean equality test *nat_eqb* and its correctness lemma *nat_eqb_agrees* from Section 3.9. We can say that the certifying test combines the boolean test and its correctness lemma into a single function.

a) Define *nat_eqb* and *nat_eqb_agrees* using *nat_eq_dec*.

b) Define *nat_eq_dec* using *nat_eqb* and *nat_eqb_agrees*.

Exercise 6.3.6 Consider the boolean test *leb* and the certifying test *le_dec* from the standard library.

a) Prove the correctness lemma for *leb*.

```
Lemma leb_iff x y : leb x y = true ↔ x ≤ y.
```

b) Define the certifying test *le_dec* using the induction tactic. Follow the definition of *nat_eq_dec* shown above. Compare this definition of *le_dec* with the proof of *leb_iff*.

Exercise 6.3.7 Write a function that given a certifying equality test for a type X yields a certifying equality test for *list X*. Write the function with and without the automation tactic *decide equality*.

Exercise 6.3.8 Complete the following definition. It establishes a function translating a boolean decision of a proposition X into a certifying decision of X.

```
Definition bool2dec (X : Prop) (b : bool) : (X ↔ b = true) → dec X.
```

Exercise 6.3.9 (Program Synthesis) One can use tactics to synthesize ordinary functions not involving proofs. Here are two examples.

```
Definition cas (X Y Z : Type) : (X * Y → Z) → X → Y → Z.
  intros f x y. exact (f (x,y)).
Defined.
Definition car (X Y Z : Type) : (X → Y → Z) → X * Y → Z.
  intros f [x y]. exact (f x y).
Defined.
```

Use the command *Print* to see the synthesized functions. It is also possible to synthesize recursive functions like addition.

```
Definition add : nat → nat → nat.
  fix f 1. intros x y. destruct x as [|x'].
  - exact y.
  - exact (S (f x' y)).
Defined.
```

Use the command *Show Proof* after each tactic to see the partial code of the function synthesized by the tactic.

6.4 Definitions and Lemmas

A definition in Coq is either **inductive** or **plain**. Inductive definitions extend the underlying type theory with new inhabitants obtained with constructors. Plain definitions do not extend the type theory but introduce names for already existing inhabitants. A **plain definition** takes the form $x : t := s$ where x is a name and t and s are terms. The term t must be a type and s must be a member of t. We call x the **name of the definition**, t the **type of the definition**, and s the **body of the definition**. The type of a plain definition acts as the type of the name of the definition.

A plain definition can be either **transparent** or **opaque**. If the definition is transparent, the name and the body of the definition are convertible (i.e., unfolding and folding of the name, known as delta conversion). If the definition is opaque, the name is abstract and cannot be unfolded. Thus all we know about an opaque name is that it is an inhabitant of its type.

Opaque definitions are a means of abstraction. Given an **opaque name** x (i.e., a name introduced by an opaque definition), we can use the specification of x (i.e., the type of x) but not the implementation of x (i.e., the body of the opaque definition of x). Thus every use of an opaque name x will be compatible with every implementation of x. Opaque names are as abstract as **variables** introduced with lambda abstractions, matches, or sections.

A transparent definition can be stated in Coq with a command of the form *Definition* $x : t := s$ or a sequence of commands taking the form

Definition $x : t$. $tactic_1 \cdots tactic_n$ **Defined**.

The tactics in the long form synthesize the body s of the definition. If we replace the command *Defined* in the long form with the command *Qed*, we obtain an opaque definition. The command *Definition* in the long form can be replaced with the command *Lemma*, which has no effect. We use the command *Lemma* only for sequences of the following form.[1]

Lemma $x : t$. **Proof**. $tactic_1 \cdots tactic_n$ **Qed**.

We also use the command sequence

Goal t. **Proof**. $tactic_1 \cdots tactic_n$ **Qed**.

This sequence has the same effect as the sequence starting with *Lemma* except that the missing name x is automatically generated by Coq.

In Coq, a **lemma** is an opaque name established with an opaque definition. The statement of the lemma is the type of the lemma, and the proof of the

[1] The Coq library doesn't always follow this convention. For instance, *le_dec* is defined with *Theorem* and *Defined*.

lemma is the hidden body of the definition. The proof of a lemma certifies that the type of the lemma is inhabited. If we work with a lemma, the uses of the lemma cannot see the proof of the lemma. So all uses of a lemma are abstract in that they do not make any assumptions about the proof of the lemma. This agrees with the mathematical use of lemmas.

There is also an engineering reason for representing lemmas as opaque names in Coq. If lemmas were transparent names, they would be subject to unfolding and their (possibly complex) proofs would unnecessarily participate in conversion checking and type checking.

A **strong lemma** is a lemma whose type is not a proposition. Strong lemmas are a speciality of constructive type theory that don't seem to have a counterpart in Mathematics.

6.5 Decidable Predicates

Every function definable in Coq is computable. Because of opaque definitions, Coq's interpreter may fail to fully evaluate a function application. So the above statement is made with respect to an idealized interpreter treating all plain declarations as transparent.

Functions are described with terms in Coq. When an idealized interpreter evaluates a term describing a function, it will always end up with a term having one of the following forms:

- A lambda abstraction.
- A recursive abstraction.
- A constructor.
- A constructor application $c\,t_1 \ldots t_n$ where c is a constructor and $t_1, \ldots, t_n$ are $n \geq 1$ terms. Examples of functions obtained as constructor applications are *cons 3* and *prod nat*.

We call a predicate $p : X \to \mathit{Prop}$ **decidable** if the there is some function that yields for every $x : X$ a decision of $p\,x$.

```
Definition decidable (X : Type) (p : X → Prop) : Type := ∀ x, dec (p x).
```

If p and some function f : *decidable p* are definable in Coq, then f is a decision procedure for p and p is computationally decidable.

Coq can define undecidable predicates. An example of an undecidable predicate is $\lambda X : \mathit{Prop}.\, X \vee \neg X$. The undecidability of this predicate follows from the fact that *XM* is independent in Coq.[2]

[2] Note that by undecidable we mean not decidable in Coq. Predicates that are undecidable in Coq may be computationally decidable. On the other hand, predicates that are decidable in Coq are

```
Goal decidable (fun X : Prop ⇒ X ∨ ¬X) → XM.
Proof. intros A X.  destruct (A X) as [B|B] ;  tauto. Qed.
```

The notion of decidability extends to predicates with more than one argument. As is, Coq doesn't give us the possibility to define decidability of predicates in one go, we have to give the definition for each number $n \geq 1$ of arguments. If, more generally, Coq would allow us to define decidability for $n \geq 0$ arguments, decidability of propositions would fall out for $n = 0$.

Exercise 6.5.1 Every predicate equivalent to a boolean test is decidable. Prove the following goal to show this fact.

```
Goal ∀ (X : Type) (p : X → Prop) (f : X → bool),  (∀ x, p x ↔ f x = true) → decidable p.
```

6.6 Sigma Types

Sigma types are existential quantifications expressed as proper types. The elim restriction does not apply to sigma types. Given a type X and a predicate $p : X \to Prop$, the elements of the sigma type $\{x : X \mid p\,x\}$ can be seen as pairs consisting of a value $x : X$ and a proof of $p\,x$. Coq defines sigma types as follows.

```
Inductive sig (X : Type) (p : X → Prop) : Type :=
  exist : ∀ x : X, p x → sig p.
Notation "{ x  |  p }" := (sig (fun x ⇒ p)).
Notation "{ x : X  |  p }" := (sig (fun x : X ⇒ p)).
```

Consider the type $\forall x : nat,\ \{y \mid y = 2 * x\}$. The elements of this types are functions that take a number x and return a pair consisting of the number $2x$ and a proof of the proposition $y = 2 * x$. Here is a construction of such a function.

```
Definition double (x : nat) : { y | y = 2*x}.
  ∃ (2*x). reflexivity.
Defined.
Compute let (y,_) := double 4 in y.
% 8 : nat
```

Note the use of the tactic *exists* to construct a member of a sigma type. We will refer to functions that yield an element of a sum or sigma type as **certifying functions**. A certifying function combines a function and a correctness proof into a single object. The types of certifying functions can be seen as specifications. For instance, while the type $nat \to nat$ gives us little information about its inhabitants, the type $\forall x : nat,\ \{y \mid y = 2x\}$ gives us much more information.

We define a certifying function that divides its argument by 2.

always computationally decidable.

```
Definition div2_cert (n : nat) : {k | n = 2*k} + {k | n = 2*k + 1}.
  induction n.
  - left. ∃ 0. reflexivity.
  - destruct IHn as [[k A]|[k A]].
    + right. ∃ k. omega.
    + left. ∃ (S k). omega.
Defined.
```

The result type of *div2_cert* is obtained with yet another kind of sum type (needed since both constituents are proper types).

```
Inductive sum (X Y : Type) :=
| inl : X → sum X Y
| inr : Y → sum X Y.

Notation "x + y" := (sum x y) : type_scope.
```

Note that the definition of the "+" notation for *sum* is restricted to types. This way the string 2 + 4 will still elaborate to the term *plus 2 4*.

Based on the certifying division function *div2_cert* we define ordinary modulo and division functions and prove a correctness lemma.

```
Definition mod2 x := if div2_cert x then 0 else 1.

Definition div2 x := match div2_cert x with
                       | inl (exist k _) ⇒ k
                       | inr (exist k _) ⇒ k
                     end.

Goal ∀ x, x = 2 * div2 x + mod2 x.

Proof.
  intros x. unfold div2, mod2.
  destruct (div2_cert x) as [[k A]|[k A]] ; omega.
Qed.
```

Exercise 6.6.1 Prove $\forall x.\ \mathit{mod2}\ x \leq 1$.

Exercise 6.6.2 Prove the following fact about Skolem functions and sigma types.

```
Lemma Sigma_Skolem (X Y : Type) (p : X → Y → Prop) :
 (∀ x, {y | p x y}) → { f | ∀ x, p x (f x) }.
```

Exercise 6.6.3 Establish the following goal and explain why the opposite direction from an existential quantification to a sigma type cannot be established.

```
Goal ∀ X (p : X → Prop), {x | p x} → ∃ x, p x.
```

Exercise 6.6.4 Prove $\forall X : \mathit{Type}\ \forall p : X \to \mathit{Prop}.\ (\exists x.\ p\,x) \leftrightarrow \mathit{inhabited}\,\{\,x \mid p\,x\,\}$.

Exercise 6.6.5 There is a function that for every decidable predicate yields an equivalent boolean test. Prove the following goal to establish this fact.

```
Goal ∀ (X : Type) (p : X → Prop),  decidable p → {f : X → bool | ∀ x, p x ↔ f x = true}.
```

Exercise 6.6.6 Write a certifying function that divides its argument by 3.

```
Definition div3_cert (n : nat) : {k | n = 3*k} + {k | n = 3*k + 1} + {k | n = 3*k + 2}.
```

6.7 Strong Truth Value Semantics

There is a canonical injective embedding of *bool* into *Prop*:

```
Definition b2P (x : bool) : Prop := if x then True else False.
```

Proving the injectivity of *b2P* is straightforward. If we assume *TVS*, we can also prove that *b2P* is surjective. In Mathematics, an injective and surjective function $f : X \to Y$ always comes with an inverse function g such that $g(fx) = x$ and $f(gy) = y$ for all $x : X$ and $y : Y$. So it is natural to ask whether under *TVS* we can define the inverse of *b2p*. The answer is no.

It is, however, consistent to assume **strong truth value semantics**.

```
Definition STVS : Type := ∀ X : Prop, {X=True} + {X=False}.
```

Assuming *STVS* means assuming a function that for every proposition X yields a decision of type $\{X = True\} + \{X = False\}$. Clearly, *STVS* implies *TVS*.

```
Goal STVS → TVS.
Proof. intros stvs X. destruct (stvs X) ; subst X ; auto. Qed.
```

If we assume *STVS*, we can construct an inverse function for *b2p*.

```
Section STVS.
  Variable stvs : STVS.
  Definition P2b (X : Prop) : bool := if stvs X then true else false.
  Lemma P2bTrue : P2b True = true.
  Proof.
    unfold P2b. destruct (stvs True) as [A|A].
    + reflexivity.
    + exfalso. rewrite ← A. exact I.
  Qed.
  Lemma P2bFalse : P2b False = false.
  Proof.
    unfold P2b. destruct (stvs False) as [A|A].
    + exfalso. rewrite A. exact I.
    + reflexivity.
  Qed.
```

```
  Goal ∀ x : bool, P2b (b2P x) = x.
  Proof. intros [|] ; simpl. exact P2bTrue.  exact P2bFalse. Qed.
  Goal ∀ X : Prop, b2P (P2b X) = X.
  Proof.
    intros X. destruct (stvs X) ; subst X.
    - rewrite P2bTrue. reflexivity.
    - rewrite P2bFalse. reflexivity.
  Qed.
End STVS.
```

The names defined in a section remain defined after a section is closed. Their types are modified such that the variables of the section used in the definitions are taken as arguments. For instance:

```
Print P2b.
>  fun (stvs : STVS) (X : Prop) ⇒ if stvs X then true else false
>  : STVS → Prop → bool
```

Exercise 6.7.1 Prove that *b2P* is injective.

Exercise 6.7.2 Prove that *TVS* implies that *b2P* is surjective.

Exercise 6.7.3 Prove that *P2b* is injective.

```
Goal ∀ A : STVS, ∀ X Y : Prop,  P2b A X = P2b A Y → X = Y.
```

Exercise 6.7.4 Show that *STVS* implies that every proposition is decidable.

```
Goal STVS → ∀ X : Prop, dec X.
```

Exercise 6.7.5 Show that *STVS* implies that every predicate is decidable.

```
Goal STVS → ∀ (X : Type) (p : X → Prop), decidable p.
```

Coq Summary

New Tactics

constructor, *decide equality*.

New Inductive Types from the Standard Library

sumbool, *sumor*, *sum*, *sig*, *inhabited*.

Applying Tactics to Assumptions

To apply a tactic to an assumption *A*, end the tactic command with "*in A*". See the definition of *le_dec* in Section 6.3 for an example.

7 Inductive Predicates

An inductive definition introduces a type constructor together with a family of value constructors. If the type constructor yields a proposition, we speak of an inductive predicate and of proof constructors. We already know Coq's inductive predicates for conjunctions (*and*), disjunctions (*or*), and existential quantifications (*ex*) (see Chapter 2). Another inductive predicate we have introduced is *inhabited*.

In this chapter we will take a closer look at inductive predicates. Coq's facility for inductive definitions is extremely powerful and supports many advanced applications. The idea of inductive definitions originated with Peano's axioms (i.e., the inductive definition of *nat* with *O* and *S*) and developed further with proof systems for logical systems.

When we define an inductive predicate, we define a family of inductive propositions by specifying the syntax and the proof rules for the propositions. The inductive predicates *and*, *or* and *ex* give us a first idea of the flexibility of this approach. It turns out that we can go much further. Every recursively enumerable predicate can be defined as an inductive predicate in Coq. This is in contrast to computable functions, which are not necessarily definable in Coq.

7.1 Nonparametric Arguments and Linearization

We start our explanation of inductive predicates with an extreme case: A definition of a predicate on numbers that holds exactly for the number 0.

```
Inductive zero : nat → Prop :=
| zeroI : zero 0.
```

The definition provides exactly one proof *zeroI*, which proves the proposition *zero 0*. The propositions *zero 1*, *zero 2*, *zero 3*, and so forth are all unprovable. We characterize the inductive predicate *zero* as follows.

```
Lemma zero_iff x :
  zero x ↔ x = 0.
```

```
Proof.
  split ; intros A.
  – destruct A. reflexivity.
  – subst x. constructor.
Qed.
```

The interesting step of the proof is *destruct A*, which does a case analysis on the proof of *zero x*. Since there is only a single proof constructor *zeroI* : *zero 0*, the case analysis yields a single subgoal where the variable x is instantiated to 0.

The argument of the inductive predicate *zero* is called **nonparametric** since it is instantiated by the proof constructor *zeroI* : *zero 0*. This is the first time we see an inductive predicate with a nonparametric argument. Check the definitions of the inductive predicates *and*, *or*, *ex*, and *inhabited* to see that all arguments of these predicates are parametric.

There is an important technicality one has to know about nonparametric arguments: When the tactics *destruct* and *induction* are applied to an inductive assumption $A : ct_1 \dots t_n$, the terms t_i for the nonparametric arguments of c must be variables not appearing in the other terms. We say that inductive assumptions must be **linear** when they are used with *destruct* and *induction*. Coq offers the tactic *remember* to **linearize** inductive assumptions.

```
Goal ¬ zero 2.
Proof. intros A. remember 2 as x. destruct A. discriminate Heqx. Qed.
```

Exercise 7.1.1 Make sure you can prove the propositions *zero 0*, $\neg$*zero 7*, and $\forall x.\ \neg zero\,(S\,x)$ without using lemmas.

Exercise 7.1.2 Prove that the predicate *zero* is decidable.

Exercise 7.1.3 Prove the following lemma.

```
Lemma remember (X : Type) (p : X → Type) (x : X) :
  (∀ y, y = x → p y) → p x.
```

Try to understand why the lemma justifies the tactic *remember*. Use the lemma and the tactic *pattern* to prove the proposition $\forall x.\ \neg zero\ (S\ x)$.

Exercise 7.1.4 Prove the following impredicative characterization of *zero*.

```
Goal ∀ x, zero x ↔ ∀ p : nat → Prop, p 0 → p x.
```

Exercise 7.1.5 Define a boolean test *zerob* : *nat* → *bool* and prove the correctness condition $\forall x.\ zero\ x \leftrightarrow zerob\ x = true$.

Exercise 7.1.6 Define an inductive predicate $leo : nat \to Prop$ with two proof constructors $leo0 : leo\ 0$ and $leo1 : leo\ 1$.

a) Prove $\forall x,\ leo\ x \leftrightarrow x \leq 1$.

b) Characterize *leo* impredicatively and prove the correctness.

c) Characterize *leo* with a boolean test $leob : nat \to bool$ and prove the correctness of the characterization.

7.2 Even

Our next example is an inductive predicate *even* that holds exactly for the even numbers. This time we use two proof rules, one for *even 0* and one for *even* $(S\,(S\,x))$.

$$\frac{}{even\ 0} \qquad\qquad \frac{even\ x}{even\ (S(S\ x))}$$

The two proof rules can be expressed with two proof constructors:

$evenO\ :\ even\ 0$

$evenS\ :\ \forall x{:}nat.\ even\ x \to even\ (S\,(S\,x))$

From the types of the proof constructors it is clear that the argument of *even* is nonparametric. We now introduce the predicate *even* and the proof constructors *evenO* and *evenS* with a single inductive definition.

```
Inductive even : nat → Prop :=
| evenO :  even 0
| evenS x :  even x → even (S (S x)).
```

The type of the constructor *evenS* is specified with a notational convenience we have seen before in the statement of lemmas. The convenience makes it possible to specify argument variables of a constructor without types, leaving it to Coq to infer the types.

We prove a lemma characterizing *even* non-inductively.

```
Lemma even_iff x :
  even x ↔ ∃ k, x = 2*k.
```

```
Proof.
  split ; intros A.
  - induction A.
    + ∃ 0. reflexivity.
    + destruct IHA as [k IHA]. subst x. ∃ (S k). simpl. omega.
  - destruct A as [k A]. subst x. induction k ; simpl.
    + constructor.
    + replace (S(k+S(k+0))) with (S (S (2*k))) by omega.
      constructor. exact IHk.
Qed.
```

Both directions of the proof deserve careful study. The direction from left to right is by induction on the proof $A : even\ x$. The induction does a case analysis for the two proof constructors of *even*. In each case the nonparametric argument x is instantiated as specified by the type constructor. For *evenS* we get $x = S(S\,x')$ and the inductive hypothesis $IHA : \exists k.\ x' = 2 * k$.[1]

The direction from right to left first eliminates the existential quantification for k and then proves the so obtained claim by induction on $k : nat$. The induction step uses the tactic *replace* to rewrite with the equation $S(k + S(k + 0)) = S(S(2 * k))$, which is established by the tactic *omega*. This is the first time we use the tactic *replace*. If the annotation *by omega* is omitted, *replace* will introduce an extra subgoal to establish the equation.

The next two lemmas prove simple facts about *even* using case analysis on proofs of propositions obtained with *even*. In each case the linearization of the inductive assumption with the tactic *remember* is essential.

```
Goal ¬ even 3.
Proof.
  intros A. remember 3 as x. destruct A.
  - discriminate Heqx.
  - destruct A ; discriminate Heqx.
Qed.

Lemma even_descent x :
  even (S (S x)) → even x.
Proof.
  intros A. remember (S (S x)) as y.
  destruct A as [|y A].
  - discriminate Heqy.
  - congruence.
Qed.
```

[1] The proof script reuses the variable x for x'. You can get the variable x' by annotating the induction command with *as* $[|x'\ A']$.

Exercise 7.2.1 Prove *even 6* and ¬*even 5* without using lemmas.

Exercise 7.2.2 Prove the following goals without using lemmas.

(a) **Goal** ∀ x y, even x → even y → even (x+y).

(b) **Goal** ∀ x y, even x → even (x+y) → even y.

(c) **Goal** ∀ x, even x → even (S x) → False.

Exercise 7.2.3 Prove the so-called **inversion lemma** for *even*.

Lemma even_inv x : even x → x = 0 ∨ ∃ x', x = S (S x') ∧ even x'.

Exercise 7.2.4 Prove the following impredicative characterization of evenness.

Goal ∀ x, even x ↔ ∀ p : nat → Prop, p 0 → (∀ y, p y → p (S (S y))) → p x.

Exercise 7.2.5 Some proofs need ideas. Try to prove $\forall x,\ \neg even\ x \to even\ (S\ x)$. As is, the induction on $x : nat$ will not go through. The problem is that the induction on $x : nat$ takes away a single S while the constructor *evenS* takes away two $S's$. The standard cure consists in generalizing the claim so that the inductive hypothesis becomes strong enough. Convince yourself that the proof of the following lemma generalizing the claim is doable.

Lemma even_succ x : (¬ even x → even (S x)) ∧ (¬ even (S x) → even x).

Hint: The apply tactic can be used with a proof of a conjunction of implications. In this case *apply* attempts to apply one of the implications.

Exercise 7.2.6 Prove $\forall x,\ even\ x \leftrightarrow \neg even\ (S\ x)$. Hint: Use the lemma *even_succ* from Exercise 7.2.5.

Exercise 7.2.7 Prove that the predicate *even* is decidable. Hint: Use the lemma *even_succ* from Exercise 7.2.5.

Exercise 7.2.8 Here is an inductive definition of an evenness predicate with a parametric argument.

```
Inductive even' (x : nat) : Prop :=
| even'O : x=0 → even' x
| even'S y : even' y → x = S (S y) → even' x.
```

a) Prove $\neg even'\ 3$.

b) Prove $\forall x.\ even'\ x \leftrightarrow even\ x$.

Exercise 7.2.9 Here is a boolean test for evenness.

```
Fixpoint evenb (x : nat) : bool :=
  match x with
    | 0 ⇒ true
    | S (S x') ⇒ evenb x'
    | _ ⇒ false
  end.
```

Try to prove $\forall x.\ even\ x \leftrightarrow evenb\ x = true$. The direction from left to right is a straightforward induction on a proof of *even x*. The direction from right to left is problematic since an induction on $x : nat$ takes away one S while the constructor *evenS* takes away two S's. Proving the following more general claim solves the problem.

```
Lemma evenb_even x :  (evenb x = true → even x) ∧ (evenb (S x) = true → even (S x)).
```

7.3 Less or Equal

Coq defines the order predicate "≤" for natural numbers inductively based on the following proof rules.[2]

$$\frac{}{x \le x} \qquad\qquad \frac{x \le y}{x \le S\,y}$$

The exact definition is

```
Inductive le (x : nat) : nat → Prop :=
| le_n : le x x
| le_S y : le  x y → le x (S y).

Notation "x ≤ y" := (le x y) (at level 70).
```

Note that the first argument of the inductive predicate *le* is parametric and that the second argument is nonparametric. We will always write inductive definitions such that all parametric arguments appear as **parameters** in the head of the inductive definition. Note that *le* is the first inductive predicate we see having both parametric and nonparametric arguments.

To get familiar with *le*, we prove a lemma characterizing *le* non-inductively.

```
Lemma le_iff x y :
  x ≤ y ↔ ∃ k, k + x = y.
```

[2] Use the commands *Locate* and *Print* to see Coq's definition of "≤" for *nat*.

```
Proof.
  split.
  - intros A. induction A as [|y A].
    + ∃ 0. reflexivity.
    + destruct IHA as [k B]. ∃ (S k). simpl. congruence.
  - intros [k A]. subst y. induction k ; simpl.
    + constructor.
    + constructor. exact IHk.
Qed.
```

The proof deserves careful study. The direction from left to right is by induction on a proof of $x \leq y$. The other direction is by induction on k.

Next we write an informative test for *le*. This takes some preparation. We leave the proofs of the first three lemmas as exercises.

```
Lemma le_O x : 0 ≤ x.

Lemma le_SS x y : x ≤ y → S x ≤ S y.

Lemma le_Strans x y : S x ≤ y → x ≤ y.

Lemma le_zero x :
  x ≤ 0 → x = 0.

Proof.
  intros A. remember 0 as y. destruct A as [|y A].
  - reflexivity.
  - discriminate Heqy.
Qed.

Lemma le_SS' x y :
  S x ≤ S y → x ≤ y.

Proof.
  intros A. remember (S y) as y'. induction A as [|y' A].
  - injection Heqy'. intros A. subst y. constructor.
  - injection Heqy'. intros B. subst y'. apply le_Strans, A.
Qed.

Definition le_dec x y : dec (x ≤ y).
  revert y. induction x ; intros y.
  - left. apply le_O.
  - destruct y.
    + right. intros A. apply le_zero in A. discriminate A.
    + destruct (IHx y) as [A|A].
      * left. apply le_SS, A.
      * right. intros B. apply A, le_SS', B.
Defined.
```

Exercise 7.3.1 Prove the lemmas *le_O*, *le_SS*, and *le_Strans* without using *omega*. Hints: Lemma *le_O* follows by induction on x. Lemmas *le_SS* and *le_Strans* follow by induction for *le*.

Exercise 7.3.2 Prove the inversion lemma for *le*.

```
Lemma le_inv x y :  x ≤ y → x = y ∨ ∃ y', y = S y' ∧ x ≤ y'.
```

Exercise 7.3.3 Prove that *le* is transitive. Do not use *omega*.

```
Lemma le_trans x y z :  x ≤ y → y ≤ z → x ≤ z.
```

Hint: Do the proof by induction for $y \leq z$.

Exercise 7.3.4 Prove the following goal not using *omega*.

```
Goal ∀ x y, S x ≤ y → x ≠ y.
```

Hint: Proceed by induction on y and use the lemmas *le_zero* and *le_SS'*.

Exercise 7.3.5 Prove that *le* is anti-symmetric. Do not use *omega*.

```
Goal ∀ x y,  x ≤ y → y ≤ x → x=y.
```

Hint: Proceed by induction on x and use *le_zero*, *le_Strans*, and *le_SS'*.

Exercise 7.3.6 Prove that *le* and the boolean test *leb* from the standard library agree. Do not use omega.

```
Goal ∀ x y, x ≤ y ↔ leb x y = true.
```

Hint: For the direction from left to right you will need two straightforward lemmas for *leb*. For the other directions use the lemmas *le_O* and *le_SS*.

7.4 Equality

Coq defines equality inductively.

```
Inductive eq (X : Type) (x : X) : X → Prop :=
| eq_refl : eq x x.

Notation "x = y" := (eq x y) (at level 70).
```

Note that the first two arguments of the inductive predicate *eq* are parametric and that the third argument is nonparametric. It is easy to establish the Leibniz characterization of equality.

```
Lemma Leibniz (X : Type) (x y : X) :
  x = y ↔ ∀ p : X → Prop, p x → p y.
```

```
Proof.
  split ; intros A.
  - destruct A. auto.
  - apply (A (fun z ⇒ x = z)). constructor.
Qed.
```

7.5 Exceptions to the Elim Restriction

The exceptions to the elim restriction can be stated as follows: If an inductive predicate has at most one proof constructor and the nonparametric arguments of the proof constructor are all proofs, then the elim restriction does not apply to matches for this predicate.

An interesting exception to the elim restriction is the inductive predicate *eq* whose single proof constructor

$$eq\_refl \ : \ \forall X : Type \ \forall x : X. \ eq \ x \ x$$

has only parametric arguments (i.e., arguments fixed in the head of the inductive definition of *eq*). Thus the elim restriction does not apply to matches on equality proofs. This provides for the definition of the following casting function.

```
Definition cast (X : Type) (x y : X) (f : X → Type) : x = y → f x → f y.
  intros A B. destruct A. exact B.
Defined.
```

The function *cast* gives us a function that given a proof of $x = y$ converts from type $f\,x$ to type $f\,y$. Here is an example for the use of *cast*.

```
Definition fin (n : nat) : Type := nat_iter n option False.

Goal ∀ n, fin n → fin (n+0).

Proof. intros n. apply cast. omega. Qed.
```

Note that the cast is needed since the terms *fin n* and *fin* $(n + 0)$ are not convertible.

Exercise 7.5.1 Prove the following goal. Explain why the elim restriction does not apply to conjunctions.

```
Goal ∀ X Y : Prop, X ∧ Y → prod X Y.
```

Exercise 7.5.2 Explain why the elim restriction applies to matches for the inductive predicate *inhabited*.

Exercise 7.5.3 Complete the following definition. Explain why your definition exploits an exception to the elim restriction.

```
Definition exfalso : False → ∀ X : Type, X := ...
```

Exercise 7.5.4 Prove the following goal.

```
Goal ∀ (X : Type) (x y : X), (∀ p : X → Prop, p x → p y) → ∀ p : X → Type, p x → p y.
```

Note that goal is formulated without making use of inductive types. Yet it can only be proven using inductive types.

7.6 Safe and Nonuniform Parameters

Our final example is an inductive predicate $safe : (nat \to Prop) \to nat \to Prop$ such that $safe\ p\ n$ holds if and only if p holds for some $k \geq n$. We base the inductive definition on the following rules.

$$\frac{p\ n}{safe\ p\ n} \qquad \frac{safe\ p\ (S\ n)}{safe\ p\ n}$$

Defining *safe* in Coq is straightforward.

```
Inductive safe (p : nat → Prop) (n : nat) : Prop :=
| safeB : p n → safe p n
| safeS : safe p (S n) → safe p n.
```

One reason for considering *safe* is that it has both a **uniform** and a **nonuniform** parameter.[3] The parameter p is uniform since it is not instantiated in the types of the proof constructors *safeB* and *safeS*. The parameter n is nonuniform since it is instantiated to $S\ n$ in the type of the constructor *safeS*. The argument n does not qualify as a nonparametric argument of *safe* since the instantiation appears in argument position rather than in result position. This is the first time we encounter a type constructor with a nonuniform parameter.

When we use the tactic *induction* on a proof of a proposition obtained with an inductive predicate, both the nonuniform parametric arguments and the nonparametric arguments of the proposition must be linearized. If we use the tactic *destruct*, it suffices if the nonparametric arguments are linearized. For instance, if we have an assumption $A : safe\ p\ 0$, *destruct* can be applied to A but *induction* must not be applied to A.

We prove that $safe\ p$ is downward closed.

```
Lemma safe_dclosed k n p :
  k ≤ n → safe p n → safe p k.
```

[3] A parameter is a parametric argument.

```
Proof.
  intros A B. induction A as [|n A].
  - exact B.
  - apply IHA. right. exact B.
Qed.
```

The proof is by induction on a proof of $k \leq n$. Note the use of the tactic *right* to apply the second constructor of *safe*. The tactics *left* and *right* can be used with every type constructor that has two value constructors.

We prove that *safe* satisfies its specification.

```
Lemma safe_iff p n :
  safe p n ↔ ∃ k, n ≤ k ∧ p k.
Proof.
  split ; intros A.
  - induction A as [n B|n A].
    + ∃ n. auto.
    + destruct IHA as [k [B C]].
      ∃ k. split. omega. exact C.
  - destruct A as [k [A B]].
    apply (safe_dclosed A). left. exact B.
Qed.
```

The direction from left to right is by induction on a proof of *safe p n*. From the destructuring pattern for the induction we learn that a name for the nonuniform parameter n must be given for both subgoals. This must be done for nonuniform parameters in general.

The direction from right to left follows with the lemma *safe_dclosed*. Using this lemma is essential since a direct proof of the more specific claim we have at this point seems impossible.

The predicate *safe* is different from the other inductive predicates we saw in this chapter in that it is impossible to express it with a boolean test. This is the case even if we assume that the argument p is a decidable predicate.[4]

Exercise 7.6.1 We define a predicate *least* such that *least p n k* holds if and only if k is the least number such that $n \leq k$ and $p\ k$ holds.

```
Inductive least (p : nat → Prop) (n : nat) : nat → Prop :=
| leastB : p n → least p n n
| leastS k : ¬ p n → least p (S n) k → least p n k.
```

Note that the first argument of *least* is a uniform parameter, the second argument is a nonuniform parameter, and the third argument is nonparametric. Prove the following correctness lemmas for *least*.

[4] Think of pn as the statement saying that a particular Turing machine halts on a particular input in at most n steps.

```
Lemma least_correct1 p n k : least p n k → p k.
Lemma least_correct2 p n k : least p n k → n ≤ k.
Lemma least_correct3 p n k : least p n k → ∀ k', n ≤ k' → p k' → k ≤ k'.
Lemma least_correct4 p n k : (∀ x, dec (p x)) → p (n+k) → ∃ k', least p n k'.
Lemma least_correct p n k (p_dec : ∀ x, dec (p x)) :
  least p n k  ↔  p k ∧ n ≤ k ∧ ∀ k', n ≤ k' → p k' → k ≤ k'.
```

Hint: Use *le_lt_eq_dec* from the standard library for the proof of *least_correct3*.

7.7 Constructive Choice for Nat

We will now construct a function

```
cc_nat : ∀ p : nat → Prop, (∀ x, dec (p x)) → (∃ x, p x) → {x | p x}
```

we call *constructive choice for nat*. For a decidable predicate p on numbers constructive choice yields a function that for every proof of an existential quantification $\exists x.px$ yields a value of the sigma type $\{x \mid px\}$. Thus *cc_nat* bypasses the elim restriction for existential quantifications of decidable predicates on numbers. We will obtain this remarkable result with a new proof technique based on the inductive predicate *safe* from the last section.

For convenience, we declare a decidable predicate p in a section.

```
Section First.
Variable p : nat → Prop.
Variable p_dec : ∀ n, dec (p n).
```

We now write a function *first* that from a proof of *safe p n* obtains a value of $\{k \mid pk\}$. The function *first* is the cornerstone of the construction of *cc_nat*. Clearly, *first* overcomes the elim restriction. We define *first* by recursion on the given proof of *safe p n*.

```
Fixpoint first (n : nat) (A : safe p n) : {k | p k} :=
  match p_dec n with
    | left  B ⇒ exist p n B
    | right B ⇒ first match A with
                       | safeB C ⇒ match B C with end
                       | safeS A' ⇒ A'
                     end
  end.
```

Given that *first* computes by recursion on A, one would expect that *first* first matches on A. However, this is impossible because of the elim restriction. So we first match on *p_dec n*. If we obtain a proof of *p n*, we are done. Otherwise, we recurse on a proof of *safe p* $(S\,n)$ we obtain by matching on the proof A of

safe p n. This time the elim restriction does not apply since we are constructing a proof. We obtain two cases. The case for *safeS* is straightforward since we get a proof of *safe p* (*S n*) by taking off the constructor. The case for *safeB* yields a proof *C* of *p n*. Since we have a proof *B* of $\neg pn$, we can match on the proof *B C* of *False*. Now we are done since each rule of the match returns a proof of *safe p* (*S n*) that is obtained by taking off a constructor of the proof *A* (vacuous reasoning).

The recursion scheme underlying *first* is nonstandard. The standard recursion scheme would first match on the proof and than recurse. The recursion scheme we see with *first* first recurses and only then matches on the proof. This way the elim restriction can be bypassed. We speak of an **eager proof term recursion**.

It is now straightforward to construct the certifying function *cc_nat*. We obtain the result by applying *first* to a proof of *safe p 0*. The proof of *safe p 0* we obtain with the lemma *safe_dclosed* from a proof of *safe p n* for some *n*. The *n* and the proof of *safe p n* we obtain from the given proof of $\exists x.\, px$.

```
Lemma cc_nat : (∃ x, p x) → {x | p x}.

Proof.
  intros A. apply first with (n:=0).
  destruct A as [n A].
  apply safe_dclosed with (n:=n). omega. left. exact A.
Qed.
```

Note the "with" annotations used with the tactic *apply*. They provide a convenient means for specifying implicit arguments of the function being applied.

There is a straightforward algorithmic idea underlying *cc_nat* we may call linear search: To find the least $k \geq n$ such that pn, increment *n* until pn holds. What is interesting about linear search from our perspective is that linear search is not structurally recursive and that it may not always terminate. We can see *first* as a logical reformulation of linear search that is structurally recursive.

Exercise 7.7.1 Write a constructive choice function for *bool*.

```
Definition cc_bool (p : bool → Prop) (p_dec : ∀ x, dec (p x)) : (∃ x, p x) → {x | p x}.
```

Exercise 7.7.2 Complete the definitions of the following recursive and certifying functions with scripts. Assume a section declaring a decidable predicate *p*. Follow the eager proof term recursion scheme from *first*.

```
Fixpoint first1 (n : nat) (A : safe p n) : {k | p k ∧ k ≥ n}.

Fixpoint first2 (n : nat) (A : safe p n) : {k | p k ∧ k ≥ n ∧ ∀ k', n ≤ k' → p k' → k ≤ k'}.
```

Hint: First redefine *first* with a script. Check the partial proof terms you obtain with the command *Show Proof*. Then refine the script for *first* to obtain the script for *first1*.

Exercise 7.7.3 Write constructive choice functions for the finite types *fin n*.

```
Definition cc_fin (n : nat) (p : fin n → Prop) (p_dec : ∀ x, dec (p x))
  : (∃ x, p x) → {x | p x}.
```

7.8 Technical Summary

An inductive definition introduces a family of typed names called constructors. One of the constructors yields types and is called type constructor. The remaining constructors are called value constructors and yield the elements of the types obtainable with the type constructor. An inductive value is a value obtained with a constructor. Thus an inductive predicate is a predicate obtained with a constructor, a proof constructor is a value constructor yielding a proof, and inductive proposition is a proposition obtained with a type constructor.

An inductive definition comes with a list of named parameters specified in the head of the definition. The parameters appear as leading arguments of every constructor introduced by the inductive definition. We speak of the parametric arguments of a constructor. The constructors may have additional arguments, which we call nonparametric arguments. There is the constraint that the result type of a value constructor must not instantiate parametric arguments of the type constructor. The parametric arguments of a value constructor do not appear in matches and the type specification of the constructor in the introducing inductive definition.

As example we consider the following inductive definition.

```
Inductive least (p : nat → Prop) (n : nat) : nat → Prop :=
| leastB : p n → least p n n
| leastS k : ¬ p n → least p (S n) k → least p n k.
```

The definition introduces the constructors

$$\begin{aligned} \mathit{least} &: (\mathit{nat} \to \mathit{Prop}) \to \mathit{nat} \to \mathit{nat} \to \mathit{Prop} \\ \mathit{leastB} &: \forall p{:}\mathit{nat} \to \mathit{Prop}\ \forall n{:}\mathit{nat}.\ p\,n \to \mathit{least}\ p\ n\ n \\ \mathit{leastS} &: \forall p{:}\mathit{nat} \to \mathit{Prop}\ \forall n{:}\mathit{nat}\ \forall k{:}\mathit{nat}.\ \neg p\,n \to \mathit{least}\ p\ (S\,n)\ k \to \mathit{least}\ p\ n\ k \end{aligned}$$

The leading two arguments of each constructor are parametric, the remaining arguments are nonparametric. The type constructor *least* and the value constructor *leastB* have one nonparametric argument each, and the value constructor *leastS* has three nonparametric arguments.

Our inductive definitions will always be such that for every nonparametric argument of the type constructor there will be at least one value constructor that instantiates this argument in its result type. Coq does not enforce this condition.

The elim restriction applies to matches on proofs of inductive propositions where the underlying inductive definition either has more than one proof constructor or has a single proof constructor taking a nonparametric argument specified with a proper type. For instance, the elim restriction applies to matches on proofs of disjunctions and existential quantifications, but it does not apply to matches on proofs of equations and conjunctions.

Coq distinguishes between uniform and nonuniform parameters of inductive definitions. A parameter of an inductive definition is nonuniform if it is instantiated in argument position in the type specification of a value constructor. For instance, the inductive definition *least* has the uniform parameter p and the nonuniform parameter n. The nonuniformity of n is due to the type of the third nonparametric argument of the value constructor *leastS*.

When we apply the tactic *destruct* to a proof A of an inductive proposition $Ct_1 \ldots t_n$, all terms t_i giving nonparametric arguments must be variables that do not appear in the other terms. Similarly, when we apply the tactic *induction* to a proof A of an inductive proposition $Ct_1 \ldots t_n$, all terms t_i giving nonparametric or nonuniform parametric arguments must be variables that do not appear in the other terms. We say that inductive propositions are linear if they satisfy this condition. Inductive propositions can be linearized with the tactic *remember*.[5]

7.9 Induction Lemmas

When we apply the tactic *induction* to an assumed value of an inductive type, the induction lemma for the underlying type constructor is applied. To have an example, we consider the inductive definition of *even*.

```
Inductive even : nat → Prop :=
| evenO : even 0
| evenS x : even x → even (S (S x)).
```

[5] Unfortunately, the tactics *destruct* and *induction* do not give warnings when they are applied to proofs of nonlinear inductive propositions. Instead, they linearize the proposition automatically and forget the equations relating the fresh variables with the moved away argument terms. This often leads to unprovable subgoals.

The type of the induction lemma *even_ind* Coq derives for *even* is as follows.

$$\forall p : nat \to Prop.$$
$$p\,0 \to$$
$$(\forall x : nat.\ even\ x \to p\ x \to p(S(S\ x))) \to$$
$$\forall x : nat.\ even\ x \to p\ x$$

Note that each constructor contributes a premise of the implication. When we apply the tactic *induction* to an assumption $A : even\ x$, the goal is rearranged by moving all assumptions depending on the variable x to the claim. Thus these assumptions become part of the induction predicate p and hence appear in the inductive hypothesis $p\,x$ of the premise for the constructor *evenS*.

Our second example is the inductive definition of *le*.

```
Inductive le (x : nat) : nat → Prop :=
| le_n : le x x
| le_S y : le x y → le x (S y).
```

The induction lemma *le_ind* Coq generates for *le* quantifies the uniform parameter x at the outside.

$$\forall x : nat\ \forall p : nat \to Prop.$$
$$p\ x \to$$
$$(\forall\ y : nat.\ le\ x\ y \to p\ y \to p(S\ y)) \to$$
$$\forall y : nat.\ le\ x\ y \to p\ y$$

If you look at the induction lemma Coq generates for *least*, you will see that the nonuniform parameter is treated like the nonparametric argument in that it appears as an argument of the induction predicate p.

When we work with paper and pencil, doing inductive proofs based on inductive definitions requires considerable training and great care. When we work with Coq, the tedious details are taken care of automatically and proof correctness is guaranteed.

Exercise 7.9.1 Complete the following definitions of the induction lemmas for *even* and *le*.

```
Definition even_ind' (p : nat → Prop) (r1 : p 0) (r2 : ∀ x, even x → p x → p (S (S x)))
  : ∀ x, even x → p x := ··· .
Definition le_ind' (x : nat) (p : nat → Prop) (r1 : p x) (r2 : ∀ y, le x y → p y → p (S y))
  : ∀ y, le x y → p y := ··· .
```

Coq Summary

New Tactics

remember, *replace*.

Automation Tactic *inversion*

The automation tactic *inversion* subsumes the capabilities of the tactics *destruct*, *discriminate*, and *injection*. The use of *inversion* is convenient if it solves the goal. Otherwise *inversion* often creates subgoals with many equational assumptions. We will use *inversion* only if it solves the goal. Here are two examples.

```
Goal ¬ even 1. Proof. intros A. inversion A. Qed.
```

```
Goal ¬ 7 ≤ 0. Proof. intros A. inversion A. Qed.
```

With Annotations for *apply*

With annotations used with the tactic *apply* are a convenient means for specifying implicit arguments of the function being applied. See the definition of *cc_nat* in Section 7.7.

8 Lists and Finite Sets

Type theory does not come with sets. However, every list represents a finite set. In this chapter we develop the basic theory of lists representing finite sets.

We study membership in and inclusion and equivalence of lists. We then study duplicate-free lists and cardinality of lists.

For many results about lists, decidability properties play an important role. Membership, inclusion, equivalence, and duplicate freeness of lists are decidable provided the base type comes with decidable equality. Moreover, quantification over lists preserves decidability.

To establish the results about lists, we will use and explain several advanced features of Coq:

- Assumption management with sections.
- Automatic resolution of decidability conditions with type class inference.
- Setoid rewriting with list inclusions and list equivalences.
- New features of the tactics *apply*, *destruct*, and *assert*.
- Hint commands strengthening the auto tactic.
- The automation tactics *eauto* and *firstorder*.

In fact, the larger part of this chapter is concerned with new Coq features.

The development of this chapter will be used in later chapters. Ideally, most of the definitions and lemmas we study in this chapter should be in Coq's standard library. This is not the case. This chapter can serve as a case study of what it takes to develop in Coq the basic theory of a basic data structure. We will develop the theory of lists and finite sets further in a later chapter.

8.1 List Membership

The connection between lists and sets is made by membership. Here is an inductive characterization of membership in lists.

$$\frac{}{x \in x :: A} \qquad \frac{x \in A}{x \in y :: A}$$

$x \in x :: A$	*eq
$x \in A \to x \in y :: A$	*cons
$x \in A \vee x \in B \to x \in A +\!\!+ B$	*or_app
$x \notin nil$	*nil
$x \in [y] \to x = y$	in_sing
$x \in y :: A \to x \neq y \to x \in A$	cons_neq
$x \in A +\!\!+ B \leftrightarrow x \in A \vee x \in B$	app_iff
$x \in map\ f\ A \leftrightarrow \exists y.\ f\,y = x \wedge y \in A$	map_iff

Figure 8.1: Membership laws for lists

A characterization of list membership with concatenation looks as follows.

$$x \in A \ \leftrightarrow\ \exists A_1\ A_2.\ A = A_1 +\!\!+ [x] +\!\!+ A_2$$

We can also characterize list membership recursively.

$$\begin{aligned}(x \in nil) &= False\\ (x \in y :: A) &= (x = y \vee x \in A)\end{aligned}$$

A list **represents a set** if the members of the list are exactly the members of the set. Thus every set representable by a list is finite and all its members belong to a common type. Since lists are ordered and can contain duplicates, different lists may represent the same set. For instance, the lists $[1;2]$, $[2;1]$, and $[1;2;2]$ are three different representations of the set $\{1,2\}$. Even the empty set has different list representations, since every list carries the type of its elements. For instance, the lists *@nil bool* and *@nil nat* both represent the empty set. We say that a list carries more information than the set of its members.

Figure 8.1 shows the most important membership laws for lists. The stared laws are registered with *auto* (explained below).

We load Coq's standard module for lists and activate the list notations.

```
Require Import List.
Export ListNotations.
```

The standard module *List* defines list membership with a recursive predicate:

```
Fixpoint In (X : Type) (x : X) (A : list X) : Prop :=
  match A with
    | nil ⇒ False
    | y::A' ⇒ y=x ∨ In x A'
  end.
```

We define an infix operator for the membership predicate:

```
Notation "x 'el' A" := (In x A) (at level 70).
```

We will always write "$\in$" for the infix operator "*el*". Coq's library provides the following lemmas for list membership.

```
Lemma in_eq X (x : X) A :
  x ∈ x :: A.

Lemma in_cons X (x y : X) A :
  x ∈ A → x ∈ y :: A.

Lemma in_nil X (x : X) :
  ¬ x ∈ nil.

Lemma in_or_app X (x : X) A B :
  x ∈ A ∨ x ∈ B → x ∈ A ++ B.
```

We use the command *Hint Resolve* to enhance Coq's auto tactic with the lemmas.

```
Hint Resolve in_eq in_cons in_nil in_or_app.
```

We now establish the characterization of list membership with concatenation.

```
Lemma in_iff X (x : X) A :
  x ∈ A ↔ ∃ B C, A = B ++ [x] ++ C.

Proof.
  induction A as [|y A] ; simpl ; split.
  - intros [].
  - intros [B [C D]]. destruct B ; discriminate D.
  - intros [D|D].
    + subst y. ∃ nil, A. reflexivity.
    + apply IHA in D as [B [C D]]. ∃ (y::B), C. simpl. f_equal. exact D.
  - intros [B [C D]]. destruct B as [|z B].
    + injection D. auto.
    + injection D. intros E _. right. subst A. auto.
Qed.
```

The proof is by induction on A and introduces four subgoals by splitting the equivalence. Note the application of *IHA* to the assumption *D* in the third subgoal where the apply command is annotated with a destructuring pattern, which immediately destructures the object obtained by the application. Also consider the use of *auto* in the fourth subgoal, which automatically applies the lemmas *in_or_app* and *in_eq* we registered with the command *Hint Resolve*.

Two lists are **disjoint** if they don't have a common element.

```
Definition disjoint (X : Type) (A B : list X) :=
  ¬ ∃ x, x ∈ A ∧ x ∈ B.
```

We prove a lemma.

```
Lemma disjoint_cons X (x : X) A B :
  disjoint (x::A) B ↔ ¬ x ∈ B ∧ disjoint A B.
Proof.
  split.
  - intros D. split.
    + intros E. apply D. eauto.
    + intros [y [E F]]. apply D. eauto.
  - intros [D E] [y [[F|F] G]].
    + congruence.
    + apply E. eauto.
Qed.
```

Note the use of the automation tactic *eauto*, a version of *auto* that attempts to find the witnesses of existential quantifications. Also note the deeply nested destructuring pattern used for the direction from right to left.

Exercise 8.1.1 Prove the following lemmas.

```
Lemma in_sing X (x y : X) : x ∈ [y] → x=y.
Lemma in_cons_neq X (x y : X) A : x ∈ y::A → x ≠ y → x ∈ A.
```

Exercise 8.1.2 Prove the following lemmas provided by the module *List*.

```
Lemma in_app_iff X (x : X) A B : x ∈ A++B ↔ x ∈ A ∨ x ∈ B.
Lemma in_map_iff X Y (f : X → Y) A y : y ∈ map f A ↔ ∃ x, f x = y ∧ x ∈ A.
Lemma in_map X Y (f : X → Y) A x : x ∈ A → f x ∈ map f A.
```

Exercise 8.1.3 Prove the following characterization of list membership.

```
Lemma in_iff X (x : X) A : x ∈ A ↔ ∃ y A', A = y::A' ∧ (y=x ∨ x ∈ A').
```

Exercise 8.1.4 Prove the following characterization of disjointness.

```
Lemma disjoint_forall X (A B : list X) : disjoint A B ↔ ∀ x, x ∈ A → ¬ x ∈ B.
```

Exercise 8.1.5 Define an inductive predicate *mem* for list membership and prove

```
Lemma mem_iff X (x : X) A : mem x A ↔ x ∈ A.
```

$A \subseteq A$	*refl
$A \subseteq B \to A \subseteq x :: B$	*tl
$x \in B \to A \subseteq B \to x :: A \subseteq B$	*cons
$A \subseteq B \to A \subseteq B +\!\!+ C$	*appl
$A \subseteq C \to A \subseteq B +\!\!+ C$	*appr
$A \subseteq C \to B \subseteq C \to A +\!\!+ B \subseteq C$	*app
$nil \subseteq A$	*nil
$A \subseteq nil \to A = nil$	nil_eq
$A \subseteq B \to x :: A \subseteq x :: B$	shift
$x :: A \subseteq B \leftrightarrow x \in B \wedge A \subseteq B$	lcons
$x :: A \subseteq x :: B \to x \notin A \to A \subseteq B$	lrcons

Figure 8.2: Inclusion laws for lists

8.2 List Inclusion

List membership gives us list inclusion:

$$A \subseteq B \;\leftrightarrow\; \forall x.\, x \in A \to x \in B$$

If a list A is included in a list B, the set represented by A is a subset of the set represented by B.

Figure 8.2 shows the most important inclusion laws for lists. The first six laws are provided by the standard module *List* as lemmas. We register the lemmas with *auto*.

```
Hint Resolve incl_refl incl_tl incl_cons incl_appl incl_appr incl_app.
```

Coq's library defines list inclusion as follows.

```
Definition incl (X : Type) (A B : list X) : Prop :=
  ∀ x, x ∈ A → x ∈ B.

Hint Unfold incl.
```

The *Hint Unfold* command tells the auto tactic to automatically unfold the definition of *incl*. We define an infix operator for the inclusion predicate.

```
Notation "A <<= B" := (incl A B) (at level 70).
```

We will always write "⊆" for the infix operator "<<=".

We prove a basic lemma and register it with *auto*.

```
Lemma incl_nil X (A : list X) :
  nil ⊆ A.
Proof. intros x []. Qed.
Hint Resolve incl_nil.
```

Exercise 8.2.1 Prove the following lemmas from the standard library.

```
Lemma incl_appl X (A B C : list X) :  A ⊆ B → A ⊆ B ++ C.
Lemma incl_appr X (A B C : list X) :  A ⊆ B → A ⊆ C ++ B.
Lemma incl_app X (A B C : list X) :  A ⊆ C → B ⊆ C → A ++ B ⊆ C.
```

Exercise 8.2.2 Prove the following lemma. Hint: Use the lemma *in_map_iff*.

```
Lemma incl_map X Y A B (f : X → Y) :  A ⊆ B → map f A ⊆ map f B.
```

Exercise 8.2.3 Prove the following lemmas.

```
Variable X : Type.
Implicit Types A B : list X.
Lemma incl_nil_eq A :  A ⊆ nil → A=nil.
Lemma incl_shift x A B :  A ⊆ B → x::A ⊆ x::B.
Lemma incl_lcons x A B :  x::A ⊆ B ↔ x ∈ B ∧ A ⊆ B.
Lemma incl_rcons x A B :  A ⊆ x::B → ¬ x ∈ A → A ⊆ B.
Lemma incl_lrcons x A B :  x::A ⊆ x::B → ¬ x ∈ A → A ⊆ B.
```

8.3 List Equivalence

List membership also gives us list equivalence:

$$A \equiv B \;\leftrightarrow\; \forall x.\, x \in A \leftrightarrow x \in B$$

Note that two lists are equivalent if and only if they represent the same set. Figure 8.3 shows some useful laws for list equivalence.

We define list equivalence in Coq and register it with *auto*.

```
Definition equi X (A B : list X) : Prop :=
  A ⊆ B ∧ B ⊆ A.
Notation "A === B" := (equi A B) (at level 70).
Hint Unfold equi.
```

We will always write "≡" for the infix operator "===".

We prove some of the laws for list equivalence.

$x \in A \to A \equiv x :: A$	push
$x :: A \equiv x :: x :: A$	dup
$x :: y :: A \equiv y :: x :: A$	swap
$x :: A +\!\!+ B \equiv A +\!\!+ x :: B$	shift
$x :: A \equiv A +\!\!+ [x]$	rotate

Figure 8.3: Equivalence laws for lists

```
Section Equi.
  Variable X : Type.
  Implicit Types A B : list X.

  Lemma equi_push x A :  x ∈ A → A ≡ x::A.

  Proof. auto. Qed.

  Lemma equi_dup x A :  x::A ≡ x::x::A.

  Proof. auto. Qed.

  Lemma equi_swap x y A:  x::y::A ≡ y::x::A.

  Proof. split ; intros z ; simpl ; tauto. Qed.

  Lemma equi_rotate x A :  x::A ≡ A++[x].

  Proof.
    split ; intros y ; simpl.
    - intros [D|D] ; subst ; auto.
    - intros D. apply in_app_iff in D as [D|D].
      + auto.
      + apply in_sing in D. auto.
  Qed.
End Equi.
```

The enclosing section makes it possible to declare the variable X and the typings of the variables A and B in one go. Note that *auto* is in fact using the definitions and lemmas we registered with it. Also note that the tactic *subst* is used without arguments. In this case *subst* will eliminate as many variables as it can.

Exercise 8.3.1 Prove the following lemma.

```
Lemma equi_shift X (x : X) A B :  x::A++B ≡ A++x::B.
```

8.4 Automatic Decision Inference

Many results for lists depend on decidability assumptions. For instance, membership in and equality between lists are decidable if the base type comes with decidable equality. We now set up an infrastructure that can derive decisions and decision functions automatically. The infrastructure is based on Coq's type classes, which provide an inference mechanism for implicit arguments that is based on user-defined rules. In the following, we constrain our interest to the inference mechanism and make no attempt to explain the type class mechanism in general.[1] [2]

Recall the basic definition for decidability.

```
Definition dec (X : Prop) : Type := {X} + {¬ X}.
```

We define a parametric identity function for *dec X*

```
Definition decision (X : Prop) (D : dec X) : dec X := D.
```

and declare the second argument rather than the first argument implicit.

```
Arguments decision X {D}.
```

Now we can write *decision X* if we need a decision for a proposition X. As is, Coq is not able to derive the implicit argument of *decision X* and thus the term *decision X* will not type check. However, we can enable Coq to derive the implicit argument of *decision X* for many propositions X by registering so-called **instance rules**.

We will register the instance rules shown in Figure 8.4 for *dec*.[3] Each instance rule will be established with a definition or lemma. To register inference rules for *dec* with Coq, we first register *dec* as a type class.

```
Existing Class dec.
```

We start with the rules for equality on *nat* and on list types. We define the rules using the certifying tests *eq_nat_dec* and *list_eq_dec* from the standard library.

```
Definition eq_nat_Dec (x y : nat) :  dec (x = y) :=
  eq_nat_dec x y.
Definition eq_list_dec (X : Type) :
  (∀ x y : X, dec (x=y)) → ∀ A B : list X, dec (A = B).
  intros D. apply list_eq_dec. exact D.
Defined.
```

[1] The idea of automatic decision inference is due to Steven Schäfer and Sigurd Schneider.

[2] Type classes are still an experimental feature of Coq. When you step through our proofs, you will notice that type class inference occasionally fails in situations where we would expect it succeeds.

[3] The two rules for list quantification can be formulated with the premise $\forall x.\ x \in A \to dec\,(p\,x)$. We don't use this weaker premise since it doesn't go well with type class inference and also is not needed in the applications to come.

$$\frac{}{dec\ True} \qquad \frac{}{dec\ False}$$

$$\frac{dec\ X \qquad dec\ Y}{dec\ (X \to Y)} \qquad \frac{dec\ X \qquad dec\ Y}{dec\ (X \wedge Y)} \qquad \frac{dec\ X \qquad dec\ Y}{dec\ (X \vee Y)}$$

$$\frac{x\ y : nat}{dec\ (x = y)} \qquad \frac{x\ y : nat}{dec\ (x \leq y)}$$

$$\frac{A : list\ X \qquad \forall x\ y : X.\ dec\ (x = y)}{dec\ (x \in A)} \qquad \frac{A\ B : list\ X \qquad \forall x\ y : X.\ dec\ (x = y)}{dec\ (A = B)}$$

$$\frac{\forall x.\ dec\ (p\,x)}{dec\ (\forall x.\ x \in A \to p\,x)} \qquad \frac{\forall x.\ dec\ (p\,x)}{dec\ (\exists x.\ x \in A \wedge p\,x)}$$

Figure 8.4: Instance rules for *dec*

Next we register the rules as instance rules.

```
Existing Instance eq_nat_Dec.
Existing Instance eq_list_dec.
```

Here are examples for the automatic decision inference we obtain with the instance rules *eq_nat_Dec* and *eq_list_dec*.

```
Set Printing Implicit.
Check decision (2 = 3).
Check decision ([0] = [0]).
Check fun A B  : list nat ⇒ decision (A = B).
Check fun A B  : list (list  nat) ⇒ decision (A = B).
Check fun X (D : ∀ x y : X, dec (x = y)) (A B  : list (list X)) ⇒ decision (A = B).
Unset Printing Implicit.
```

The enclosing commands *Set* and *Unset* switch printing of implicit arguments on ond off. This way we can see the implicit arguments Coq derives for *decision*. For a given example we can prove that Coq derives the correct certifying test.

```
Goal eq_list_dec (eq_list_dec eq_nat_dec) = fun A B ⇒ decision (A = B).
Proof. reflexivity. Qed.
```

We define a notation for saying that a type has decidable equality

```
Notation "'eq_dec' X" := (∀ x y : X, dec (x=y)) (at level 70).
```

and register instance rules for list membership $x \in A$ and comparisons $x \leq y$ using certifying tests from Coq's library. This time we write the definitions with the command *Instance*, which defines and registers the rules in one go.

```
Instance in_Dec (X : Type) (x : X) (A : list X) : eq_dec X → dec (x ∈ A).
  intros D. apply in_dec. exact D.
Defined.

Instance le_Dec (x y : nat) : dec (x ≤ y) :=
  le_dec x y.

Compute if decision (1 ∈ [2;1]) then true else false.
% true : bool
```

Decidable propositions are closed under the logical connectives. We register instance rules implementing this insight. Since we do not intend to compute with the certifying tests justifying the rules, we establish the more complex rules with opaque definitions (i.e., lemmas).

```
Instance True_dec : dec True :=
  left I.

Instance False_dec : dec False :=
  right (fun A ⇒ A).

Instance impl_dec (X Y : Prop) : dec X → dec Y → dec (X → Y).

Proof. unfold dec ; tauto. Qed.

Instance and_dec (X Y : Prop) : dec X → dec Y → dec (X ∧ Y).

Proof. unfold dec ; tauto. Qed.

Instance or_dec (X Y : Prop) : dec X → dec Y → dec (X ∨ Y).

Proof. unfold dec ; tauto. Qed.

Instance not_dec (X : Prop) : dec X → dec (¬ X).

Proof. intros D. exact (decision (X → False)). Qed.
```

Note the use of the automation tactic *tauto*, which solves the goals by treating boolean sums like disjunctions.[4]

We finish our infrastructure setup with a few additional commands. First, we register the definition of *dec* with *auto*.

```
Hint Unfold dec.
```

Next we use Coq's notation facility for tactics to establish a tactic notation *decide claim* that for a claim p expands to the command *exact* (*decision p*)).

[4] The rule *not_dec* is needed since *not* is made opaque for type class inference by many standard modules (e.g., *List*).

```
Tactic Notation "decide" "claim" :=
  match goal with
    | |- dec (?p) ⇒ exact (decision p)
  end.
Goal ∀ x y A , dec (x ≤ y → x ∈ A).
Proof. intros x y A. decide claim. Qed.
```

Next we establish a tactic notation *decide p* that expands to the command *destruct* (*decision p*).

```
Tactic Notation "decide" constr(p) :=
  destruct (decision p).
Tactic Notation "decide" constr(p) "as" simple_intropattern(i) :=
  destruct (decision p) as i.
Goal ∀ (x : nat) A, x ∈ A ∨ ¬ x ∈ A.
Proof. intros x A. decide (x ∈ A) as [D|D] ; auto. Qed.
```

The destructuring pattern $[D|D]$ used with *decide* in the proof of the goal can be omitted. In this case the first notational rule for *decide* applies and the tactic *destruct* generates the names for the assumptions.

Finally, we strengthen the instance inference for *dec* with a hint command. As is, Coq cannot infer the implicit argument for *decision* ((*fun* _ ⇒ *True*) *0*) since it does not reduce the beta redex. We fix the problem with a hint command.

```
Hint Extern 4 ⇒
match goal with
  | [  |- dec ((fun _ ⇒ _) _) ] ⇒ simpl
end : typeclass_instances.
Goal decision ((fun _ ⇒ True) 0) = True_dec.
Proof. reflexivity. Qed.
```

Exercise 8.4.1 Prove the following goals stating the decidability of list membership and list equality. Do not use certifying tests from the library or instance rules defined above.

```
Goal ∀ X (x : X) A , eq_dec X → dec (x ∈ A).
Goal ∀ X, eq_dec X → eq_dec (list X).
```

Exercise 8.4.2 Prove the following lemmas.

```
Lemma dec_DN X :  dec X → ¬¬ X → X.
Lemma dec_DM_and X Y :  dec X → dec Y → ¬ (X ∧ Y) → ¬ X ∨ ¬ Y.
Lemma dec_DM_impl X Y :  dec X → dec Y → ¬ (X → Y) → X ∧ ¬ Y.
```

Exercise 8.4.3 Prove that decidability propagates through logical equivalence.

```
Lemma dec_prop_iff (X Y : Prop) :  (X ↔ Y) → dec X → dec Y.
```

8.5 List Quantification Preserves Decidability

We now establish the instance rules implementing the fact that quantification over lists preserves decidability.

$$\frac{\forall x.\ dec\ (p\,x)}{dec\ (\forall x.\ x \in A \to p\,x)} \qquad \frac{\forall x.\ dec\ (p\,x)}{dec\ (\exists x.\ x \in A \wedge p\,x)}$$

We will obtain both rules from the following lemma. The lemma gives us a function that for a list and a decidable predicate yields either an element of the list not satisfying the predicate or a proof that every element of the list satisfies the predicate.

```
Lemma sigma_forall_list X A (p : X → Prop) (p_dec : ∀ x, dec (p x)) :
  {x | x ∈ A ∧ ¬ p x} + {∀ x, x ∈ A → p x}.
```

```
Proof.
  induction A as [|x A] ; simpl.
  − right. tauto.
  − destruct IHA as [[y [D E]]|D].
    + left. eauto.
    + destruct (p_dec x) as [E|E].
      * right. intros y [[]| F] ; auto.
      * left. eauto.
Qed.
```

To ease the use of the lemma *sigma_forall_list*, we declare A and p as explicit arguments and X and p_dec as implicit arguments.

```
Arguments sigma_forall_list {X} A p {p_dec}.
```

We now establish the instance rule for universal list quantification.

```
Instance forall_list_dec X A (p : X → Prop) (p_dec : ∀ x, dec (p x)) :
  dec (∀ x, x ∈ A → p x).
```

```
Proof.
  destruct (sigma_forall_list A p) as [[x [D E]]|D] ; unfold dec ; auto.
Qed.
```

Next we establish the rule for existential list quantification.

```
Instance exists_list_dec X A (p : X → Prop) (p_dec : ∀ x, dec (p x)) :
  dec (∃ x, x ∈ A ∧ p x).
```

```
Proof.
  destruct (sigma_forall_list A (fun x ⇒ ¬ p x)) as [[x [D E]]|D].
  − left. apply dec_DN in E ; eauto.
  − right. intros [x [E F]]. exact (D x E F).
Qed.
```

We have now enough instance rules so that Coq can derive decisions for inclusion, equivalence, and disjointness of lists.

```
Set Printing Implicit.
Check fun X (A B :list X) (D : eq_dec X) ⇒ decision (A ⊆ B).
Check fun X (A B :list X) (D : eq_dec X) ⇒ decision (A ≡ B).
Check fun X (A B :list X) (D : eq_dec X) ⇒ decision (disjoint A B).
Unset Printing Implicit.
```

Exercise 8.5.1 Prove the De Morgan law for universal list quantification.

```
Lemma dec_DM_forall X A  (p : X → Prop) :
  (∀ x, dec (p x)) →
  ¬ (∀ x, x ∈ A → p x) → ∃ x, x ∈ A ∧ ¬ p x.
```

Exercise 8.5.2 Prove constructive choice for existential list quantification.

```
Lemma dec_cc X (p : X → Prop) A :
  eq_dec X → (∀ x, dec (p x)) →
  (∃ x, x ∈ A ∧ p x) → {x | x ∈ A ∧ p x}.
```

8.6 Filtering of Lists

We define a function *filter* that given a decidable predicate and a list yields the sublist containing all elements satisfying the predicate.[5] [6]

```
Section Filter.
  Variable X : Type.
  Variable p : X → Prop.
  Variable p_dec : ∀ x, dec (p x).
  Fixpoint filter  (A : list  X) :  list  X :=
    match A with
      |  nil ⇒ nil
      |  x::A' ⇒ if decision (p x) then x :: filter A' else filter A'
    end.
End Filter.

Arguments filter {X} p {p_dec} A.
```

The function *filter* will play an important role in proofs of existential claims. We establish a few properties of *filter*. The proofs are left as exercises.

[5] Our definition of *filter* shadows the definition of *filter* in the standard library, which employs a boolean test rather than a decidable predicate.

[6] Note that an application of the command *Arguments* inside a section will have a local effect only.

```
Section FilterLemmas.
Variable X : Type.
Variable p : X → Prop.
Context {p_dec : ∀ x, dec (p x)}.

Lemma in_filter x A :
  x ∈ filter A ↔ x ∈ A ∧ p x.

Lemma filter_incl A :
  filter  A ⊆ A.

Lemma filter_mono A B :
  A ⊆ B → filter A ⊆ filter B.

End FilterLemmas.
```

Note the use of the command *Context*. It is a variant of the command *Variable* that accommodates the assumption as an implicit argument when the enclosing section is closed.

We establish two further lemmas. We do this in a new section since one of the proofs will use the lemma *in_filter* for the predicate *q*. The first lemma says that *filter* is monotone in the strength of the filtering predicate. The second lemma says that *filter* yields the same value for equivalent filtering predicates. The proofs are left as exercises.

```
Section FilterLemmas_pq.
  Variable X : Type.
  Variable p q : X → Prop.
  Context {p_dec : ∀ x, dec (p x)}.
  Context {q_dec : ∀ x, dec (q x)}.

Lemma filter_pq_incl A :
  (∀ x, x ∈ A → p x → q x) → filter p A ⊆ filter q A.

Lemma filter_pq_eq A :
  (∀ x, x ∈ A → (p x ↔ q x)) → filter p A = filter q A.

End FilterLemmas_pq.
```

Exercise 8.6.1 Prove the lemmas stated in this section.

Exercise 8.6.2 Prove the following lemma.

```
Lemma separation X A p (D : ∀ x : X, dec (p x)) :
  {B | ∀ x, x ∈ B ↔ x ∈ A ∧ p x}.
```

Exercise 8.6.3 The result of *filter* does not depend on the particular decision function used. Prove the following lemma formulating this statement.

```
Lemma filter_independence X A p (D D' : ∀ x : X, dec (p x)) :
  filter  p A (p_dec:=D) = filter p A (p_dec:=D').
```

8.7 Rewriting with List Equivalences

Many operations on lists respect list equivalence in that they yield equivalents results for equivalent arguments. Here are the equivalence rules we are going to use.

$$\frac{A \equiv A'}{x :: A \equiv x :: A'} \qquad \frac{A \equiv A' \qquad B \equiv B'}{A +\!\!+ B \equiv A' +\!\!+ B'} \qquad \frac{A \equiv A'}{x \in A \leftrightarrow x \in A'} \qquad \frac{A \equiv A' \qquad B \equiv B'}{A \subseteq B \leftrightarrow A' \subseteq B'}$$

The equivalence rules justify rewriting with list equivalences. We can perform such rewritings with the tactic *setoid_rewrite* once we have registered the rules with Coq. To do so, we need to load the standard module *Morphisms*, register the predicate *equi* with the predefined class *Equivalence*, and then establish every rule as an instance rule. We give the necessary Coq commands below. If you step through the scripts you will get goals for all the facts that must be proved. While it is important to understand the logical structure of what you see, the technical details of the commands do not matter.[7]

```
Instance equi_Equivalence X : Equivalence (@equi X).
Proof. constructor ; hnf ; firstorder. Qed.
Instance cons_equi_proper X :
  Proper (eq ==> @equi X ==> @equi X) (@cons X).
Proof. hnf. intros x y []. hnf. firstorder. Qed.
Instance app_equi_proper X :
  Proper (@equi X ==> @equi X ==> @equi X) (@app X).
Proof.
  hnf. intros A B D. hnf. intros A' B' E.
  destruct D, E ; auto using incl_app, incl_appl, incl_appr.
Qed.
Instance in_equi_proper X :
  Proper (eq ==> @equi X ==> iff) (@In X).
Proof. hnf. intros x y []. hnf. firstorder. Qed.
Instance incl_equi_proper X :
  Proper (@equi X ==> @equi X ==> iff) (@incl X).
Proof. hnf. intros x y D. hnf. firstorder. Qed.
```

Note that several of the proofs use the automation tactic *firstorder*, which can solve simple goals involving quantifiers. Also note that in one of the proofs *auto* is enhanced with the lemmas *incl_app* and *incl_appl*.

Here is an example proof employing rewriting with the equivalences *equi_swap* and *equi_shift* established in Section 8.1.

[7] Recall that we are already using *setoid_rewrite* with logical equivalences (see Section 2.9). In fact, the predicate *iff* is registered with the predefined class *Equivalence*, and the rules justifying rewriting with logical equivalences are registered with *iff*.

```
Goal ∀ X (x y : X) A B,  x::A ++ [y] ++ A ++ B ≡ A ++ [y;x] ++ A ++ B.
Proof.
  intros X x y A B. simpl.
  setoid_rewrite equi_swap.
  setoid_rewrite equi_shift at 1.
  reflexivity.
Qed.
```

Note the use of the tactic *reflexivity* to solve the final goal. This is justified since we have established *equi* as a reflexive predicate when we registered it as an equivalence predicate (see the instance definition *equi_Equivalence* above).

Next we register list inclusion *incl* as a preorder (i.e., a reflexive and transitive predicate). This makes it possible to use the tactics *reflexivity* and *transitivity* for list inclusions. Moreover, it justifies top level setoid rewriting of list inclusions with list inclusions.

```
Instance incl_preorder X : PreOrder (@incl X).
Proof. constructor ; hnf ; unfold incl ; auto. Qed.
Goal ∀ A B C D : list nat, A ⊆ B → B ⊆ C →  C ⊆ D → A ⊆ D.
Proof. intros A B C D F G H. setoid_rewrite F. setoid_rewrite ← H. exact G. Qed.
```

8.8 Duplicate-free Lists

A list is duplicate-free if it contains no element twice. Duplicate-free lists have the important property that equivalent duplicate-free lists have the same length. Moreover, the length of a duplicate-free list is the cardinality of the set the list represents.

We start with an inductive definition of duplicate freeness.

$$\frac{}{\mathit{dupfree}\ \mathit{nil}} \qquad \frac{x \notin A \qquad \mathit{dupfree}\ A}{\mathit{dupfree}\ (x :: A)}$$

Writing the definition in Coq is straightforward.

```
Inductive dupfree (X : Type) : list X → Prop :=
| dupfreeN : dupfree nil
| dupfreeC x A : ¬ x ∈ A → dupfree A → dupfree (x::A).
```

We prove an inversion lemma for the predicate *dupfree*.

```
Section Dupfree.
Variable X : Type.
Implicit Types A B : list X.
```

```
Lemma dupfree_inv x A :
  dupfree (x::A) → ¬x ∈ A ∧ dupfree A.
Proof. intros D. inversion D ; subst ; auto. Qed.
```

Note the use of the automation tactic *inversion*. We use *inversion* only if it solves the goal, possibly together with other automation tactics as above. The inversion lemma *dupfree_inv* will spare us further uses of the inversion tactic for *dupfree*.

Under certain conditions, the functions *app*, *map*, and *filter* preserve duplicate freeness. We state these facts with three lemmas.

```
Lemma dupfree_app A B :
  disjoint A B → dupfree A → dupfree B → dupfree (A++B).
Lemma dupfree_map Y A (f : X → Y) :
  (∀ x y, x ∈ A → y ∈ A → f x = f y → x=y) → dupfree A → dupfree (map f A).
Lemma dupfree_filter p (p_dec : ∀ x, dec (p x)) A :
  dupfree A → dupfree (filter p A).
Proof.
  intros D. induction D as [|x A C D] ; simpl.
  - left.
  - decide (p x) as [E|E] ;  [|exact IHD].
    right ;  [|exact IHD].
    intros F. apply C. apply filter_incl in F. exact F.
Qed.
End Dupfree.
```

The proof script for *dupfree_filter* uses the construction ; [|*exact IHD*] twice. This construction applies in situations where two subgoals are created and the second subgoal can be solved with a single tactic. In such situations it is usually convenient to immediately solve the second subgoal and then continue without branching with the first subgoal.

Exercise 8.8.1 Prove the lemma *dupfree_inv* without using the inversion tactic.

Exercise 8.8.2 Prove the lemma *dupfree_filter* without using the construct ; [|*exact IHD*].

Exercise 8.8.3 Prove the lemmas *dupfree_app* and *dupfree_map* using induction for *dupfree*.

Exercise 8.8.4 Prove that the predicate *dupfree* is decidable.

```
Lemma dupfree_dec X (A : list X) :  eq_dec X → dec (dupfree A).
```

8.9 Undup

We now define a function *undup* that maps a list to an equivalent duplicate-free list by keeping only the last occurrence of an element. We need the assumption that the base type has decidable equality.

```
Section Undup.
Variable X : Type.
Context {eq_X_dec : eq_dec X}.
Implicit Types A B : list X.

Fixpoint undup (A : list X) : list X :=
  match A with
    | nil ⇒ nil
    | x::A' ⇒ if decision (x ∈ A') then undup A' else  x :: undup A'
  end.
```

We prove the correctness of *undup*. Note that rewriting with list equivalences is essential in the proofs.

```
Lemma undup_equi A :
  undup A ≡ A.

Proof.
  induction A as [|x A] ; simpl.
  - reflexivity.
  - decide (x ∈ A) as [E|E].
    + setoid_rewrite IHA. apply equi_push, E.
    + setoid_rewrite IHA. reflexivity.
Qed.

Lemma undup_dupfree A :
  dupfree (undup A).

Proof.
  induction A as [|x A] ; simpl.
  - left.
  - decide (x ∈ A) as [E|E].
    + exact IHA.
    + right.
      * setoid_rewrite undup_equi. exact E.
      * exact IHA.
Qed.
```

Exercise 8.9.1 Prove the following lemmas.

```
Lemma undup_homo A B :
  A ⊆ B ↔ undup A ⊆ undup B.

Lemma undup_iso A B :
  A ≡ B ↔ undup A ≡ undup B.
```

$A \subseteq B \to \lvert A\rvert \leq \lvert B\rvert$	le
$A \equiv B \to \lvert A\rvert = \lvert B\rvert$	eq
$A \subseteq B \to \lvert A\rvert = \lvert B\rvert \to A \equiv B$	equi
$\lvert A\rvert < \lvert B\rvert \to \exists x.\, x \in B \wedge x \notin A$	ex
$A \subseteq B \to x \in B \to x \notin A \to \lvert A\rvert < \lvert B\rvert$	lt

Laws (equi) and (ex) require a base type with decidable equality

Figure 8.5: Length laws for duplicate-free lists

```
Lemma undup_eq A :
  dupfree A → undup A = A.
Lemma undup_idempotent A :
  undup (undup A) = undup A.
```

8.10 Length of Duplicate-free Lists

The recursive function *length* from the standard library yields the length of a list. The definition of *length* is based on the equations

$$\begin{aligned} \lvert nil\rvert &= 0 \\ \lvert x :: A\rvert &= 1 + \lvert A\rvert \end{aligned}$$

We will use the notation

```
Notation "| A |" := (length A) (at level 65).
```

In the following we prove basic laws about the length of duplicate-free lists. The laws are shown in Figure 8.5. Note that the laws reflect basic facts about the cardinality of finite sets. Recall that the length of a duplicate-free list is the cardinality of the represented set.

For the proofs of the laws we need a reordering lemma for duplicate-free lists.

```
Section DupfreeLength.
Variable X : Type.
Implicit Types A B : list X.
Lemma dupfree_reorder A x :
  dupfree A → x ∈ A →
  ∃ A', A ≡ x::A' ∧ |A'| < |A| ∧ dupfree (x::A').
```

```
Proof.
  intros E. revert x. induction E as [|y A H] ; intros x F.
  - contradiction F.
  - destruct F as [F|F].
    + subst y. ∃ A. auto using dupfree.
    + specialize (IHE x F). destruct IHE as [A' [G [K1 K2]]].
      ∃ (y::A'). split  ;[| split ].
      * setoid_rewrite G. apply equi_swap.
      * simpl. omega.
      * { apply dupfree_inv in K2 as [K2 K3]. right.
          - intros [M|M] ; subst ; auto.
          - right ;  [|exact K3]. intros M ; apply H. apply G. auto. }
Qed.
```

The proof is by induction on the assumption *dupfree A*. The annotation *using dupfree* enhances the auto tactic with all constructors of the inductive type *dupfree*. Note the use of the construct ; [|*split*] to further split the second subgoal of a split. Also note the use of the inversion lemma for *dupfree*.

Next we prove that length respects inclusion of duplicate-free lists.

```
Lemma dupfree_le A B :
  dupfree A → dupfree B → A ⊆ B → |A| ≤ |B|.
Proof.
  intros E. revert B. induction A as [|x A] ; simpl ; intros B F G.
  - omega.
  - apply incl_lcons in G as [G H].
    destruct (dupfree_reorder F G) as [B' [K [L M]]].
    apply dupfree_inv in E as [E1 E2]. apply dupfree_inv in M as [M1 M2].
    cut (A ⊆ B').
    { intros N. specialize (IHA E2 B' M2 N). omega. }
    apply incl_rcons with (x:=x) ; [|exact E1].
    setoid_rewrite H. apply K.
Qed.
```

The proof is by induction on A and uses the reordering lemma *dupfree_reorder*.

It is now straightforward to prove that equivalent duplicate-free lists have the same length.

```
Lemma dupfree_eq A B :
  dupfree A → dupfree B → A ≡ B → |A|=|B|.
```

Using the lemmas *sigma_forall_list* and *dupfree_le*, we can prove that a duplicate-free list containing a shorter duplicate-free list must have an element that is not in the shorter list.

```
Lemma dupfree_ex A B :
  eq_dec X → dupfree A → dupfree B → |A| < |B| → ∃ x, x ∈ B ∧ ¬ x ∈ A.
```

We can also prove the opposite direction: If a duplicate-free list is contained in a duplicate-free list containing an extra element, the list with the etra argument must be longer.

```
Lemma dupfree_lt A B x :
  dupfree A → dupfree B → A ⊆ B → x ∈ B → ¬ x ∈ A  → |A| < |B|.
```

This brings us to the final result of this section: Duplicate-free lists of the same length are equivalent if one contains the other.

```
Lemma dupfree_equi A B :
  eq_dec X → dupfree A → dupfree B → A ⊆ B → |A| = |B| → A ≡ B.
```

Exercise 8.10.1 Prove the lemma *dupfree_eq*.

Exercise 8.10.2 Prove the lemma *dupfree_ex*. Use the lemmas *sigma_forall_list* and *dupfree_le*. No induction is needed.

Exercise 8.10.3 Prove the lemma *dupfree_lt*. Use the lemmas *dupfree_reorder*, *dupfree_eq*, and *dupfree_le*. No induction is needed.

Exercise 8.10.4 Prove the lemma *dupfree_equi*. Use the lemmas *sigma_forall_list* and *dupfree_lt*. No induction is needed.

Exercise 8.10.5 Prove the following Lemma.

```
Lemma dupfree_or X (A B : list X) :
  eq_dec X → dupfree A → dupfree B → A ⊆ B → A ≡ B ∨ |A| < |B|.
```

Hint: Use *dupfree_le*, *dupfree_equi*, and *le_lt_eq_dec* from the standard library.

8.11 Cardinality of Lists

We are now ready to define the cardinality of lists.

```
Section Cardinality.
Variable X : Type.
Context {eq_X_dec : eq_dec X}.
Implicit Types A B : list X.

Definition card (A : list X) : nat := |undup A|.
```

Note that the cardinality function requires a base type with decidable equality since the function *undup* does.

We will prove the basic cardinality laws shown in Figure 8.6. Because of the cardinality function, all laws require a base type with decidable equality. Mathematically, the cardinality laws are straightforward consequences of the length

$A \subseteq B \to card\ A \leq card\ B$	le
$A \equiv B \to card\ A = card\ B$	eq
$A \subseteq B \to card\ A = card\ B \to A \equiv B$	equi
$card\ A < card\ B \to \exists x.\ x \in B \wedge x \notin A$	ex
$A \subseteq B \to x \in B \to x \notin A \to card\ A < card\ B$	lt
$A \subseteq B \to A \equiv B \vee card\ A < card\ B$	or

Figure 8.6: Cardinality laws for lists

laws for duplicate-free lists and a few facts about the undup function. Proving the laws in Coq nevertheless takes a little effort. We use the occasion to introduce new features of the tactics *destruct* and *assert* that ease the proofs.

The proofs of the first three cardinality laws in Figure 8.6 are left as exercise. We show the proof of the law *card_equi* since it illustrates certain weaknesses of the tactic *apply* in the current version of Coq (8.4pl2).

```
Lemma card_equi A B :
  A ⊆ B → card A = card B → A ≡ B.
Proof.
  intros D E. apply ← undup_iso.
  apply dupfree_equi.
  – exact eq_X_dec.
  – apply undup_dupfree.
  – apply undup_dupfree.
  – apply undup_homo, D.
  – exact E.
Qed.
```

Note the arrow "←" in the first use of *apply*. It specifies which direction of the equivalence lemma *undup_iso* is to be applied. Given that only one direction of the lemma applies, one would expect that *apply* chooses this direction automatically, but this is not the case. Also note that the second use of apply spans a subgoal *eq_dec X* for an argument of *dupfree_equi*. This seems unnecessary given the type class *dec* and the assumption *eq_X_dec*.

The proof of the cardinality law *card_or* illustrates a useful feature of the tactic *assert*.

```
Lemma card_or A B :
  A ⊆ B → A ≡ B ∨ card A < card B.
Proof.
  intros D.
```

```
  assert (F:= card_le D).
  apply le_lt_eq_dec in F as [F|F].
  − auto.
  − left. exact (card_equi D F).
Qed.
```

The tactic *assert* is used with the definition symbol ":=" to establish an assumption $F : card\ A \leq card\ B$ directly with a proof term. Then the decision lemma *le_lt_eq_dec* from the library is used to split the proof into cases for $card\ A < card\ B$ and $card\ A = card\ B$.

The proof of the cardinality law *card_ex* illustrates a new feature of the tactic *destruct*.

```
Lemma card_ex A B :
  card A < card B → ∃ x, x ∈ B ∧ ¬ x ∈ A.
Proof.
  intros E.
  destruct (dupfree_ex (A:=undup A) (B:=undup B)) as [x F].
  − apply undup_dupfree.
  − apply undup_dupfree.
  − exact E.
  − ∃ x. setoid_rewrite undup_equi in F. exact F.
Qed.
```

The proof applies the tactic *destruct* to a function that yields a proof of an existential quantification. In this situation, *destruct* automatically introduces subgoals to establish the arguments of the function it cannot derive automatically. The subgoals for the missing arguments go before the subgoals for the case analysis. Using *destruct* in *mediating mode* spares the effort of establishing the arguments of the function with the tactic *assert*.

Exercise 8.11.1 Prove the following cardinality laws.

```
Lemma card_le A B :  A ⊆ B → card A ≤ card B.
Lemma card_eq A B :  A ≡ B → card A = card B.
Lemma card_lt A B x :  A ⊆ B → x ∈ B → ¬ x ∈ A → card A < card B.
```

8.12 Library LFS

In the rest of the book we will use the infrastructure defined in this chapter. For Coq developments we provide a file *LFS.v* containing the Coq code establishing the infrastructure of this chapter. Copy the file into the directory you are working in and compile it with the shell command *coqc*. This will install the library *LFS*. If you now start your Coq development with the line

```
Require Import LFS.
```

you will get all the infrastructure set up in this chapter. This includes setting the implicit arguments mode and importing the standard modules *Omega*, *List*, and *Morphisms*. Browse the file *LFS.v* to see the details.

Coq Summary

New Tactics and Tacticals

- *eauto*. Variant of *auto* that can prove existential quantifications if strong enough assumptions exist. See lemma *disjoint_cons* in Section 8.1.
- *firstorder*. Automation tactic that can solve simple goals involving quantifiers. See Section 8.7 on setoid rewriting.
- ; [|*t*]. If a tactic is used with the suffix ; [|*t*], it must have produced two subgoals. The tactic *t* is applied to the second subgoal. Can be used to solve or split the second subgoal.

Tactic *apply* with Destructuring Pattern

If the tactic *apply* is applied to an assumption, a destructuring pattern can be specified so that the value obtained by the application is immediately destructured. See the command *apply IHA in D as* [*B* [*C D*]] in the proof of *in_iff* in Section 8.1.

Tactic *destruct* Applied to Functions

If the tactic *destruct* is applied to a function, Coq spans subgoals for the arguments it cannot derive automatically. The subgoals for the arguments go before the subgoals for the case analysis. This is also the case for implicit destructs invoked with a destructuring pattern used with *apply*. See the proof of lemma *card_ex* in Section 8.11.

Tactic *assert* with :=

The tactic *assert* can establish an assumption directly from a proof term. See the proof of lemma *card_or* in Section 8.11.

Tactic *auto* with *using*

The tactic *auto* can be enhanced with the constructors of an inductive type specified with a *using* suffix. See the proof of the lemma *dupfree_reorder* in Section 8.10.

Setoid Rewriting with List Equivalences and List Inclusions

One can register equivalence and preorder predicates for setoid rewriting. We have registered list equivalence and list inclusion. See Section 8.7 on setoid

rewriting.

New Commands

- *Hint Resolve* enhances *auto* with lemmas.
- *Hint Unfold* tells *auto* to unfold definitions.
- *Existing Class x* registers the constant x as a type class. See the registration of *dec* in Section 8.4.
- *Existing Instance x* registers the constant x as an instance rule. See the registration of *eq_nat_dec* in Section 8.4.
- *Instance* establishes and registers an instance rule for a type class.
- *Implicit Types* associates names with types to be used with type inference.
- *Context* declares an assumption in a section that will be accommodated as an implicit argument when the enclosing section is closed. See the section *FilterLemmas* in Section 8.6.
- *Set Printing Implicit* and *Unset Printing Implicit* switch printing of implicit arguments on and off.
- *Print Instances C* lists all instance rules for class *C*.

Library LFS

We provide the infrastructure set up in this chapter through a library *LFS*. The Coq developments of the remaining chapters all start with the line *Require Import LFS*. For this command to work the library LFS needs to be in your load path. You can install the library *LFS* by copying the file *LFS.v* into your working directory and then compiling it with the shell command *coqc*.

Tactic Notations Defined in LFS

- *decide claim*. Expands to *exact* (*decision p*) for a claim p.
- *decide p*. Expands to *destruct* (*decision p*).

9 Proof Systems for Propositional Logic

Propositional logic is a logical system for propositional formulas. Propositional formulas consist of atomic propositions (e.g., propositional variables, $\bot$ and $\top$) and are closed under logical connectives (e.g., implication, conjunction and disjunction). In this chapter we will restrict ourselves to propositional logic with propositional variables and $\bot$ and closed under implication. The systems and results can be extended to include other connectives, but we leave such extensions to exercises.

We will first consider a natural deduction style proof system for intuitionistic propositional logic. The natural deduction system will correspond closely to the proof system in Coq. We next consider a classical natural deduction style proof system. We will prove a result of Glivenko: a propositional formula is classically provable if and only if its double negation is intuitionistically provable. We will then consider a Hilbert style proof system and prove the equivalence of the natural deduction system and the Hilbert system.[1]

9.1 Propositional Formulas

We start with **(propositional) formulas** given by the following grammar where x ranges over variables and s and t range over propositional formulas.

$$s, t ::= x \mid \bot \mid s \to t$$

We can represent such formulas in Coq using an inductive type. We use natural numbers to represent variables.

```
Definition var := nat.

Inductive form : Type :=
| Var : var → form
| Imp : form → form → form
| Fal : form.
```

Just as in Coq, we consider $\neg s$ as meaning $s \to \bot$. We implement this using a Coq definition.

[1] Proof systems are often called "calculi." In this context, "calculus" is a synonym for "system."

```
Definition Not s := Imp s Fal.
```

Equality of formulas is decidable.

```
Instance form_eq_dec : ∀ s t:form, dec (s = t).
Proof. unfold dec. repeat decide equality. Qed.
```

Exercise 9.1.1 Prove decidability of equality of formulas without using the *decide equality* tactic.

Exercise 9.1.2 Consider the following specification for a function which has the type $form \to (var \to Prop) \to Prop$.

```
Definition valspec (v:form → (var → Prop) → Prop) : Prop :=
 ∀ p, (∀ x, v (Var x) p ↔ p x) ∧
      (¬ v Fal p) ∧
      (∀ s t, v (Imp s t) p ↔ (v s p → v t p)).
```

a) Prove the following goals.

```
Goal ∀ v p s, valspec v → v (Imp Fal s) p.
Goal ∀ v p s, valspec v → v (Imp s s) p.
Goal ∀ v p s t, valspec v → v (Imp s (Imp t s)) p.
```

b) Define a function *val* and prove the following:

```
Lemma valspec_val : valspec val.
```

9.2 Natural Deduction System

In this section we consider our first proof system for propositional formulas. A proof system defines when a formula s is provable from a finite collection of assumptions A. We write $A \vdash s$ to mean s is provable from A in the particular proof system under discussion. The symbol $\vdash$ is a "turnstile." Note that $\vdash$ is a predicate on A and s and we will call it the **provability predicate**. Provability predicates are typically defined inductively. The **judgment** of the proof system is the provability predicate and the propositions built from it. We write $A \nvdash s$ to mean the negation of $A \vdash s$.

Deduction rules for logical connectives were given in Figure 2.1. The introduction and elimination rules for $\to$ together with the elimination rule for $\bot$ essentially give a proof system for propositional formulas. Since the introduction rule for $\to$ changes the assumptions, we must have some explicit notion of a collection of assumptions and some way of checking if a formula is such an assumption. We use lists of formulas to represent such a collection of assumptions

$$\mathsf{A}\ \frac{}{A \vdash s}\ s \in A \qquad \mathsf{II}\ \frac{A, s \vdash t}{A \vdash s \to t} \qquad \mathsf{IE}\ \frac{A \vdash s \to t \qquad A \vdash s}{A \vdash t} \qquad \mathsf{E}\ \frac{A \vdash \bot}{A \vdash s}$$

Figure 9.1: Natural Deduction Rules

and refer to such a list of formulas as a **context**. We can check if a formula is an assumption in the context using an **assumption rule** to check if the formula is an element of the list. These rules are given in Figure 9.1.

The rules in Figure 9.1 define when $A \vdash s$ in the system $\mathcal{N}$. That is, the rules define when s is provable from A in $\mathcal{N}$. One can use the rules in Figure 9.1 to justify $A \vdash s$.

Consider the following example.

Example 9.2.1 Let A be a context and s and t be formulas. We can use the rules of $\mathcal{N}$ to derive $A \vdash s \to \neg s \to t$ as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{\mathsf{A}\ \dfrac{}{A, s, \neg s \vdash s \to \bot} \qquad \mathsf{A}\ \dfrac{}{A, s, \neg s \vdash s}}{\mathsf{IE}\ \dfrac{}{A, s, \neg s \vdash \bot}}}{\mathsf{E}\ \dfrac{}{A, s, \neg s \vdash t}}}{\mathsf{II}\ \dfrac{}{A, s \vdash \neg s \to t}}}{\mathsf{II}\ \dfrac{}{A \vdash s \to \neg s \to t}}$$

We can represent the natural deduction system $\mathcal{N}$ in Coq as an inductive predicate *nd*. The proposition *nd A s* is provable precisely when $A \vdash s$. The predicate *nd* and propositions of the form *nd A s* are the Coq representations of the judgment of $\mathcal{N}$. Note that A is a nonuniform parametric argument of *nd* and s is a nonparametric argument of *nd*.

```
Definition context := list form.

Inductive nd (A : context) : form → Prop :=
| ndA s :    s ∈ A → nd A s
| ndII s t :  nd (s::A) t → nd A (Imp s t)
| ndIE s t :  nd A (Imp s t) → nd A s → nd A t
| ndE s :    nd A Fal → nd A s.
```

We can now reconsider Example 9.2.1 as a proof in Coq. Compare the Coq proof script with the diagram in Example 9.2.1.

```
Goal ∀ A s t, nd A (Imp s (Imp (Not s) t)).
```

```
Proof.
  intros A s t. apply ndII, ndII. apply ndE. apply ndIE with (s := s).
  - apply ndA. left. reflexivity.
  - apply ndA. right. left. reflexivity.
Qed.
```

The following alternative Coq proof script may also be enlightening if one compares it with the diagram in Example 9.2.1.

```
Goal ∀ A s t, nd A (Imp s (Imp (Not s) t)).
Proof.
  intros A s t.
  assert (B:Not s ∈ Not s::s::A) by auto.
  assert (C:s ∈ Not s::s::A) by auto.
  exact (ndII (ndII (ndE t (ndIE (ndA B) (ndA C))))).
Qed.
```

As one can see in the Coq scripts above, in general to prove $A \vdash s$ using the assumption rule we must prove the subgoal $s \in A$. It is useful to prove three special cases $A, s \vdash s$ and $A, s, t \vdash s$ and $A, s, t, u \vdash s$ where the assumption is one of the three most recently added formulas in the context. The use of these lemmas avoids subgoals of the form $s \in s :: A$, $s \in t :: s :: A$ and $s \in u :: t :: s :: A$.

```
Lemma ndA1 A s :
 nd (s :: A) s.
Proof. apply ndA. left. reflexivity. Qed.
Lemma ndA2 A s t :
 nd (t :: s :: A) s.
Proof. apply ndA. right. left. reflexivity. Qed.
Lemma ndA3 A s t u :
 nd (u :: t :: s :: A) s.
Proof. apply ndA. right. right. left. reflexivity. Qed.
```

A rule is **admissible** in a proof system if adding the rule to the proof system does not change what is provable in the system. This is equivalent to saying that the conclusion of the rule is provable whenever the premises are provable. The lemmas *ndA1*, *ndA2* and *ndA3* prove that the rules A1, A2 and A3 in Figure 9.2 are all admissible in the proof system $\mathcal{N}$. We will go on to prove that the other rules in Figure 9.2 are admissible in $\mathcal{N}$.

We will often do proofs by induction over $A \vdash s$, i.e., over the inductive predicate *nd*. In the process of doing such an inductive proof we must consider each of the four rules in Figure 9.1. Each rule has the form

$$\frac{A_1 \vdash s_1 \cdots A_n \vdash s_n}{A \vdash s}$$

$$\mathsf{A1}\ \frac{}{A, s \vdash s} \qquad \mathsf{A2}\ \frac{}{A, s, t \vdash s} \qquad \mathsf{A3}\ \frac{}{A, s, t, u \vdash s} \qquad \mathsf{app}\ \frac{A \vdash s}{A \vdash u}\ s \to u \in A$$

$$\mathsf{app1}\ \frac{A, s \to u \vdash s}{A, s \to u \vdash u} \qquad \mathsf{app2}\ \frac{A, s \to u, t \vdash s}{A, s \to u, t \vdash u} \qquad \mathsf{app3}\ \frac{A, s \to u, t, v \vdash s}{A, s \to u, t, v \vdash u}$$

$$\mathsf{weak}\ \frac{A \vdash s}{A' \vdash s}\ A \subseteq A' \qquad \mathsf{W}\ \frac{A \vdash s}{A, t \vdash s} \qquad \mathsf{DN}\ \frac{A \vdash s}{A \vdash \neg\neg s}$$

Figure 9.2: Some Admissible Rules

for $n \in \{0, 1, 2\}$. For such a rule we assume the desired property for A_i and s_i as inductive hypotheses and prove the desired property for A and s. In Coq the induction principle of *nd* corresponds to the type of *nd_ind*:

```
Check (nd_ind :
  ∀ p : context → form → Prop,
  (∀ (A : context) (s : form), s ∈ A → p A s) →
  (∀ (A : context) (s t : form), nd (s :: A) t → p (s :: A) t → p A (Imp s t)) →
  (∀ (A : context) (s t : form), nd A (Imp s t) → p A (Imp s t) → nd A s → p A s → p A t) →
  (∀ (A : context) (s : form), nd A Fal → p A Fal → p A s) →
  ∀ (A : context) (s : form), nd A s → p A s).
```

Note that the induction principle makes precise that we must prove a case for each rule in Figure 9.1 and what inductive hypotheses we obtain in each case. We can also visualize these four proof obligations in the form of rules.

$$\mathsf{A}\ \frac{}{p\ A\ s}\ s \in A \qquad \mathsf{II}\ \frac{A, s \vdash t}{(IH)\ p\ (A, s)\ t} \qquad \mathsf{IE}\ \frac{A \vdash s \to t \qquad A \vdash s}{(IH)\ p\ A\ (s \to t) \qquad (IH)\ p\ A\ s}$$

$$\mathsf{E}\ \frac{A \vdash \bot}{(IH)\ p\ A\ \bot}$$

In each case we must prove the conclusion using the premises (including the inductive hypotheses) and the side conditions as assumptions.

Our first such inductive proof will be of a property called **weakening**: if $A \subseteq A'$ and $A \vdash s$, then $A' \vdash s$. In other words, we prove the following rule weak in Figure 9.2 is admissible. Here the desired property of A and s is $\forall A', A \subseteq A' \to$

$A' \vdash s$. We prove by induction that A and s have this desired property whenever $A \vdash s$.

```
Lemma nd_weak A A' s :
  A ⊆ A' → nd A s → nd A' s.
```

We first give a mathematical proof.

Proof We argue by induction on the proof of $A \vdash s$ and consider each rule carefully. Future mathematical proofs will not be given at this level of detail.

Consider the assumption rule A:

$$\mathsf{A}\ \frac{}{A \vdash s}\ s \in A$$

Note that in this case $s \in A$. We must prove $\forall A', A \subseteq A' \to A' \vdash s$. Assume $A \subseteq A'$. Hence $s \in A'$ and so $A' \vdash s$ by the assumption rule.

Consider the introduction rule II for implication:

$$\mathsf{II}\ \frac{A, s \vdash t}{A \vdash s \to t}$$

We must prove $\forall A', A \subseteq A' \to A' \vdash s \to t$. The inductive hypothesis is $\forall A', A, s \subseteq A' \to A' \vdash t$. Assume $A \subseteq A'$. Clearly $A, s \subseteq A', s$. By the inductive hypothesis we know $A', s \vdash t$. Hence $A' \vdash s \to t$ by II.

Consider the elimination rule IE for implication:

$$\mathsf{IE}\ \frac{A \vdash s \to t \qquad A \vdash s}{A \vdash t}$$

We must prove $\forall A', A \subseteq A' \to A' \vdash t$. Since there are two premises, there are two inductive hypotheses. The first inductive hypothesis is $\forall A', A \subseteq A' \to A' \vdash s \to t$. The second inductive hypothesis is $\forall A', A \subseteq A' \to A' \vdash s$. Assume $A \subseteq A'$. By the inductive hypotheses we know $A' \vdash s \to t$ and $A' \vdash s$. Hence $A' \vdash t$ by IE.

Consider the elimination rule E for $\bot$:

$$\mathsf{E}\ \frac{A \vdash \bot}{A \vdash s}$$

We must prove $\forall A', A \subseteq A' \to A' \vdash s$. The inductive hypothesis is $\forall A', A \subseteq A' \to A' \vdash \bot$. Assume $A \subseteq A'$. By the inductive hypotheses we know $A' \vdash \bot$ and so $A' \vdash s$ by E. ∎

We now give the corresponding Coq proof script. The reader should step through the Coq proof and compare it with the mathematical proof above.

```
Proof.
  intros E F. revert A' E.
  induction F as [A s F|A s t F IHF|A s t F1 IHF1 F2 IHF2|A s F IHF] ; intros A' E.
  - apply ndA, E. assumption.
  - apply ndII, IHF. apply incl_shift, E.
  - apply ndIE with (s:=s) ; auto.
  - apply ndE, IHF, E.
Qed.
```

As an obvious corollary, we know that $A, t \vdash s$ whenever $A \vdash s$.

```
Lemma ndW A s t :
 nd A s → nd (t::A) s.

Proof. apply nd_weak; auto. Qed.
```

We could also obtain corollaries which combine weakening with the defining rules of the calculus. We will only do so for IE: If $B \vdash s \to t$, $B \subseteq A$ and $A \vdash s$, then $A \vdash t$.

```
Lemma ndIE_weak A B s t :
  nd B (Imp s t) → B ⊆ A → nd A s → nd A t.

Proof.
  intros E F G. apply ndIE with (s:=s).
  - exact (nd_weak F E).
  - exact G.
Qed.
```

We next prove lemmas that allow us to simulate Coq's *apply* tactic. If an implication $s \to t$ is in the context A and we want to prove $A \vdash t$, then it is enough to prove $A \vdash s$. This is the content of the following lemma. The proof is simply a combination of the IE and A rules.

```
Lemma ndapp A s u :
  Imp s u ∈ A → nd A s → nd A u.

Proof.
  intros E F. apply ndIE with (s := s).
  - apply ndA, E.
  - exact F.
Qed.
```

We also prove three simple corollaries for the cases when the implication is one of the three most recent assumptions. Using these corollaries avoids the need to prove trivial subgoals that would result from applying *ndapp*.

```
Lemma ndapp1 A s u :
  nd (Imp s u :: A) s → nd (Imp s u :: A) u.

Proof. apply ndapp. left. reflexivity. Qed.
```

```
Lemma ndapp2 A s t u :
  nd (t :: Imp s u :: A) s → nd (t :: Imp s u :: A) u.
Proof. apply ndapp. right. left. reflexivity. Qed.
Lemma ndapp3 A s t u v :
  nd (t :: v :: Imp s u :: A) s → nd (t :: v :: Imp s u :: A) u.
Proof. apply ndapp. right. right. left. reflexivity. Qed.
```

Finally we prove that if $A \vdash s$, then $A \vdash \neg\neg s$.

```
Lemma ndDN A s :
  nd A s → nd A (Not (Not s)).
Proof. intros E. apply ndII, ndapp1, ndW, E. Qed.
```

It turns out that $A \vdash s$ is decidable, but we do not yet have the tools to prove this. The decidability proof will come later.

Exercise 9.2.2 Extend the formulas and natural deduction system to include conjunction and disjunction.

Exercise 9.2.3 Prove the following goals.

```
Goal ∀ A s, nd A (Imp s s).
Goal ∀ A s, nd A (Imp Fal s).
Goal ∀ A s t, nd A (Imp s (Imp t s)).
Goal ∀ A s t, nd A (Imp (Imp s t) (Imp (Not t) (Not s))).
```

Exercise 9.2.4 Prove the following two lemmas.

```
Lemma ndassert (A : context) (s u : form) :
 nd A s → nd (s::A) u → nd A u.
Lemma ndappbin (A : context) (s t u : form) :
 Imp s (Imp t u) ∈ A → nd A s → nd A t → nd A u.
```

Exercise 9.2.5 Prove the following result.

```
Lemma nd_eval_sound A s (e:form → Prop) :
 ¬e Fal → (∀ t u, e (Imp t u) ↔ e t → e u) →
 nd A s → (∀ t, t ∈ A → e t) → e s.
```

Exercise 9.2.6 Use the results of Exercises 9.1.2 and 9.2.5 to prove the following.

```
Goal ¬ nd nil Fal.
```

$$\mathsf{A}\ \frac{}{A \vdash s}\ s \in A \qquad \mathsf{II}\ \frac{A, s \vdash t}{A \vdash s \to t} \qquad \mathsf{IE}\ \frac{A \vdash s \to t \qquad A \vdash s}{A \vdash t} \qquad \mathsf{C}\ \frac{A, \neg s \vdash \bot}{A \vdash s}$$

Figure 9.3: Classical Natural Deduction Rules

9.3 Classical Natural Deduction

We now consider classical propositional logic. Classical propositional logic can prove formulas such as instances of double negation $\neg\neg s \to s$ and instances of Peirce's law $((s \to t) \to s) \to s$. The propositional formulas provable in classical propositional logic correspond to those which evaluate to true under every boolean assignment, if one interprets $\bot$ as *false* and interprets implication by truth tables (i.e., *implb* in the Coq library).

The classical natural deduction system $\mathcal{N}_C$ is defined by the rules in Figure 9.3. Note that the difference from the previous system is that the elimination rule E for $\bot$ has been replaced by the **contradiction rule** C. In Coq, the classical natural deduction system can be defined as the following inductive predicate *ndc*.

```
Inductive ndc (A : context) : form → Prop :=
| ndcA s :    s ∈ A → ndc A s
| ndcII s t :  ndc (s::A) t → ndc A (Imp s t)
| ndcIE s t :  ndc A (Imp s t) → ndc A s → ndc A t
| ndcC s :     ndc (Not s :: A) Fal → ndc A s.
```

As before, we can prove lemmas for the special cases of the assumption rule for which the assumption is recent. In this case, we only prove $A, s \vdash s$.

```
Lemma ndcA1 A s :
 ndc (s :: A) s.
```

```
Proof. apply ndcA. left. reflexivity. Qed.
```

As before, this lemma means the rule A1 from Figure 9.2 is admissible. One can also prove the other rules in Figure 9.2 are admissible in the classical system $\mathcal{N}_C$. We will only prove weak and W from Figure 9.2 are admissible and leave the others to the reader.

We next prove weakening (weak) is admissible in $\mathcal{N}_C$. That is, we prove a weakening lemma by induction on the proof of the inductive predicate *ndc*.

```
Lemma ndc_weak A A' s :
  A ⊆ A' → ndc A s → ndc A' s.
```

```
Proof.
```

```
  intros E F. revert A' E.
  induction F as [A s F|A s t F IHF|A s t F1 IHF1 F2 IHF2|A s F IHF] ; intros A' E.
  - apply ndcA, E. assumption.
  - apply ndcII, IHF. apply incl_shift, E.
  - apply ndcIE with (s:=s) ; auto.
  - apply ndcC, IHF. intros t [G|G].
    + rewrite G. left. reflexivity.
    + right. apply E. exact G.
Qed.
```

We again obtain as a special case that $A, t \vdash s$ whenever $A \vdash s$. That is, W is admissible in $\mathcal{N}_C$.

```
Lemma ndcW A s t :
 ndc A s → ndc (t::A) s.
Proof. apply ndc_weak; auto. Qed.
```

Since we have omitted the elimination rule for $\bot$, a natural question is whether we can infer $A \vdash s$ from $A \vdash \bot$. That is, one may ask if the E rule (a defining rule for $\mathcal{N}$) is admissible in the system $\mathcal{N}_C$. We can prove admissibility of E in $\mathcal{N}_C$ easily using the contradiction rule and weakening.

```
Lemma ndcE A s :
  ndc A Fal → ndc A s.
Proof. intros E. apply ndcC, ndcW, E. Qed.
```

Now we have enough information to know $A \vdash s$ in $\mathcal{N}$ implies $A \vdash s$ in $\mathcal{N}_C$. The proof is by a simple induction on the proof of $A \vdash s$ in $\mathcal{N}$.

```
Lemma nd_ndc A s :
  nd A s → ndc A s.
Proof.
  intros F ; induction F as [A s F|A s t F IHF|A s t F1 IHF1 F2 IHF2|A s F IHF].
  - apply ndcA. assumption.
  - apply ndcII, IHF.
  - apply ndcIE with (s:=s) ; auto.
  - apply ndcE, IHF.
Qed.
```

Finally we prove $A \vdash s$ if and only if $A, \neg s \vdash \bot$.

```
Lemma ndc_refute A s :
  ndc A s ↔ ndc (Not s :: A) Fal.
Proof.
  split ; intros E.
  - apply ndcIE with (s:=s).
    + apply ndcA1.
    + apply ndcW, E.
  - apply ndcC, E.
Qed.
```

Exercise 9.3.1 Prove $A \vdash \neg\neg s \to s$.

```
Goal ∀ A s, ndc A (Imp (Not (Not s)) s).
```

Exercise 9.3.2 Prove the following lemmas for $\mathcal{N}_C$.

```
Lemma ndcA2 A s t :
 ndc (t :: s :: A) s.

Lemma ndcapp A s u :
  Imp s u ∈ A → ndc A s → ndc A u.

Lemma ndcapp1 A s u :
  ndc (Imp s u :: A) s → ndc (Imp s u :: A) u.

Lemma ndcapp2 A s t u :
  ndc (t :: Imp s u :: A) s → ndc (t :: Imp s u :: A) u.

Lemma ndcapp3 A s t u v :
  ndc (t :: v :: Imp s u :: A) s → ndc (t :: v :: Imp s u :: A) u.
```

Use the lemmas above to prove $A \vdash ((s \to t) \to s) \to s$. That is, prove Peirce's Law.

```
Goal ∀ A s t, ndc A (Imp (Imp (Imp s t) s) s).
```

Exercise 9.3.3 Consider the following implemention of truth table semantics.

```
Fixpoint valb (s:form) (p:var → bool) : bool :=
match s with
| Var x ⇒ p x
| Fal ⇒ false
| Imp s t ⇒ implb (valb s p) (valb t p)
end.
```

Prove the following result.

```
Lemma ndc_valb_sound A s p :
 ndc A s → (∀ t, t ∈ A → valb t p = true) → valb s p = true.
```

Exercise 9.3.4 Prove the following result.

```
Lemma ndc_eval_xm_sound A s (e:form → Prop) :
 XM →
 ¬e Fal → (∀ t u, e (Imp t u) ↔ e t → e u) →
 ndc A s → (∀ t, t ∈ A → e t) → e s.
```

9.4 Glivenko's Theorem

Glivenko's Theorem states that a propositional formula s is classically provable if and only if its double negation is intuitionistically provable. The most interesting half of this equivalence is that $\neg\neg s$ is intuitionistically provable if s is classically provable. In particular, if $A \vdash s$, then $A \vdash \neg\neg s$. We prove this implication by induction on the proof of $A \vdash s$. We leave the converse implication as an exercise.

```
Lemma Glivenko A s :
  ndc A s → nd A (Not (Not s)).
Proof.
  intros E. induction E as [A s E|A s t E IHE|A s t E1 IHE1 E2 IHE2|A s E IHE].
  - apply ndDN, ndA. assumption.
  - apply ndII, ndapp1.
    apply ndII, ndE. apply (ndIE_weak IHE).
    + auto.
    + apply ndII. apply ndapp3. apply ndII. apply ndA2.
  - apply ndII. apply (ndIE_weak IHE2).
    + auto.
    + apply ndII. apply (ndIE_weak IHE1).
      * auto.
      * apply ndII. apply ndapp3. apply ndapp1. apply ndA2.
  - apply ndII. apply (ndIE IHE). apply ndII, ndA1.
Qed.
```

As a consequence of Glivenko's theorem, we can prove that refutability in $\mathcal{N}$ is equivalent to refutability in $\mathcal{N}_C$.

```
Corollary Glivenko_refute A :
  nd A Fal ↔ ndc A Fal.
Proof.
  split.
  - apply nd_ndc.
  - intros E. apply Glivenko in E. apply (ndIE E). apply ndII, ndA1.
Qed.
```

A further consequence is that $A \vdash s$ in $\mathcal{N}_C$ if and only if $A, \neg s \vdash \bot$ in $\mathcal{N}$.

```
Corollary nd_embeds_ndc A s :
  ndc A s ↔ nd (Not s :: A) Fal.
Proof. setoid_rewrite ndc_refute. symmetry. apply Glivenko_refute. Qed.
```

Exercise 9.4.1 Prove the easy half of Glivenko's theorem.

```
Lemma Glivenko_converse A s :
  nd A (Not (Not s)) → ndc A s.
```

$$\mathsf{A}\ \frac{}{s}\ s \in A \qquad \mathsf{K}\ \frac{}{s \to t \to s} \qquad \mathsf{S}\ \frac{}{(s \to t \to u) \to (s \to t) \to s \to u} \qquad \mathsf{E}\ \frac{}{\bot \to u}$$

$$\mathsf{MP}\ \frac{s \to t \qquad s}{t}$$

Figure 9.4: Hilbert Rules for Intuitionistic Propositional Logic

Exercise 9.4.2 Prove the following consequence of Glivenko's theorem.

Goal ∀ A, ¬ ∃ s, ndc A (Not s) ∧ ¬ nd A (Not s).

9.5 Hilbert System

The natural deduction systems require assumption management. In particular the implication introduction rule II changes the assumptions. It turns out that we can omit the implication introduction rule if we replace it with a number of **initial rules** – i.e., rules with no premises. One initial rule states that every formula of the form $s \to t \to s$ is provable. We call such a formula a K-formula.

```
Definition FK (s t : form) : form :=
  Imp s (Imp t s).
```

Another initial rule states that every formula $(s \to t \to u) \to (s \to t) \to s \to u$ is provable. Such formulas are called S-formulas.

```
Definition FS (s t u : form) : form :=
  (Imp (Imp s (Imp t u))
       (Imp (Imp s t)
            (Imp s u))).
```

Doing this would yield a system in which only two rules have premises: a rule like IE and a rule like E. Indeed we can define a system in which the only rule with premises is a rule known as **modus ponens** which has the same form as IE since we can replace the E rule with an initial rule stating that every explosion formula is provable. Such systems are called **Hilbert systems**. The rules in Figure 9.4 define our **Hilbert system for intuitionistic propositional logic**, which refer to by the name $\mathcal{H}$. In particular, we have $A \vdash s$ in system $\mathcal{H}$ when s is derivable from context A using the rules in Figure 9.4.

We can define this in Coq as an inductive predicate *hil*.

```
Inductive hil (A : context) : form → Prop :=
```

```
| hilA s :       s ∈ A → hil A s
| hilK s t :     hil A (FK s t)
| hilS s t u :   hil A (FS s t u)
| hilE s :       hil A (Imp Fal s)
| hilMP s t :    hil A (Imp s t) → hil A s → hil A t.
```

Using the lemmas from the previous section, we can easily prove by induction (on proof terms) that if $A \vdash s$ in $\mathcal{H}$, then $A \vdash s$ in $\mathcal{N}$. We first give the mathematical proof.

Proof We argue by induction on the proof of $A \vdash s$ in $\mathcal{H}$. We must argue a case for each rule in Figure 9.4. If $s \in A$, then we know $A \vdash s$ in $\mathcal{N}$ by A. The next three cases involve proving K-formulas, S-formulas and formulas of the form $\bot \to s$ in $\mathcal{N}$. Each of these cases is easy. Finally, we consider the modus ponens case. Assume $A \vdash s \to t$ and $A \vdash s$ in $\mathcal{H}$. The inductive hypotheses yield $A \vdash s \to t$ and $A \vdash s$ in $\mathcal{N}$. We conclude $A \vdash t$ in $\mathcal{N}$ using IE. ∎

Note that in each case of the inductive proof, we have proven one of the defining rules of $\mathcal{H}$ is admissible in $\mathcal{N}$. Each case is easy. Here is the proof as a Coq proof script.

```
Lemma hil_nd A s :
  hil A s → nd A s.
Proof.
  intros E. induction E as [s E|s t|s t u|s|s t E1 IHE1 E2 IHE2].
  - apply ndA. assumption.
  - apply ndII, ndII, ndA2.
  - apply ndII, ndII, ndII.
    apply ndIE with (s:= Imp t u).
    + apply ndII. apply ndapp1. apply ndapp3. apply ndA2.
    + apply ndapp3, ndA1.
  - apply ndII. apply ndE, ndA1.
  - exact (ndIE IHE1 IHE2).
Qed.
```

The converse also holds: If $A \vdash s$ in $\mathcal{N}$, then $A \vdash s$ in $\mathcal{H}$. In order to prove this, we must prove that each of the defining rules for $\mathcal{N}$ is admissible in $\mathcal{H}$. Before we can prove this, we need a few preliminary results.

We can combine the initial rule for K-formulas with modus ponens to obtain the following result.

```
Lemma hilAK A s t :
  hil A s → hil A (Imp t s).
Proof. apply hilMP. apply hilK. Qed.
```

We can similarly combine the initial rule for S-formulas with modus ponens.

```
Lemma hilAS A s t u :
  hil  A (Imp s (Imp t u)) → hil A (Imp s t) → hil A (Imp s u).
Proof. intros B. apply hilMP. revert B. apply hilMP. apply hilS. Qed.
```

We can also use a combination of the initial rules to prove formulas of the form $s \to s$.

```
Lemma hilI A s :
  hil  A (Imp s s).
Proof.
  assert (E:= hilS A s (Imp s s) s).
  assert (F:= hilK A s (Imp s s)).
  assert (G:= hilK A s s).
  unfold FS, FK in *.
  exact (hilMP (hilMP E F) G).
Qed.
```

We can now prove an important result called **the deduction theorem**. The deduction theorem states that if $A, s \vdash t$, then $A \vdash s \to t$. In other words, the II rule (a defining rule of the system $\mathcal{N}$) is admissible in $\mathcal{H}$. The proof is by induction on the proof of $A, s \vdash t$ using the results above.

```
Lemma hilD s A t :
  hil  (s::A) t → hil A (Imp s t).
Proof.
  intros E. induction E as [t E|t u|t u v|t|t u E1 IHE1 E2 IHE2].
  - destruct E as [E|E].
    + subst t. apply hilI.
    + apply hilAK. apply hilA, E.
  - apply hilAK, hilK.
  - apply hilAK, hilS.
  - apply hilAK, hilE.
  - apply hilAS with (t:=t) ; assumption.
Qed.
```

We can now prove $A \vdash s$ in $\mathcal{N}$ implies $A \vdash s$ in $\mathcal{H}$. The proof is by induction on the proof of $A \vdash s$ in $\mathcal{N}$. The deduction theorem is used for the II-case. The remaining cases are straightforward.

```
Lemma nd_hil A s :
  nd A s → hil A s.
Proof.
  intros E. induction E as [A s E|A s t E IHE|A s t E1 IHE1 E2 IHE2|A s E IHE].
  - apply hilA. assumption.
  - apply hilD, IHE.
  - exact (hilMP IHE1 IHE2).
  - exact (hilMP (hilE A s) IHE).
Qed.
```

Combining the results we know $A \vdash s$ in $\mathcal{H}$ if and only if $A \vdash s$ in $\mathcal{N}$.

```
Theorem hil_iff_nd A s :
  hil A s ↔ nd A s.
Proof. split. apply hil_nd. apply nd_hil. Qed.
```

Exercise 9.5.1 Prove the following form of weakening for the Hilbert calculus.

```
Lemma hilW A s t :
  hil A t → hil (s::A) t.
```

Exercise 9.5.2 Prove the following.

```
Lemma hilassert A s u :
  hil A s → hil (s::A) u → hil A u.
```

Exercise 9.5.3 Give a Hilbert calculus for classical propositional logic and define a corresponding inductive predicate *hilc* in Coq. Prove the deduction theorem for *hilc* and use the deduction theorem to prove the equivalence between *hilc* and *ndc*.

Exercise 9.5.4 Define a substitution operation on formulas.

```
Fixpoint subst (theta : var → form) (s : form) : form :=
match s with
| Var x ⇒ theta x
| Imp s t ⇒ Imp (subst theta s) (subst theta t)
| Fal ⇒ Fal
end.
```

Prove the following.

```
Lemma nd_subst A s theta : nd A s → nd (map (subst theta) A) (subst theta s).
Lemma ndc_subst A s theta : ndc A s → ndc (map (subst theta) A) (subst theta s).
Lemma hil_subst A s theta : hil A s → hil (map (subst theta) A) (subst theta s).
```

9.6 Properties of Proof Systems

The provability predicates $\vdash$ for all the proof systems we have considered in this chapter satisfy a number of key properties.

- **Assumption** If $s \in A$, then $A \vdash s$.
  This is part of the definition of all of the proof systems we considered.
- **Modus Ponens** If $A \vdash s \to t$ and $A \vdash s$, then $A \vdash t$.
  This is also part of the definition of all of the proof systems we considered.

- **Deductivity** If $A, s \vdash t$, then $A \vdash s \to t$.
  This is the implication introduction rule for the natural deduction systems. For Hilbert systems, this is the deduction theorem.
- **Explosivity** If $A \vdash \bot$, then $A \vdash s$.
  This is true in each of the proof systems, but for different reasons in each case. It is a defining rule of the intuitionistic natural deduction system. It is an admissible rule of the classical natural deduction system. In the Hilbert system it follows from the explosion axiom with the modus ponens rule.
- **Weakening** If $A \vdash s$ and $A \subseteq A'$, then $A' \vdash s$.
  This is an admissible rule for each of the systems. We did not explicitly prove it is admissible for the Hilbert system, but this follows form the equivalence between the Hilbert system and the natural deduction system combined with weakening for the natural deduction system. Alternatively, one can easily prove weakening directly for the Hilbert system by an easy induction.
- **Substitutivity** Consider a substitution operation defined as
  $$\begin{array}{rcl} \hat{\theta}x & = & \theta x \\ \hat{\theta}(s \to t) & = & \hat{\theta}s \to \hat{\theta}t \\ \hat{\theta}\bot & = & \bot \end{array}$$
  for functions $\theta : \mathit{var} \to \mathit{form}$. The provability predicate satisfies **substitutivity** if $A \vdash s$ implies $\hat{\theta}A \vdash \hat{\theta}s$. This property can be easily proven for each of our proof systems by induction. (See Exercise 9.5.4.)
- **Consistency** $\bot$ is not provable from the empty context: $\nvdash \bot$.
  Consistency follows from some of the exercises in this chapter. In particular, see Exercises 9.2.6 and 9.3.3.

Note that deductivity, assumption, weakening and modus ponens together imply $A, s \vdash t$ if and only if $A \vdash s \to t$. The turnstile can be seen as an external implication which corresponds to the internal implication in propositional formulas.

Let $\vdash$ be the provability predicate satisfying the properties above. By induction using the first four properties, it is clear that if $\mathit{nd}\ A\ s$, then $A \vdash s$. Hence every provability predicate satisfying the properties above must at least include the provability predicate for intuitionistic propositional logic.

Classical propositional logic demonstrates that there are provability predicates satisfying the properties which include $A \vdash s$ where s is not intuitionistically provable from A. In particular, $\vdash \neg\neg x \to x$ in $\mathcal{N}_C$, but not in $\mathcal{N}$.

Classical propositional logic provides the largest provability predicate satisfying the properties above. Suppose $\vdash$ is a provability predicate satisfying the properties. We can prove if $A \vdash s$, then $\mathit{ndc}\ A\ s$ by induction on the number of variables which occur in A and s. Note that the number of variables which occur can be decreased using a substitution which maps a variable x to either $\bot$ or

$\bot \to \bot$.

There are also provability predicates satisfying the properties which are between intuitionistic propositional logic and classical propositional logic. An example of such a logic between the two can be obtained by adding the following rule to the rules defining $\mathcal{N}$:

$$\frac{A, \neg s \vdash t \qquad A, \neg\neg s \vdash t}{A \vdash t}$$

In Coq, the definitions above look as follows.

```
Definition Assumption (p:context → form → Prop) :=
forall A s, s ∈ A → p A s.

Definition ModusPonens (p:context → form → Prop) :=
forall A s t, p A (Imp s t) → p A s → p A t.

Definition Deductivity (p:context → form → Prop) :=
forall A s t, p (s::A) t → p A (Imp s t).

Definition Explosivity (p:context → form → Prop) :=
forall A s, p A Fal → p A s.

Definition Weakening (p:context → form → Prop) :=
forall A A' s, A ⊆ A' → p A s → p A' s.

Definition Substitutivity (p:context → form → Prop) :=
forall A s theta, p A s → p (map (subst theta) A) (subst theta s).

Definition Consistency (p:context → form → Prop) :=
¬p nil Fal.

Definition ProvPred (p:context → form → Prop) :=
Assumption p ∧ ModusPonens p ∧ Deductivity p ∧ Explosivity p
 ∧ Weakening p ∧ Substitutivity p ∧ Consistency p.
```

Exercise 9.6.1 Prove *nd* is the least provability predicate by first proving *nd* is a provability predicate and then proving it is contained by every other provability predicate.

```
Lemma ProvPred_nd : ProvPred nd.

Lemma nd_min (p:context → form → Prop) :
 ProvPred p →
∀ A s, nd A s → p A s.
```

Exercise 9.6.2 We say a formula is **closed** if it contains no variables. We can define this in Coq as follows.

```
Inductive closed : form → Prop :=
| closedFal : closed Fal
| closedImp s t : closed s → closed t → closed (Imp s t).
```

Prove that for all provability predicates $\vdash$ and all closed formulas s we either have $A \vdash s$ (for every context A) or $A \vdash \neg s$ (for every context A).

```
Lemma ProvPred_closed_or p s :
 ProvPred p → closed s → (∀ A, p A s) ∨ (∀ A, p A (Not s)).
```

9.7 Remarks

The first deduction systems developed by Frege in 1879 were in the Hilbert style. (Hilbert studied and popularized such systems later.) Natural deduction systems were created independently by Gentzen and Jaśkowski in 1934.

10 Proof Terms and Type Theory

In this chapter we give a variant of the natural deduction systems with proof terms. We define proof terms as a certain class of terms using, in particular, λ-abstraction and application. The main judgment of the modified natural deduction system is $A \vdash d : s$ which means that d proves s in context A. If we view the propositions as types, then the system with proof terms is basic simple type theory. From this point of view, $A \vdash d : s$ means d has type s in context A.

We first informally consider the intuitionistic case. In order to make the definitions precise, we define a way of identifying assumptions by a natural number. We then give a formal definition of proof terms d and the judgment $A \vdash d : s$. We prove that $A \vdash s$ is provable in $\mathcal{N}$ if and only if there is a proof term d such that $A \vdash d : s$. We also prove that a proof term proves at most one proposition (in a given context). In type theoretic terminology, this means we have uniqueness of types. Furthermore, we prove that given a context A and a proof term d, one can either compute a formula s such that $A \vdash d : s$ or compute a proof that there is no such formula s. A consequence is that the judgment $A \vdash d : s$ is decidable. In type theoretic terminology, this means that type checking is decidable.

The same program can be realized in the classical case. We sketch this, leaving some details as exercises.

10.1 Proof Terms, Informally

We consider a natural deduction system with proof terms. In order to motivate proof terms, let us reconsider the proof terms for certain propositions in Coq. Suppose X, Y : *Prop*. In order to prove $X \to Y$ in Coq, we may assume $x : X$ and construct a term D of type Y under this assumption. In this case, the proof term of $X \to Y$ will be $\lambda x : X.D$. On the other hand, if we have a term D of type $X \to Y$ and a term E of type X, then $D\ E$ is a term of type Y. In this way, λ-abstraction and application give proof terms corresponding to the introduction and elimination rules for implication. For the elimination rule for $\bot$, recall that if D is of type *False*, then *match D return X with end* is of type X. We will use a proof term constructor **E** corresponding to such a *match* in Coq.

Proof terms are given by the following grammar where x ranges over variables (natural numbers), s ranges over propositional formulas, and d and e range

$$\mathsf{A}\ \frac{}{A \vdash x : s}\ x : s \in A \qquad \mathsf{II}\ \frac{A, x : s \vdash d : t}{A \vdash \lambda x : s.d : s \to t} \qquad \mathsf{IE}\ \frac{A \vdash d : s \to t \qquad A \vdash e : s}{A \vdash de : t}$$

$$\mathsf{E}\ \frac{A \vdash d : \bot}{A \vdash \mathbf{E}ds : s}$$

Figure 10.1: Rules for Natural Deduction with Proof Terms with Named Assumptions

over proof terms.

$$d, e ::= x \mid de \mid \lambda x : s.e \mid \mathbf{E}ds$$

The four cases in the grammar correspond to the four rules defining the natural deduction system $\mathcal{N}$.

Let us temporarily assume our contexts A are of the form $x_0 : s_0, \ldots, x_{n-1} : s_{n-1}$. That is, variables give names to assumptions in the context. Under this assumption we can give a first version of a natural deduction system with proof terms as the rules in Figure 10.1. Following the propositions as types principle we can consider propositional formulas to be types. The proof terms are essentially simply typed lambda terms. From this point of view we have also defined a type theory we call **basic simple type theory**. This type theory is a small fragment of the type theory of Coq.

Example 10.1.1 Reconsider the natural deduction derivation of $A \vdash s \to \neg s \to t$ from Example 9.2.1.

$$\mathsf{II}\ \frac{\mathsf{II}\ \dfrac{\mathsf{E}\ \dfrac{\mathsf{IE}\ \dfrac{\mathsf{A}\ \dfrac{}{A, s, \neg s \vdash s \to \bot} \qquad \mathsf{A}\ \dfrac{}{A, s, \neg s \vdash s}}{A, s, \neg s \vdash \bot}}{A, s, \neg s \vdash t}}{A, s \vdash \neg s \to t}}{A \vdash s \to \neg s \to t}$$

Let us assume the assumptions in A have been associated with variables. (This will be made precise in the next section.) By examining the proof rules used above, one can easily see that the corresponding proof term is $\lambda x : s.\lambda y : \neg s.\mathbf{E}\ (y\ x)\ t$. Here is a derivation of

$$A \vdash (\lambda x : s.\lambda y : \neg s.\mathbf{E}\ (y\ x)\ t) : (s \to \neg s \to t)$$

using the rules in Figure 10.3.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{A, x : s, y : \neg s \vdash y : s \to \bot}\,\mathsf{A} \quad \dfrac{}{A, x : s, y : \neg s \vdash x : s}\,\mathsf{A}}{A, x : s, y : \neg s \vdash y\ x : \bot}\,\mathsf{IE}}{A, x : s, y : \neg s \vdash (\mathbf{E}\ (y\ x)\ t) : t}\,\mathsf{E}}{A, x : s \vdash (\lambda y : \neg s.\mathbf{E}\ (y\ x)\ t) : \neg s \to t}\,\mathsf{II}}{A \vdash (\lambda x : s.\lambda y : \neg s.\mathbf{E}\ (y\ x)\ t) : s \to \neg s \to t}\,\mathsf{II}$$

Exercise 10.1.2 Consider the following questions.

a) If d is of the form $\lambda x : s.e$, then which of the rules in Figure 10.3 could possibly justify $A \vdash d : t$?

b) If s is of the form $t \to u$, then which of the rules in Figure 10.3 could possibly justify $A \vdash d : s$?

c) Is it possible that $A \vdash d : s$ can be justified by two different rules in Figure 10.3?

d) Is it possible that $A \vdash (\lambda x : s.d) : s$?

10.2 Naming Assumptions

Before we can represent the judgment $A \vdash d : s$ in Coq, we must first determine how to represent contexts in which assumptions are named by variables. We can reasonably restrict to contexts in which the variables (i.e., natural numbers) come in the obvious order:

$$0 : s_0, \ldots, n-1 : s_{n-1}$$

In this case it is redundant to include the variables and hence we can continue to consider a context A to be a list of propositional formulas. In the A rule, the side condition $x : s \in A$ will mean that s is assumption x in A. Also, in the II rule, there will need to be a side condition that the variable x equals the length of A since the new assumption s must be assumption x in A, s.

We define the relation *s* **is assumption** *x* **in** *A* as an inductive predicate *assum*. We do this for a general type F which will be specialized to the type *form* of formulas in this chapter.

```
Section Assum.
  Variable F : Type.
  Implicit Types A : list F.
```

$$\frac{}{assum\ s\ n\ (s :: A)}\ n = |A| \qquad \frac{assum\ s\ n\ A}{assum\ s\ n\ (t :: A)}$$

Figure 10.2: Rules for identifying assumptions

The rules defining *assum* are given in Figure 10.2. The Coq definitions are as follows.

```
Inductive assum (s : F) (n : nat) : list F → Prop :=
  | assumB A : n = |A| → assum s n (s::A)
  | assumS t A : assum s n A → assum s n (t::A).
```

Note that $assum\ s\ n\ A$ holds when s is in the n^{th} position of the *reverse* of A.

An easy induction on lists verifies that an s is a member of a list A if and only if there is some n such that $assum\ s\ n\ A$.

```
Lemma in_assum s A :
  s ∈ A ↔ ∃ n, assum s n A.
Proof.
  split.
  – intros D. induction A as [|t A].
    + contradiction D.
    + destruct D as [D|D].
      * subst t. ∃ (|A|). left. reflexivity.
      * destruct (IHA D) as [n E]. ∃ n. right. exact E.
  – intros [n D]. induction D as [A'|y A'] ; auto.
Qed.
```

We also have the following inversion principle on $assum\ s\ n\ A$: If s is assumption n in A, then A must be nonempty and either s is the first element of the list with $n = |A|$ or s is assumption n in the rest of the list and $n < |A|$.

```
Lemma assum_inv s n A :
  assum s n A →
  match A with
  | (t::A) ⇒ s = t ∧ n = |A| ∨ assum s n A ∧ n < |A|
  | nil ⇒ False
  end.
Proof.
  intros B. induction B as [|t  [|u A] B IHB].
  – tauto.
  – contradiction IHB.
  – simpl. right. split.
    + assumption.
```

```
    + destruct IHB as [[C D]|[C D]]; omega.
Qed.
```

Using the inversion principle above, we can prove that there is at most one assumption n of A.

```
Lemma assum_uniq s t n A :
 assum s n A → assum t n A → s = t.
Proof.
  induction A as [|u A].
  − intros B. apply assum_inv in B. contradiction B.
  − intros B D. apply assum_inv in B as [[B C]|[B C]].
    + apply assum_inv in D as [[D E]|[D E]].
      * congruence.
      * omega.
    + apply assum_inv in D as [[D E]|[D E]].
      * omega.
      * tauto.
Qed.
```

Finally, there is a certifying function which, given n and A, can compute an s such that s is assumption n in A or yields a proof that there is no such s.

```
Lemma assum_sig n A :
 {s | assum s n A} + {∀ s, ¬ assum s n A}.
Proof.
  induction A as [|s A]; simpl.
  − right. intros s B. apply assum_inv in B. contradiction B.
  − destruct IHA as [[t B]|B].
    + left. ∃ t. apply assumS, B.
    + decide (n = |A|) as [C|C].
      * left. ∃ s. apply assumB, C.
      * { right. intros t D. apply assum_inv in D as [[D E]|[D E]].
          − tauto.
          − revert D. apply B.
        }
Qed.
```

Finally, we close the section and from now on use *assum* and the results above with F instantiated to *form*.

```
End Assum.
```

Exercise 10.2.1 Prove the following.

```
Goal ∀ s t:F, {n:nat|assum s n (t::s::nil)}.
Goal ∀ s t:F, {n:nat|assum t n (t::s::nil)}.
```

$$\mathsf{A}\ \frac{}{A \vdash x : s}\ \mathit{assum}\ s\ x\ A \qquad \Pi\ \frac{A, s \vdash d : t}{A \vdash \lambda x : s.d : s \to t}\ x = |A|$$

$$\mathsf{IE}\ \frac{A \vdash d : s \to t \qquad A \vdash e : s}{A \vdash de : t} \qquad \mathsf{E}\ \frac{A \vdash d : \bot}{A \vdash \mathbf{E}ds : s}$$

Figure 10.3: Rules for Natural Deduction with Proof Terms

Exercise 10.2.2 Prove that if equality on F is decidable, then *assum* is decidable. Then prove that if *assum* is decidable, then equality on F is decidable. Do not use the *inversion* tactic. (Hint: Use *assum_sig* and *assum_uniq* for one direction and use *assum_inv* for the other direction.)

Goal eq_dec F → ∀ s n A, dec (assum s n A).

Goal (∀ s n A, dec (assum s n A)) → eq_dec F.

10.3 Proof Terms, Formally

Now that we have a way of associating variables with assumptions in a context, we can refine the rules in Figure 10.1 to be the rules in Figure 10.3. The **natural deduction system with proof terms**, system $\mathcal{N}_\mathcal{P}$, is defined by the rules in Figure 10.3. That is, $A \vdash d : s$ in the system $\mathcal{N}_\mathcal{P}$ means derivability using the rules in Figure 10.3.

Example 10.3.1 Recall the derivation from Example 10.1.1:

$$\Pi\ \frac{\Pi\ \dfrac{\mathsf{E}\ \dfrac{\mathsf{IE}\ \dfrac{\mathsf{A}\ \dfrac{}{A, x : s, y : \neg s \vdash y : s \to \bot} \qquad \mathsf{A}\ \dfrac{}{A, x : s, y : \neg s \vdash x : s}}{A, x : s, y : \neg s \vdash y\ x : \bot}}{A, x : s, y : \neg s \vdash (\mathbf{E}\ (y\ x)\ t) : t}}{A, x : s \vdash (\lambda y : \neg s.\mathbf{E}\ (y\ x)\ t) : \neg s \to t}}{A \vdash (\lambda x : s.\lambda y : \neg s.\mathbf{E}\ (y\ x)\ t) : s \to \neg s \to t}$$

We give the derivation in $\mathcal{N}_\mathcal{P}$ using the rules in Figure 10.3. Let x be the length of A. Let y be $x + 1$, i.e., the length of $|A, s|$. The $\mathcal{N}_\mathcal{P}$-derivation is simply given by removing the variables x and y from the assumptions. Note that s is assumption

x of $A, s, \neg s$ and that $\neg s$ is assumption y of $A, s, \neg s$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{A, s, \neg s \vdash y : s \to \bot}\,\mathsf{A} \quad \dfrac{}{A, s, \neg s \vdash x : s}\,\mathsf{A}}{A, s, \neg s \vdash y\ x : \bot}\,\mathsf{IE}}{A, s, \neg s \vdash (\mathbf{E}\ (y\ x)\ t) : t}\,\mathsf{E}}{A, s \vdash (\lambda y : \neg s.\mathbf{E}\ (y\ x)\ t) : \neg s \to t}\,\mathsf{II}}{A \vdash (\lambda x : s.\lambda y : \neg s.\mathbf{E}\ (y\ x)\ t) : s \to \neg s \to t}\,\mathsf{II}$$

In Coq we use the inductive type *pf* to represent these proof terms.

```
Inductive pf : Type :=
| PVar : nat → pf
| Lam : nat → form → pf → pf
| Ap : pf → pf → pf
| Expl : pf → form → pf.
```

The corresponding inductive predicate in Coq, *ndp*, is defined as follows.

```
Inductive ndp (A : context) : pf → form → Prop :=
| ndpA n s : assum s n A → ndp A (PVar n) s
| ndpII n d s t : n = |A| → ndp (s::A) d t → ndp A (Lam n s d) (Imp s t)
| ndpIE d e s t : ndp A d (Imp s t) → ndp A e s → ndp A (Ap d e) t
| ndpE d s : ndp A d Fal → ndp A (Expl d s) s.
```

We can now show Example 10.3.1 as a Coq proof script.

```
Goal ∀ A s t, {d | ndp A d (Imp s (Imp (Not s) t))}.
Proof.
  intros A s t.
  pose (x := |A|).
  pose (y := S (|A|)).
  ∃ (Lam x s (Lam y (Not s) (Expl (Ap (PVar y) (PVar x)) t))).
  apply ndpII.
  - reflexivity.
  - apply ndpII.
    + reflexivity.
    + apply ndpE. apply ndpIE with (s := s).
      * apply ndpA. apply assumB. reflexivity.
      * apply ndpA. apply assumS. apply assumB. reflexivity.
Qed.
```

We prove $A \vdash s$ in $\mathcal{N}$ if and only if there is a proof term d such that $A \vdash d : s$ in $\mathcal{N}_\mathcal{P}$. We give the informal mathematical proof. The Coq proof script is available online.

```
Lemma nd_ndp A s :
 nd A s ↔ ∃ d, ndp A d s.
```

Proof We prove the two directions by induction. Proving *ndp A d s* implies *nd A s* is easy, since one simply ignores the proof term d and applies the corresponding natural deduction rule. We explain only the more interesting direction: *nd A s* implies $\exists d.ndp\ A\ d\ s$. Since the two judgments have a different number of arguments, we can, without ambiguity, write this as $A \vdash s \to \exists d.A \vdash d : s$. We argue by induction on $A \vdash s$. For each of the rules A, II, IE and E defining the intuitionistic natural deduction calculus, we will assume as inductive hypotheses that there are appropriate proof terms for the premises and we must construct an appropriate proof term for the conclusion.

Suppose $s \in A$. By *in_assum* there is some x such that s is assumption x of A. Hence we know $A \vdash x : s$.

Assume as an inductive hypothesis that there is a proof term d such that $A, s \vdash d : t$. Let x be the length of A. Note that s is assumption x of A, s. Hence $A \vdash (\lambda x : s.d) : s \to t$, as desired.

Assume as inductive hypotheses that there are proof terms d and e such that $A \vdash d : s \to t$ and $A \vdash e : s$. Clearly we have $A \vdash de : t$.

Finally, assume $A \vdash d : \bot$. Clearly $A \vdash \mathbf{E}\ d\ s : s$. ∎

We can prove the following inversion lemma. Essentially the inversion lemma says that if $A \vdash d : s$, then the form of d gives certain information about A and s. Note that we are using a *match* to form the proposition. The reader should carefully consider how this proposition simplifies for each kind of proof term d. The formal proof is by a simple case analysis on the assumption $A \vdash d : s$. The informal proof is simply by inspecting the rules in Figure 10.3.

```
Lemma ndp_inv A d s :
  ndp A d s →
  match d with
  | PVar x ⇒ assum s x A
  | Lam x t d =>
    match s with
    | Imp s1 s2 ⇒ x = |A| ∧ s1 = t ∧ ndp (t::A) d s2
    | _ ⇒ False
    end
  | Ap d e ⇒ exists t, ndp A d (Imp t s) ∧ ndp A e t
  | Expl d t ⇒ t = s ∧ ndp A d Fal
  end.
Proof.
  intros B. destruct B as [x s B|x d1 s1 s2 E B|d1 d2 s1 s2 B1 B2|s B].
  – assumption.
  – tauto.
  – ∃ s1. tauto.
  – tauto.
Qed.
```

Using the inversion lemma above, we can prove that, in a given context, a proof term proves at most one proposition. This property of a type theory is called **uniqueness of types**. We give the informal mathematical proof. The Coq proof script is available online.

```
Lemma ndp_uniq_typ A d s t :
 ndp A d s → ndp A d t → s = t.
```

Proof We prove by induction on $A \vdash d : s$ that for all t, if $A \vdash d : t$, then $s = t$.

Suppose d is a variable x and s is assumption x of A. Assume $A \vdash x : t$. Applying the inversion lemma to $A \vdash x : t$, we also know t must be assumption x of A. Since there is at most one assumption x of A (by *assum_inv*), $s = t$.

Suppose s is $s_1 \to s_2$, d is $\lambda x : s_1.d_1$, x is $|A|$ and $A, s_1 \vdash d_1 : s_2$. Assume $A \vdash (\lambda x : s_1.d_1) : t$. Applying the inversion lemma, we know t must be of the form $s_1 \to t_2$ where $A, s_1 \vdash d_1 : t_2$. By the inductive hypothesis, $s_2 = t_2$ and hence $s = t$.

Suppose d is $d_1 d_2$, $A \vdash d_1 : s_1 \to s$ and $A \vdash d_2 : s_1 \to s$. Assume $A \vdash d_1 d_2 : t$. By the inversion lemma there is some t_1 such that $A \vdash d_1 : t_1 \to t$ and $A \vdash d_2 : t_1$. By the inductive hypothesis $s_1 \to s$ is the same as $t_1 \to t$. In particular, $s = t$.

Suppose d is $\mathbf{E}\ d_1\ s$ and $A \vdash d_1 : \bot$. Assume $A \vdash \mathbf{E}\ d_1\ s : t$. By the inversion lemma, t must be s. (Note that the inductive hypothesis was not needed for this case.) ∎

In Coq proof scripts, *ndp_uniq_typ* can often profitably be used in combination with the tactic *discriminate*. Consider the following small example.

```
Goal ∀ A d x, ndp A d (Var x) → ¬ ndp A d Fal.
Proof.
  intros A d x B C. discriminate (ndp_uniq_typ B C).
Qed.
```

Of course, sometimes the equations proven by Coq terms of the form *ndp_uniq_typ B C* are not contradictory. In such a case *discriminate* will not help. A tactic which is useful when one wants to introduce propositions justified by a Coq proof term is *generalize*. Suppose we are trying to prove a goal with claim P and suppose D is a term with type Q. Using *generalize D* will change the claim to be $Q \to P$, so that we may afterwards make use of Q.

Consider the following small example in which we make use of the *generalize* tactic twice.

```
Goal ∀ A d e s1 t1 u1 s2 t2,
 ndp A d (Imp s1 t1) → ndp A e u1 →
 ndp A d (Imp s2 t2) → ndp A e s2 → s1 = u1.
```

```
Proof.
  intros A d e s1 t1 u1 s2 t2 B C D E.
  generalize (ndp_uniq_typ B D).
  generalize (ndp_uniq_typ C E).
  congruence.
Qed.
```

Before the first application of *generalize* above, the claim of the goal is $s_1 = u_1$. Since $B : ndp\ A\ d\ (Imp\ s_1\ t_1)$ and $D : ndp\ A\ d\ (Imp\ s_2\ t_2)$, the term $ndp\_uniq\_typ\ B\ D$ has type $(Imp\ s_1\ t_1) = (Imp\ s_2\ t_2)$. After applying *generalize* with this term, the claim of the goal is

$$(Imp\ s_1\ t_1) = (Imp\ s_2\ t_2) \to s_1 = u_1.$$

Likewise, the term $ndp\_uniq\_typ\ C\ E$ has type $u_1 = s_2$ and so after the second *generalize* the claim is

$$u_1 = s_2 \to (Imp\ s_1\ t_1) = (Imp\ s_2\ t_2) \to s_1 = u_1.$$

The final *congruence* does the necessary equational reasoning to finish the proof.

We next turn to the proof of decidability of $A \vdash d : s$. One might try to prove decidability by induction on the proof term d. Such an attempt will fail when attempting to argue the IE case because there is a formula s occurring in the premises of IE which do not occur in the conclusion. (The reader is encouraged to try such a proof in Coq and notice where the problem occurs.)

The key to proving decidability is to first construct a certifying function which synthesizes the formula a proof term proves or yields a proof that there is no such formula. We give the informal mathematical proof. The Coq proof script is available online. The Coq proof script for *ndp_synth* makes use of the tactic *generalize*.

```
Lemma ndp_synth A d :
 {s | ndp A d s} + {¬ ∃ s, ndp A d s}.
```

Proof The proof is by induction on the proof term d. (That is, the certifying function is defined by recursion over the structure of d.)

Assume d is a variable x. We know (using *assum_sig*) that there is either an assumption x of A or there is no such assumption. If s is an assumption x of A, then we have $A \vdash x : s$. If there is no such assumption, then using the inversion lemma we know $\neg\exists s.A \vdash x : s$.

Assume d is $\lambda x : s_1.e$. If x is not the length of A, then the inversion lemma implies $\neg\exists s.A \vdash (\lambda x : s_1.e) : s$. Assume x is $|A|$. By the inductive hypothesis, either there is some t such that $A, s_1 \vdash e : t$ or there is no such t. If there is no

such t, then the inversion lemma again implies $\neg\exists s.A \vdash (\lambda x : s_1.e) : s$. Assume $A, s_1 \vdash e : t$. Clearly $A \vdash (\lambda x : s_1.e) : s_1 \to t$.

Assume d is $d_1 d_2$. By the inductive hypothesis and the inversion lemma, it is enough to consider the case in which we have terms t and u such that $A \vdash d_1 : t$ and $A \vdash d_2 : u$. If t is not of the form $u \to s$, then the inversion lemma implies $\neg\exists s.A \vdash (d_1 d_2) : s$. Assume t is of the form $u \to s$. In this case, $A \vdash (d_1 d_2) : s$.

Assume d is $\mathbf{E}\ e\ s$. By the inductive hypothesis and the inversion lemma, it is enough to consider the case in which $A \vdash e : \bot$. In this case, $A \vdash \mathbf{E}\ e\ s : s$. ∎

Using the previous two results, decidability easily follows.

```
Goal ∀ A d s, dec (ndp A d s).
```

Proof Let A, d and s be given. Using *ndp_synth* we either obtain a term t such that $A \vdash d : t$ or we know there is no such t. If there is no such t, then we know $A \nvdash d : s$. Suppose t is such that $A \vdash d : t$. If t is s, then we have $A \vdash d : s$. If t is different from s, then uniqueness of types (*ndp_uniq_typ*) implies $A \nvdash d : s$. ∎

Here is the Coq proof script.

```
Proof.
  intros A d s. destruct (ndp_synth A d) as [[t B]|B].
  – decide (t = s) as [C|C].
    + subst. left. assumption.
    + right. intros D. apply C. apply (ndp_uniq_typ B D).
  – right. intros D. apply B. ∃ s. exact D.
Qed.
```

As remarked earlier, $A \vdash s$ in $\mathcal{N}$ is also decidable, but this is not as easy to prove as decidability of $A \vdash d : s$.

Exercise 10.3.2 Prove the following.

```
Goal ∀ s t, {d | ndp (t::s::nil) d s}.
Goal ∀ s u, {d | ndp (Imp s u::s::nil) d u}.
Goal ∀ s t u, {d | ndp (Imp s (Imp t u)::t::nil) d (Imp s u)}.
Goal ∀ s u, {d | ndp (Imp (Imp s s) u::nil) d u}.
```

Exercise 10.3.3 Prove the following.

```
Goal ∀ A s u d e, ndp A d s → ndp (s::A) e u → {d' | ndp A d' u}.
```

Exercise 10.3.4 Prove the following weakening result.

```
Lemma ndp_weak A A' d s : A ⊆ A' → ndp A d s → ∃ d', ndp A' d' s.
```

$$\mathsf{A}\ \frac{}{A \vdash x : s}\ x : s \in A \qquad \mathsf{II}\ \frac{A, x : s \vdash d : t}{A \vdash \lambda x : s.d : s \to t} \qquad \mathsf{IE}\ \frac{A \vdash d : s \to t \qquad e : A \vdash s}{A \vdash de : t}$$

$$\mathsf{E}\ \frac{A, x : \neg s \vdash d : \bot}{A \vdash \mathbf{C}x : s.d : s}$$

Figure 10.4: Rules for Classical Natural Deduction with Proof Terms

The simpler weakening property using the same proof term is not provable. In fact, one can prove there are contexts A and A', a proof term d and a formula s such that $A \subseteq A'$, $A \vdash d : s$ and $A' \nvdash d : s$. (This is a consequence of our way of referencing assumptions.) Prove this in Coq.

Goal ∃ A A' d s, A ⊆ A' ∧ ndp A d s ∧ ¬ndp A' d s.

10.4 Proof Terms for Classical Propositional Logic

The proof terms can be modified to give classical proof terms with easy modifications. **Classical proof terms** are given by the following grammar:

$$d, e ::= x \mid de \mid \lambda x : s.e \mid \mathbf{C}x : s.d$$

In Coq we represent these using an inductive type.

```
Inductive pfc : Type :=
| PVarc : nat → pfc
| Lamc : nat → form → pfc → pfc
| Apc : pfc → pfc → pfc
| Contrac : nat → form → pfc → pfc.
```

The classical rules defining $A \vdash d : s$ are given in Figure 10.4. Note that both λ and $\mathbf{C}$ act as binders. The corresponding Coq definition is *ndcp*.

```
Inductive ndcp (A : context) : pfc → form → Prop :=
| ndcpA n s : assum s n A → ndcp A (PVarc n) s
| ndcpII n d s t : n = |A| → ndcp (s::A) d t → ndcp A (Lamc n s d) (Imp s t)
| ndcpIE d e s t : ndcp A d (Imp s t) → ndcp A e s → ndcp A (Apc d e) t
| ndcpC d s n : n = |A| → ndcp (Not s::A) d Fal → ndcp A (Contrac n s d) s.
```

With minor modifications to the proofs for the intuitionistic case, one can prove the following results. We leave the proofs as exercises for the reader.

A formula s is classically provable from A if and only if there is a classical proof term d such that $A \vdash d : s$.

```
Lemma ndc_ndcp A s :
  ndc A s ↔ ∃ d, ndcp A d s.
```

A classical proof term proves at most one formula.

```
Lemma ndcp_uniq_typ A d s t :
  ndcp A d s → ndcp A d t → s = t.
```

There is a certifying function which, given A and d, will either compute a formula s such that $A \vdash d : s$ or give a proof that no such s exists.

```
Lemma ndcp_synth A d :
  {s | ndcp A d s} + {¬ ∃ s, ndcp A d s}.
```

Exercise 10.4.1 Prove the following.

```
Goal (∀ A s, {d | ndcp A d (Imp (Not (Not s)) s)}).
```

Exercise 10.4.2 Prove *ndc_ndcp*.

Exercise 10.4.3 Formulate an inversion lemma *ndcp_inv*, prove this inversion lemma, and then use the inversion lemma to prove *ndcp_uniq_typ* and *ndcp_synth*.

Exercise 10.4.4 Use *ndcp_synth* and *ndcp_uniq_typ* to prove the following decidability result.

```
Goal ∀ A d s, dec (ndcp A d s).
Proof.
  intros A d s. destruct (ndcp_synth A d) as [[t B]|B].
  - decide (t = s) as [C|C].
    + subst. left. assumption.
    + right. intros D. apply C. apply (ndcp_uniq_typ B D).
  - right. intros D. apply B. ∃ s. exact D.
Qed.
```

10.5 Remarks

The recognition that simply typed λ-terms can act as proof terms for propositional formulas is the origin of the propositions as types principle. This was first recognized by Curry and Howard. As discussed in Chapter 2, the correspondence extends to richer type theories in which the types are dependent and the propositions include quantifiers. The Coq system is based on such a rich type theory. The representation of bound variables we use to avoid putting names in the context is called de Bruijn levels. An alternative way to avoid putting names into the context is to use de Bruijn indices.

Coq Summary

New Tactics

- *generalize* takes a term of type s and changes the claim from t to $s \to t$. Examples are given in Section 10.3.

11 Classical Semantics

In this chapter we analyze the entailment relation of classical propositional logic with a semantic characterization. The semantic characterization complements the proof-theoretic characterizations we have seen so far. The semantic characterization will give us a method for obtaining non-provability results. The semantic characterization also provides the basis for a decision procedure for classical provability of propositional formulas.

11.1 Semantic Entailment and Soundness

We will define a predicate *sem* such that we can eventually prove

$$\forall A\ s.\ ndc\ A\ s \leftrightarrow sem\ A\ s$$

We call the predicate *sem* **semantic entailment**. The definition of *sem* constitutes a semantic characterization of classical propositional provability. In this section we will see the definition of *sem* and a proof of $\forall A\ s.\ ndc\ A\ s \to sem\ A\ s$. This direction of the equivalence result is known as **soundness**. The other direction $\forall A\ s.\ sem\ A\ s \to ndc\ A\ s$ of the equivalence result is known as **completeness**.

Completeness is much harder to establish than soundness. We will obtain completeness with a decision procedure for classical propositional provability.

We start with the definition of semantic entailment. The idea is to interpret formulas as propositions such that implication of formulas is interpreted as implication of propositions. Consider the formula $\neg\neg x \to x$. Interpreting the formula as the proposition

$$\forall X : Prop,\ \neg\neg X \to X$$

does not work since the formula is classically provable while the proposition is not provable in Coq. We can fix the problem by modeling the variable x of the formula with a boolean variable.

$$\forall x : bool,\ \neg\neg(x = true) \to x = true$$

This time the proposition is provable in Coq. It turns out that the boolean coding of variables works in general. The reason is that boolean equality is decidable and that decidable propositions behave classically.

We use **boolean assignments**

```
Definition assn := var → bool.
```

to provide boolean values for all variables at once. Given a boolean assignment, we can map every formula to a proposition.

```
Fixpoint satis (f : assn) (s : form) : Prop :=
  match s with
    | Var x ⇒ f x = true
    | Imp s1 s2 ⇒ satis f s1 → satis f s2
    | Fal ⇒ False
  end.
```

We read a proposition *satis f s* as "*f* **satisfies** *s*".

It is decidable whether an assignment satisfies a formula.

```
Instance eq_bool_dec x y :  dec (x = y :> bool).
Proof. unfold dec ; decide equality. Qed.
Instance satis_dec f s :  dec (satis f s).
Proof. induction s ; decide claim. Qed.
```

Note the notation for typed equality in the formulation of *eq_bool_dec*. The notation "$x = y :> X$" stands for the term *@eq X x y*.

We now define **semantic entailment**.

```
Definition sem (A : context) (s : form) : Prop :=
  ∀ f, (∀ t, t ∈ A → satis f t) → satis f s.
```

We read *sem A s* as "*A* **semantically entails** *s*". Our definition is such that *A* semantically entails *s* if and only if every assignment satisfying every assumption in *A* satisfies *s*.

We show that classical provability entails semantic entailment. Recall that this property is known as soundness.

```
Lemma ndc_sem A s :
  ndc A s → sem A s.
Proof.
  intros E f F.
  induction E as [A s E|A s t _ IH|A s t _ IHs _ IHt|A s _ IH] ; simpl in *.
  – apply F, E.
  – intros G. apply IH. intros u [[]|H]. exact G. apply F, H.
  – apply (IHs F), (IHt F).
  – decide (satis f s) as [G|G]. exact G.
    exfalso ; apply IH. intros t  [[]| H]. exact G. apply F, H.
Qed.
```

The proof shows that each proof rule preserves semantic entailment. For the soundness of the contradiction rule it is essential that propositions of the form

satis f s are decidable. Note the use of the tactic *simpl in* ∗, which simplifies the assumptions and the claim of the goal in one go. Step carefully through the proof with Coq to understand every detail.

The contraposition of soundness says that s is not provable from A if s is not semantically entailed by A.

```
Lemma contra_ndc_sem A s :
  ¬ sem A s → ¬ ndc A s.
Proof. auto using ndc_sem. Qed.
```

Using this fact it is easy to prove that classical provability is consistent (i.e., $\bot$ is not provable in the empty context).

```
Lemma ndc_consistent : ¬ ndc nil Fal.
Proof. apply contra_ndc_sem. intros D.  apply (D (fun x ⇒ true)). intros t []. Qed.
```

Exercise 11.1.1 Prove the following equivalences stating that implication of formulas is interpreted classically. We will refer to the equivalences as **decomposition equivalences for implications**.

```
Lemma satis_pos_impl f s t :
  satis  f  (Imp s t) ↔ ¬ satis f s ∨ satis f t.
Lemma satis_neg_impl f s t :
  ¬ satis f  (Imp s t) ↔ satis f s ∧ ¬ satis f t.
```

Can you say beforehand for each equivalence which direction of the proof requires the decidability of *satis*?

Exercise 11.1.2 Prove the following goal saying that negation of formulas is interpreted as negation in Coq.

```
Goal ∀ f s,  satis f (Not s) = ¬ satis f s.
```

Exercise 11.1.3 Prove the following goal. Do not use soundness.

```
Goal ∀ f x,  satis f (Imp (Not (Not (Var x))) (Var x)).
```

Exercise 11.1.4 Define a boolean evaluation function *evalb* : *assn* → *form* → *bool* and prove that it agrees with the evaluation predicate.

```
Lemma evalb_agree f s :  evalb f s = true ↔ satis f s.
```

Exercise 11.1.5 Show that variables are not provable in the empty context.

```
Lemma ndc_var x :  ¬ ndc nil (Var x).
```

a) Give a semantic proof using soundness.

b) Give a syntactic proof of consistency using *ndc_var*. That is, prove the proposition $\neg ndc\ nil\ Fal$ without using assignments.

Exercise 11.1.6 You will show that two assignments that agree on all variables of a formula map the formula to the same proposition. This property is known as *coincidence*.

a) Define a function $vars : form \to list\ var$ that for a formula yields a list containing exactly the variables occurring in the formula. The list may contain duplicates.

b) Prove the coincidence property.

```
Lemma coincidence f g s : (∀ x, x ∈ vars s → f x = g x) → satis f s = satis g s.
```

11.2 Clauses and Satisfiability

For the remaining results, which concern decidability and completeness, we will employ lists of signed formulas called **clauses**.[1] A **signed formula** is a pair of a sign and a formula, where a **sign** is either positive or negative.

```
Inductive sform : Type :=
| Pos : form → sform
| Neg : form → sform.

Notation "+ s" := (Pos s) (at level 35).
Notation "− s" := (Neg s).

Definition clause := list sform.
```

For a positively signed formula we write s^+ or simply s, and for a negatively signed formula we write s^-. We will speak of **positive** and **negative** formulas.

An **assignment satisfies a clause** if it satisfies every positive formula of the clause and dissatisfies every negative formula of the clause. A clause is **satisfiable** if it is satisfied by at least one assignment.

By our definitions an assignment dissatisfies a formula if and only if it satisfies the negation of the formula. Thus a negative formula s^- is semantically equivalent to the formula $\neg s$. While signs are redundant semantically, they will matter computationally.

In Coq, we define satisfiability of clauses based on an unsign function and a recursive satisfaction predicate for clauses.

```
Definition uns (S : sform) : form :=
  match S with +s ⇒ s | −s ⇒ Not s end.
```

[1] For experts: Clauses are like sequents in Gentzen systems. In a sequent the positive formulas would appear on the left and the negative formulas would appear on the right.

```
Fixpoint satis' (f : assn) (C : clause) : Prop :=
  match C with
    | nil ⇒ True
    | T::C' ⇒ satis f (uns T) ∧ satis' f C'
  end.

Definition sat (C : clause) := ∃ f, satis' f C.
```

The definitions are done such that they maximize conversion, which will ease many proofs.

We establish three characterizations of the satisfaction predicate for clauses.

```
Lemma satis_iff f C :
  satis' f C ↔ ∀ S, S ∈ C → satis f (uns S).

Lemma satis_uns f C :
  satis' f C ↔ ∀ s, s ∈ map uns C → satis f s.

Lemma satis_Pos f A :
  satis' f (map Pos A) ↔ ∀ s, s ∈ A → satis f s.
```

Each of the characterizations can be established with the following brute force script (replace C with A for the third characterization).

```
Proof. induction C ; simpl ; firstorder ; subst ; auto. Qed.
```

The following lemma specializing *satis_iff* will be useful in proofs.

```
Lemma satis_in f C S :
  satis' f C → S ∈ C → satis f (uns S).
```

Semantic entailment can be characterized as unsatisfiability of clauses.

```
Lemma sem_unsat_iff A s :
  sem A s ↔ ¬ sat (−s :: map Pos A).

Proof.
  split.
  − intros D [f [E F]]. apply E, (D f). apply satis_Pos, F.
  − intros D f E. decide (satis f s) as [F|F]. exact F. exfalso.
    apply D. ∃ f. split. exact F. apply satis_Pos, E.
Qed.
```

Exercise 11.2.1 Prove the following weakening results.

```
Lemma satis_weak f C C' : C ⊆ C' → satis' f C' → satis' f C.

Lemma sat_weak C C' : C ⊆ C' → sat C' → sat C.
```

Exercise 11.2.2 Prove the following fact about the unsign function.

```
Lemma uns_Pos A : map uns (map Pos A) = A.
```

Exercise 11.2.3 In the following we assume that an instance rule *sform_eq_dec* for the decidability of equality of signed formulas is registered.

a) Register such a rule.

b) Explain why such a rule yields the decidability of membership and equality for clauses.

Exercise 11.2.4 A formula is **valid** if it is satisfied by every assignment. Show the following with Coq.

a) Every formula provable in the empty context is valid.

b) A formula s is valid if and only if the clause $[s^-]$ is unsatisfiable.

11.3 Main Results

Completeness and decidability of classical natural deduction follow from a lemma saying that a clause is either satisfiable or refutable with natural deduction.[2] We will refer to this lemma as *main lemma*. We assume the main lemma in a section and prove decidability and completeness.

```
Section MainResults.
Variable main : ∀ C, {sat C} + {ndc (map uns C) Fal}.

Lemma ndc_dec A s :
  dec (ndc A s).

Proof.
  destruct (main (-s :: map Pos A)) as [E|E].
  - right. destruct E as [f [E F]]. intros G. apply ndc_sem in G.
    simpl in E. apply E. apply G. apply satis_Pos, F.
  - left. simpl in E. rewrite uns_Pos in E. apply ndcC, E.
Qed.

Lemma ndc_iff_sem A s :
  ndc A s ↔ sem A s.

Proof.
  split.
  - apply ndc_sem.
  - intros E. apply sem_iff_unsat in E.
    destruct (main (-s :: map Pos A)) as [F|F].
    + contradiction (E F).
    + simpl in F. rewrite uns_Pos in F. apply ndcC, F.
Qed.

End MainResults.
```

[2] Recall that classical refutability agrees with intuitionistic refutability by Glivenko's Theorem.

Exercise 11.3.1 Assume the main lemma and prove the following propositions.

a) ∀ s, ndc nil s ↔ ∀ f, satis f s.

b) ∀ s, ndc nil s ↔ ¬ sat [−s].

c) ∀ A, ndc A Fal ↔ ¬ sat (map Pos A).

d) ∀ A s, ndc A s ↔ ¬ sat (−s :: map Pos A).

e) ∀ C, dec (sat C).

11.4 Solved Clauses

We will prove the main lemma with a recursive procedure that simplifies a given clause until it arrives at a solved clause or a clashed clause. Solved clauses are always satisfiable and clashed clauses are always refutable.

A clause is **clashed** if it contains ⊥ or a conflicting pair s^+ and s^- of signed formulas.

A clause is **solved** if it contains only signed variables and no conflicting pair x^+ and x^-. A solved clause can be understood as a partial assignment that fixes the values of finitely many variables. A solved clause is satisfied by every assignment that respects the constraints imposed by the signed variables of the clause. Since the signed variables of a solved clause do not clash, every signed clause is satisfiable. We will work with an inductive definition of solved clauses.

```
Inductive sol : clause → Prop :=
| solNil :       sol  nil
| solPV C x :   ¬ −Var x ∈ C → sol C → sol (+Var x :: C)
| solNV C x :   ¬ +Var x ∈ C → sol C → sol (−Var x :: C).

Lemma sol_clash C x :
  sol C → +Var x ∈ C → −Var x ∈ C → False.

Proof.
  intros A E F.
  induction A as [|C y G _ IH|C y G _ IH] ; simpl in *.
  − contradiction E.
  − apply IH ; destruct E, F ; congruence.
  − apply IH ; destruct E, F ; congruence.
Qed.

Lemma sol_satis' f C :
  sol C →
  (∀ x, +Var x ∈ C → f x = true) →
  (∀ x, −Var x ∈ C → f x = false) →
  satis'  f  C.
```

```
Proof.
  intros A E F.
  induction A as [|C y G _ IH|C y G _ IH] ; simpl in *.
  - exact I.
  - auto.
  - split ; [| now auto].
    intros H. rewrite F in H. discriminate H. auto.
Qed.

Lemma sol_sat C :
  sol C → sat C.

Proof.
  intros A.
  ∃ (fun x ⇒ if decision (+Var x ∈ C) then true else false).
  apply (sol_satis' A) ; intros x E ; decide (+Var x ∈ C) as [G|G] ; auto.
  contradiction (sol_clash A G E).
Qed.
```

Exercise 11.4.1 Prove that every clashed clause is refutable.

Exercise 11.4.2 A *Hintikka set* is a set H of formulas satisfying the following conditions:

1. $\bot \notin H$.
2. If $x^- \in H$, then $x \notin H$.
3. If $s \to t \in H$, then either $s^- \in H$ or $t \in H$.
4. If $s \to t^- \in H$, then $s \in H$ and $t^- \notin H$.

Show that every clause representing a Hintikka set is satisfiable. Proceed as follows.

```
Definition Hintikka' (S : sform) (C : clause) : Prop :=
  match S with
    | - Var x ⇒ ¬ + Var x ∈ C
    | + Imp s t ⇒ -s ∈ C ∨ +t ∈ C
    | - Imp s t ⇒ +s ∈ C ∧ -t ∈ C
    | + Fal ⇒ False
    | _ ⇒ True
  end.

Definition Hintikka (C : clause) : Prop :=
  ∀ S, S ∈ C → Hintikka' S C.

Lemma Hintikka_satis C f s :
  Hintikka C →
  (∀ x, +Var x ∈ C → f x = true) →
  (∀ x, -Var x ∈ C → f x = false) →
  (+s ∈ C → satis f s) ∧ (-s ∈ C → ¬ satis f s).
```

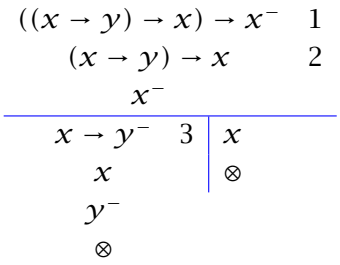

Figure 11.1: Complete tableau for the clause $[((x \to y) \to x) \to x^-]$

Lemma hin_sat C : Hintikka C → sat C.

Hint: *Hintikka_satis* can be shown by induction on s.

11.5 Tableau Procedure

There is a straightforward procedure that given a clause decides whether the clause is satisfiable. We call the procedure **tableau procedure**. The tableau procedure can be refined so that it yields a certifying function proving the main lemma.

The tableau procedure reduces the satisfiability of clauses with implications to the satisfiability of clauses without implications. For implication-free clauses satisfiability is obviously decidable. The removal of implications is justified by the equivalences

$$sat\ (s \to t^+ :: C) \ \leftrightarrow\ sat\ (s^- :: C) \ \vee\ sat\ (t^+ :: C)$$
$$sat\ (s \to t^- :: C) \ \leftrightarrow\ sat\ (s^+ :: t^- :: C)$$

The equivalences are straightforward consequences of the decomposition equivalences for implications (see Exercise 11.1.1).

If we run the tableau procedure by hand, we can do the necessary bookkeeping with tables known as **tableaux**. Figure 11.1 shows a complete tableau for the clause $[((x \to y) \to x) \to x^-]$. We start with the signed formulas of the clause and decompose the present implications one by one. For positive implications we branch since we need to consider two clauses. This yields a tree structure where each branch represents a clause. New formulas are added at the end of a branch. Decomposed implications are marked with a number. We stop the exploration of a branch if it contains a clash, which we mark with the symbol $\otimes$. A clash is either a pair s and s^- or a positive occurrence of $\bot$. We stop the expansion of the tableau if either there is a solved branch or all branches are clashed.

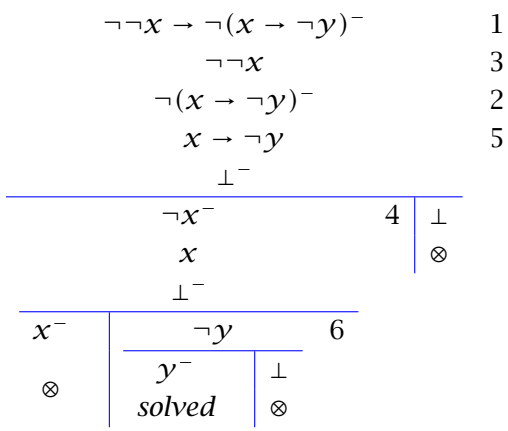

Figure 11.2: Complete tableau for the clause $[\neg\neg x \to \neg(x \to \neg y)^-]$

A branch is solved if it contains no clash and all implications are decomposed. If all branches are clashed, we know that the initial clause is unsatisfiable. If there is a solved branch, we know that the initial clause is satisfiable. In fact, every assignment satisfying all signed variables of a solved branch will satisfy all formulas on the branch.

Figure 11.2 shows a complete tableau for the clause $[\neg\neg x \to \neg(x \to \neg y)^-]$. The tableau has 4 branches, three of them clashed and one of them solved. Thus the initial clause is satisfiable. In fact, the initial clause is satisfied by every assignment satisfying the solved clause $[x, y^-]$.

Exercise 11.5.1 For each of the following formulas s give a complete tableau for the clause $[s^-]$. Then say whether the formula is valid. If the formula is not valid, give a solved clause such that every assignment satisfying the clause dissatisfies the formula.

a) $x \to y \to x$

b) $(x \to y \to z) \to (x \to y) \to x \to z$

c) $(x \to \neg y \to \bot) \to \neg\neg(x \to y)$

d) $\neg\neg x \to \neg y \to \neg(x \to y)$

e) $(x \to y) \to (y \to x) \to z$

Exercise 11.5.2 The tableau procedure can be improved by adding decomposition rules. Two such rules are

$$sat\ (\neg s^- :: C) \leftrightarrow sat\ (s^+ :: C)$$
$$sat\ (\neg s^+ :: C) \leftrightarrow sat\ (s^- :: C)$$

$$\frac{}{rec\ C\ nil} \qquad \frac{}{rec\ C\ (\bot :: D)} \qquad \frac{rec\ C\ D}{rec\ C\ (\bot^- :: D)}$$

$$\frac{x^- \in C}{rec\ C\ (x :: D)} \qquad \frac{x^- \notin C \qquad rec\ (x :: C)\ D}{rec\ C\ (x :: D)}$$

$$\frac{x \in C}{rec\ C\ (x^- :: D)} \qquad \frac{x \notin C \qquad rec\ (x^- :: C)\ D}{rec\ C\ (x^- :: D)}$$

$$\frac{rec\ C\ (s^- :: D) \qquad rec\ C\ (t :: D)}{rec\ C\ (s \to t :: D)} \qquad \frac{rec\ C\ (s :: t^- :: D)}{rec\ C\ (s \to t^- :: D)}$$

Figure 11.3: Recursion rules for the tableau procedure

Prove the correctness of the rules and redo some of the examples of Exercise 11.5.1 using the rules.

11.6 Tableau Recursion Rules

We refine the tableau procedure such that it works on two clauses rather than one. The concatenation of the two clauses represents the single clause of the naive procedure. We refer to the two clauses as **partial assignment** and **agenda**. The partial assignment is a solved clause and contains the signed variables seen so far. When we start the procedure, the partial assignment is empty and the agenda contains all the formulas. The procedure proceeds by processing the formulas on the agenda in the order they appear. If it finds a clash, it stops and announces unsatisfiability. If it ends up with an empty agenda, it announces satisfiability. In this case every assignment satisfying the partial assignment satisfies the initial clause.

The rules in Figure 11.3 describe the recursion structure of the procedure. Each rule says that the satisfiability of the partial assignment and the agenda in the conclusion can be decided by analyzing the results of the recursive calls specified by the premises. Note that to each pair of clauses exactly one rule applies (taking into account the side conditions of the variable rules). The first rule handles the case of an empty agenda. It is correct since the partial assignment will always be a solved clause.

The termination of the procedure follows from the fact that each recursion step reduces the size of the agenda, where the size of the agenda is the sum of

the sizes of the formulas on the agenda.

It is now routine to implement the tableau procedure in a programming language. In Coq, however, we face the problem that the procedure is not structurally recursive. The problem has an elegant solution. We will write two functions called **provider** and **decider**. The provider takes two clauses C and D and returns a **decision tree** for C and D as established by the recursion rules. The decider recurses on the decision tree and decides the satisfiability of $C +\!\!+ D$. Make sure you can say for each recursion rule how the decider decides the satisfiability of the conclusion given the decisions for the premises. Writing the decider in Coq is straightforward.

We formalize the recursion rules and the concomitant decision trees with an inductive type definition.

```
Inductive rec (C : clause) : clause → Type :=
| recNil :       rec C nil
| recPF D :      rec C (+Fal :: D)
| recNF D :      rec C D → rec C (−Fal ::D)
| recPV D x :    −Var x ∈ C → rec C (+Var x :: D)
| recPV' D x :   ¬ −Var x ∈ C → rec (+Var x :: C) D → rec C (+Var x :: D)
| recNV D x :    +Var x ∈ C → rec C (−Var x :: D)
| recNV' D x :   ¬ +Var x ∈ C → rec (−Var x :: C) D → rec C (−Var x :: D)
| recPI D s t :  rec C (−s :: D) → rec C (+t :: D) → rec C (+Imp s t :: D)
| recNI D s t :  rec C (+s :: −t :: D) → rec C (−Imp s t :: D).
```

From the structure of the rules it is clear that every type *rec C D* has an element which is a decision tree for the clauses C and D. The provider is a function that for two clauses C and D yields a decision tree in *rec C D*. We realize the provider by recursion on the size of the agenda. The necessary recursion operator will be obtained as a straightforward generalization of size induction.

Writing the provider in Coq amounts to proving the termination of the tableau procedure. The provider cound be written as an ordinary proof if we made the type constructor *rec* an inductive predicate. Given that the proof method size induction smoothly generalizes to the computation method size recursion, there is no problem in accommodating *rec* as an inductive type constructor. Accommodating *rec* as an inductive type constructor has the advantage that the decider is not hindered by the elim restriction.

We define the size of signed formulas and clauses as one would expect.

```
Fixpoint sizeF (s : form) : nat :=
  match s with
    | Imp s1 s2 ⇒ 1 + sizeF s1 + sizeF s2
    | _ ⇒ 1
  end.
```

```
Fixpoint size (C : clause) : nat :=
  match C with
    | nil ⇒ 0
    | +s::C' ⇒ sizeF s + size C'
    | −s::C' ⇒ sizeF s + size C'
  end.
```

We establish size recursion with a lemma. The proof is the same as for size induction (see Section 4.3).

```
Lemma size_recursion X (f : X → nat) (t : X → Type) :
  (∀ x, (∀ y, f y < f x → t y) → t x) → ∀ x, t x.
```

We can now write the provider.

```
Lemma rec_total C D :  rec C D.
```

```
Proof.
  revert D C.
  refine (size_recursion (f:= size) _).
  intros  [|[[ x|s  t |]|[ x|s  t |]]  D] IH C.
  − constructor.
  − decide (−Var x ∈ C) as [E|E].
    + apply (recPV _ E).
    + apply (recPV' E). apply IH. simpl ; omega.
  − constructor ; apply IH ; simpl ; omega.
  − constructor ; apply IH ; simpl ; omega.
  − decide (+Var x ∈ C) as [E|E].
    + apply (recNV _ E).
    + apply (recNV' E). apply IH. simpl ; omega.
  − constructor ; apply IH ; simpl ; omega.
  − constructor ; apply IH ; simpl ; omega.
Qed.
```

Note that size recursion is applied with the tactic *refine*. The tactic *refine* is like the tactic *exact* but creates subgoals for underivable wildcard arguments (arguments specified with "_"). The tactic *refine* is often the right solution if the tactic *apply* fails to do the right thing.

11.7 Generic Certifying Tableau Procedure

We will formalize the decider for the tableau procedure as a certifying function. This way we verify the correctness of the tableau procedure. In fact, writing the provider amounts to proving termination, and writing the decider amounts to proving partial correctness of the tableau procedure.

Given our design, the decider should have the type

$$\forall C\ D.\ rec\ C\ D \to sol\ C \to \{sat\ (C +\!\!+ D)\} + \{\neg sat\ (C +\!\!+ D)\}$$

where the argument of type *sol C* formalizes the invariant that the partial assignment C is a solved clause. We will write the decider in a generalized form where the right hand side of the sum is written with an abstract **refutation predicate** *ref*.

$$\forall C\ D.\ rec\ C\ D \to sol\ C \to \{sat\ (C +\!\!+ D)\} + \{ref\ (C +\!\!+ D)\}$$

The decider for satisfiability can then be obtained by choosing the refutation predicate $\lambda C.\ \neg sat\ C$.

To write the generalized decider, we need certain assumptions about the refutation predicate. The necessary assumptions become obvious once we construct the decider by recursion on the decision tree. We state the assumptions for the refutation predicate beforehand.

```
Section GCTP.
Variable ref : clause → Prop.
Variable ref_Fal : ∀ C,  +Fal ∈ C → ref C.
Variable ref_weak : ∀ C C',  C ⊆ C' → ref C → ref C'.
Variable ref_clash : ∀ x C,  +Var x ∈ C → −Var x ∈ C → ref C.
Variable ref_pos_imp : ∀ s t C,  ref (−s::C) → ref (+t::C) → ref (+Imp s t::C).
Variable ref_neg_imp : ∀ s t C,  ref (+s::−t::C) → ref (−Imp s t::C).
```

It is now straightforward to write the decider by recursion on the decision tree. We generate the code for the decider with a script.

```
Lemma rec_sat_ref C D :
  rec C D → sol C → {sat (D++C)} + {ref (D++C)}.
Proof.
  intros A B.
  induction A as [C|C D|C D _ IH|C D x E|C D x E _ IH|C D x E|C D x E _ IH|
                  C D s t _ IHs _ IHt|C D s t _ IH] ; simpl.
  one subgoal for every recursion rule
Qed.
```

Constructing the certifying decision procedure *rec_sat_ref* with a script amounts to computer-assisted programming. Of great help is the induction tactic, which takes care of the bureaucratic details and gives us a subgoal for every recursion rule. The subgoals provides us with precise specifications of the missing code fragments, which must realize the local deciders for the recursion rules. Here are the scripts for the fragments.

```
- left. apply sol_sat, B.
- right. apply ref_Fal. auto.
- destruct (IH B) as [F|F].
  + left. destruct F as [f F]. ∃ f. simpl ; auto.
  + right. revert F. apply ref_weak. auto.
- right. apply ref_clash with (x:=x) ; auto.
- destruct IH as [F|F].
  + constructor ; assumption.
  + left. revert F. apply sat_weak. auto.
  + right. revert F. apply ref_weak. auto.
- right. apply ref_clash with (x:=x) ; auto.
- destruct IH as [F|F].
  + constructor ; assumption.
  + left. revert F. apply sat_weak. auto.
  + right. revert F. apply ref_weak. auto.
- destruct (IHs B) as [F|F].
  + left. destruct F as [f [F G]]. ∃ f. simpl in *. tauto.
  + destruct (IHt B) as [G|G].
    * left. destruct G as [f G]. ∃ f. simpl in *. tauto.
    * right. apply (ref_pos_imp F G).
- destruct (IH B) as [F|F].
  + left. destruct F as [f F]. ∃ f. simpl in *. tauto.
  + right. apply ref_neg_imp, F.
```

We now combine the provider with the generalized decider and obtain a generic certifying tableau procedure.

```
Lemma sat_plus_ref C :
  {sat C} + {ref C}.
Proof.
  destruct (rec_sat_ref (C:=nil) (D:=C)) as [E|E].
  - apply rec_total.
  - constructor.
  - left. revert E. apply sat_weak. auto.
  - right. revert E. apply ref_weak. auto.
Qed.
End GCTP.
```

We now instantiate the generic certifying tableau procedure to a certifying decision procedure for satisfiability. To do this, we have to prove that unsatisfiability satisfies the properties of a refutation predicate.

```
Lemma sat_dec C :
  dec (sat C).
Proof.
  revert C. apply sat_plus_ref with (ref:= fun C ⇒ ¬ sat C).
```

```
- intros C E [f F].  apply (satis_in F E).
- intros C C' A E F. apply E. revert A F. apply sat_weak.
- intros s C A B [f E].  apply (satis_in E B),  (satis_in E A).
- intros s t C A B [f  [E F]].  apply satis_pos_imp in E as [E|E].
  + apply A. ∃ f. simpl. auto.
  + apply B. ∃ f. simpl. auto.
- intros s t C A [f  [E F]].  apply A. ∃ f. simpl in *.
  apply satis_neg_imp in E as [E G]. auto.
Qed.
```

Exercise 11.7.1 Extend the section *GCTP* with the following lemmas for sound refutation predicates.

```
Variable ref_sound : ∀ C, ref C → ¬ sat C.
Lemma ref_iff_unsat C :  ref C ↔ ¬ sat C.
Lemma ref_dec C :  dec (ref C).
```

11.8 Proof of the Main Lemma

Given the generic certifying tableau procedure, the proof of the main lemma is straightforward: We simply show that $(\lambda C.\ ndc\ (map\ uns\ C)\ \bot)$ is a refutation predicate.

```
Lemma main C :
  {sat C} + {ndc (map uns C) Fal}.
Proof.
  revert C. apply sat_plus_ref ; simpl.
- intros C A. apply ndcA. exact (in_map uns _ _ A).
- intros C C' A. apply ndc_weak. apply incl_map, A.
- intros x C A B. apply ndcIE with (s:=Var x) ; apply ndcA.
  + exact (in_map uns _ _ B).
  + exact (in_map uns _ _ A).
- intros s t C A B. apply ndcIE with (s:= Not s).
  + apply ndcW, ndcII, A.
  + apply ndcII. apply ndcIE with (s:=t).
    * apply ndcW, ndcW, ndcII, B.
    * apply ndcIE with (s:=s) ; apply ndcA ; auto.
- intros s t C A. apply ndcIE with (s:= Imp s t).
  + apply ndcA ; auto.
  + apply ndcII, ndcE. apply ndcIE with (s:= Not t).
    * { apply ndcIE with (s:= s).
        - apply ndcW, ndcW, ndcII, ndcII. revert A. apply ndc_weak. firstorder.
        - apply ndcA ; auto. }
    * { apply ndcII. apply ndcIE with (s:= Imp s t).
```

```
- apply ndcA ; auto.
- apply ndcII, ndcA ; auto. }
```
Qed.

Exercise 11.8.1 Consider the following *refutation rules* for clauses.

$$\frac{}{C, \bot} \qquad \frac{}{C, s, s^-} \qquad \frac{C, s^- \qquad C, t}{C, s \to t} \qquad \frac{C, s, t^-}{C, s \to t^-} \qquad \frac{C}{C'}\ C \subseteq C'$$

You will show that the rules yield a sound and complete derivation system for unsatisfiable clauses.

a) Verify for each rule that it has a satisfiable premise if the conclusion is satisfiable. Note that this implies that the rules derive unsatisfiable conclusions from unsatisfiable premises.

b) Formalize the rules as an inductive predicate $tab : clause \to Prop$.

c) Verify that *tab* is a refutation predicate.

d) Prove $\forall C.\ tab\ C \to \neg sat\ C$.

e) Prove $\forall C.\ \{sat\ C\} + \{tab\ C\}$.

f) Prove $\forall C.\ tab\ C \leftrightarrow \neg sat\ C$.

g) Prove $\forall C.\ dec\ (tab\ C)$.

Exercise 11.8.2 Consider the following *demo rules* for clauses.

$$\frac{}{C}\ C \text{ solved} \qquad \frac{C}{C, \bot^-} \qquad \frac{C, s^-}{C, s \to t} \qquad \frac{C, t}{C, s \to t} \qquad \frac{C, s, t^-}{C, s \to t^-} \qquad \frac{C}{C'}\ C' \subseteq C$$

You will show that the rules yield a sound and complete derivation system for satisfiable clauses.

a) Verify for each rule that the conclusion is satisfiable if all premises are satisfiable.

b) Formalize the rules as an inductive predicate $dem : clause \to Prop$.

c) Prove $\forall C.\ dem\ C \to sat\ C$.

d) Prove $\forall C.\ \{dem\ C\} + \{\neg sat\ C\}$.

e) Prove $\forall C.\ dem\ C \leftrightarrow sat\ C$.

f) Prove $\forall C.\ dec\ (dem\ C)$.

The final three results are best obtained with a tableau procedure of the type $\forall C.\ \{dem\ C\} + \{\neg sat\ C\}$.

Exercise 11.8.3 Prove that a clause is satisfiable if and only if it can be extended to a Hintikka clause. The direction from Hintikka to satisfiability is already covered by Exercise 11.4.2. The other direction can be shown by proving

$$\forall C.\ \mathit{dem}\ C \to \exists H.\ \mathit{Hintikka}\ H \wedge C \subseteq H$$

for the demo predicate from Exercise 11.8.2.

Coq Summary

New Tactics

- *simpl in* $*$ simplifies all assumptions and the claim of the goal.
- *refine* is like *exact* but creates subgoals for underivable wildcard arguments (arguments specified with "_"). See the proof of *rec_total* in Section 11.6.

Tactical *now*

In the Coq file for this chapter you will see a few uses of the tactical *now*. If *now* is written in front of a tactic command, the tactic command will fail if it does not solve the goal. For instance, *now auto* succeeds if and only if *auto* solves the goal. If a split spawns a first subgoal that can be solved by t, we may write

```
split. now t. script for second subgoal
```

If the second subgoal of a split can be solved by t, we may write

```
split ; [|now t]. script for first subgoal
```

If t is *exact* or *reflexivity* or another tactic that always fails if it doesn't solve the goal, we omit *now*.

The use of *now* and bullets (i.e., $-$, $+$, $*$) is a matter of style. It improves readability of scripts. It pays off when you adapt the existing script of a modified lemma.

New Notations

- "$x = y$:> X" stands for the term *@eq X x y*.

12 Intuitionistic Semantics

In this chapter we prove results about intuitionistic propositional logic. We begin by giving an appropriate semantics for the intuitionistic case. The semantics will be given by models which consist of a collection of states where each state represents a boolean assignment. We will prove a soundness result for the intuitionistic natural deduction system relative to this class of models. The notion of a model allows us to define when clauses are satisfiable or unsatisfiable.

We next consider a special case of these models called demos. Demos have properties which make it easy to see that certain formulas are either satisfied or dissatisfied at a state of the demo. We will use a demo to prove an independence result: $\neg\neg x \to x$ is not intuitionistically provable. There is a tableau procedure which can be used to construct a demo for a satisfiable clause or to determine that a clause is unsatisfiable. The algorithmic interpretation of a demo motivates a tableau refutation system which characterizes the unsatisfiable clauses. We prove one can translate from tableau refutations to natural deduction derivations.

We finally turn to the question of decidability of $A \vdash s$ in $\mathcal{N}$ and completeness of $\mathcal{N}$ with respect to clausal models. These results will easily follow from an assumption which we call the *main lemma*. This is analogous to the main lemma for the classical case considered in Section 11.3. The main lemma states that every clause is either satisfiable or tableau refutable. The proof of the main lemma will be the subject of the next chapter.

12.1 Clausal Models

In the classical case it was enough to consider boolean assignments. That is, when a formula is not classically provable, there is an assignment making the formula false. Assignments do not provide enough counterexamples to handle intuitionistically unprovable formulas. For example, $\neg\neg x \to x$ is intuitionistically unprovable, but is true when evaluated under an assignment. What we need is some way to interpret formulas so that a formula may be neither true nor false. We begin with an informal description of how to obtain such an interpretation.

One option is to use sets of assignments and to reconsider how we interpret implication. As a simple example, consider two assignments f and g where

$f\,y = \mathit{false}$ for all variables y, $g\,x = \mathit{true}$, and $g\,y = \mathit{false}$ for all $y \neq x$. An equivalent way to represent an assignment is as the set of variables assigned to *true*. The set of variables assigned to *true* by f is the empty set $\emptyset$. The set of variables assigned to *true* by g is the singleton set $\{x\}$. Since

$$\emptyset \subseteq \{x\}$$

we consider $\{x\}$ as an extension of $\emptyset$. We can consider $\emptyset$ and $\{x\}$ as two states of one model. The state $\emptyset$ is the "earlier" state and the state $\{x\}$ is the "later" state. In the earlier state, x is not satisfied. In the later state, x is satisfied. Suppose we interpret implication so that $\neg s$ is satisfied if there is no later state where s is satisfied. Then $\neg x$ is not satisfied in either state. Consequently $\neg\neg x$ is satisfied in the state $\emptyset$, even though x is not satisfied in $\emptyset$.

Instead of defining models using sets of assignments, we will use sets of clauses. In terms of the satisfaction of formulas, it will only matter which positive variables occur in the clauses. Using clauses will pay off later when we consider a special case of models.

Recall that a clause is a list of signed formulas. We will define **clausal models** as lists of clauses. There will be an important relation between these clauses. In order to define and prove properties of this relation, we first need to consider positive signed formulas and an operation which removes negative formulas from a clause.

In Coq, we have the following (obviously decidable) predicate for testing if a signed formula is positive:

```
Definition positive (S : sform) : Prop :=
  match S with +s ⇒ True | −s ⇒ False end.
```

For a clause A, we define A^+ to be $\{s^+ | s^+ \in A\}$. That is, A^+ is the set of positive formulas in A.

```
Fixpoint pos (C : clause) : clause :=
  match C with
    | nil ⇒ nil
    | + s :: C' ⇒ + s :: pos C'
    | − s :: C' ⇒ pos C'
  end.
```

There are a number of easily proven properties of *pos*. We list three and leave the reader to do the proofs.

```
Lemma pos_iff S C :
  S ∈ pos C ↔ S ∈ C ∧ positive S.

Lemma pos_incl C :
  pos C ⊆ C.
```

```
Lemma pos_incl_pos A B :
  pos A ⊆ B → pos A ⊆ pos B.
```

In this chapter we will also use the term **state** for the clauses of a model. A **model** is a list of states.

```
Definition state := clause.
Definition model := list state.
```

For a model M, we define $A \preceq_M B$ to hold when $A \in M$, $B \in M$ and $A^+ \subseteq B$. We will write $A \preceq B$ when the model M is clear from the context. When $A \preceq B$ holds, we will say "A **before** B" and refer to A as an "earlier" state and B as a "later" state.

We will now define $\preceq$ in Coq. Let M be a model.

```
Section Model.
  Variable M : model.
```

We define $\preceq$ as follows.

```
Definition before (A B : state) : Prop :=
  A ∈ M ∧ B ∈ M ∧ pos A ⊆ B.
```

It is easy to see that $\preceq$ is a reflexive and transitive relation on M. In particular, $A \preceq A$ since $A^+ \subseteq A$. Also, if $A \preceq B$ and $B \preceq C$, then $A^+ \subseteq C$ since $A^+ \subseteq B^+ \subseteq C$. Note that if $A \preceq B$ and $s^+ \in A$, then $s^+ \in B$.

```
Lemma before_refl A :
  A ∈ M → before A A.
Proof. unfold before. intuition. apply pos_incl. Qed.
Lemma before_tran A B C :
  before A B → before B C → before A C.
Proof.
  intros [F [_ G]] [_ [H K]]. unfold before. intuition.
  setoid_rewrite ← K. apply pos_incl_pos, G.
Qed.
Lemma before_in s A B :
  before A B → +s ∈ A → +s ∈ B.
Proof. intros [_ [_ E]] F. apply E. apply pos_iff. simpl. auto. Qed.
```

The Coq proof scripts for reflexivity and transtivity use a new automation tactic *intuition*. *intuition* simplifies goals by means of intros and propositional reasoning. In general, *intuition* may leave several subgoals for the user. In this chapter we will only use *intuition* when proving a conjunctive claim of the form $s_1 \wedge \cdots \wedge s_n$ where Coq can prove all the conjuncts except one. After such an application of *intuition*, we will be left with one subgoal to prove.

We now define when a state A in M **satisfies** a formula s. The definition is by recursion on s:

- A satisfies x if $x^+ \in A$.
- A satisfies $s \to t$ if B satisfies t whenever B is a state such that $A \preceq B$ and B satisfies s.
- A does not satisfy $\bot$.

In Coq, the definition is as follows.

```
Fixpoint satis (A : state) (s : form) : Prop :=
  match s with
    | Var x ⇒ +Var x ∈ A
    | Imp s t ⇒ forall B, before A B → satis B s → satis B t
    | Fal ⇒ False
  end.
```

The following result follows easily from reflexivity of $\preceq$: For all $A \in M$, if A satisfies $\neg s$, then A does not satisfy s.

```
Lemma satis_not A s :
  A ∈ M → satis A (Not s) → ¬ satis A s.
```

Transitivity of $\preceq$ can be used to easily prove all later states satisfy all the formulas the earlier state satisfied. We call this property **monotonicity**.

```
Lemma satis_mono A B s :
  before A B → satis A s → satis B s.
```

We now extend the notion of satisfaction from formulas to clauses. We say a state A **satisfies** a clause C if A satisfies s for every $s^+ \in C$ and A dissatisfies s for every $s^- \in C$. In Coq we realize this definition using recursion over the clause.

```
Fixpoint satis' (A : state) (C : clause) : Prop :=
  match C with
    | nil ⇒ True
    | +s :: C' ⇒ satis A s ∧ satis' A C'
    | −s :: C' ⇒ ¬ satis A s ∧ satis' A C'
  end.
```

An induction on C justifies the equivalence between the recursive definition and a characterization closer to the mathematical definition above.

```
Lemma satis_iff A C :
  satis' A C ↔ ∀ S, S ∈ C →
                    match S with
                      | +s ⇒ satis A s
                      | −s ⇒ ¬ satis A s
                    end.
```

Clearly for states A and clauses C and D if $C \subseteq D$ and A satisfies D, then A satisfies C.

```
Lemma satis_weak C D A :
  C ⊆ D → satis' A D → satis' A C.
```

We noted above that if a state A satisfies a formula s, then every later state B also satisfies s. The converse does not hold in general. It is possible for a state A to dissatisfy a formula s but for a later state B to satisfy s. Consequently, a state A may satisfy the clause $[s^-]$ while a later state B does not satisfy $[s^-]$.

It is not difficult to prove the satisfaction predicate is decidable.

```
Global Instance satis_dec A s :
  dec (satis A s).
```

The proof is by induction on s using the following lemma which follows from the fact that list quantification preserves decidability.

```
Global Instance before_forall_dec A p :
  (∀ B, dec (p B)) → dec (∀ B, before A B → p B).
```

12.2 Soundness

We now define **semantic entailment** using clausal models. We say A **semantically entails** s if for every clausal model M and every state B in M, if B satisfies all the formulas in A, then B satisfies s.

```
Definition sem (A : context) (s : form) : Prop :=
  ∀ M B, B ∈ M → (∀ t, t ∈ A → satis M B t) → satis M B s.
```

We can now prove soundness. That is, we prove that if $A \vdash s$, then A semantically entails s.

```
Lemma nd_sem A s :
  nd A s → sem A s.
```

The proof is by induction. We fix a model M. We must check the four rules defining $\mathcal{N}$ preserve the following property (of A and s):

$$\forall B \in M. \text{if } B \text{ satisfies } A, \text{ then } B \text{ satisfies } s.$$

- AI: Assume $s \in A$ and B satisfies A. Clearly B satisfies s.
- II: The inductive hypothesis in this case is

  $$\forall M, B \in M. \text{if } B \text{ satisfies } A, s, \text{ then } B \text{ satisfies } t.$$

  Assume B satisfies A. We will prove B satisfies $s \to t$. Let B' be such that $B \preceq B'$ and B' satisfies s. In particular, $B' \in M$. By monotonicity B' satisfies the context A. By the inductive hypothesis B' satisfies t, as desired.

- IE: In this case there are two inductive hypotheses:

$$\forall M, B \in M. \text{if } B \text{ satisfies } A, \text{ then } B \text{ satisfies } s \to t.$$

$$\forall M, B \in M. \text{if } B \text{ satisfies } A, \text{ then } B \text{ satisfies } s.$$

  Assume B satisfies A. We must prove B satisfies t. By the inductive hypotheses B satisfies $s \to t$ and s. Since $B \preceq B$ we know B satisfies t.
- E: In this case the inductive hypothesis implies there is no state satisfying the context A. We vacuously conclude that every B satisfying A also satisfies s.

12.3 Satisfiability

A clause is **satisfiable** if there is a model M and a state $A \in M$ such that A satisfies C.

```
Definition sat (C : clause) : Prop :=
  ∃ M A, A ∈ M ∧ satis' M A C.
```

A clause is **unsatisfiable** if it is not satisfiable.

Clearly if a clause D is satisfiable and $C \subseteq D$, then C is satisfiable. (The same model can act as a witness.)

```
Lemma sat_weak C D :
  C ⊆ D → sat D → sat C.
```

If A semantically entails s, then clearly there is no model with a state satisfying A and dissatisfying s and so the clause A, s^- is unsatisfiable.

```
Lemma sem_unsat A s :
  sem A s → ¬ sat (-s :: map Pos A).
```

The equivalence can also be proven.

```
Lemma sem_iff_unsat A s :
 sem A s ↔ ¬ sat (-s :: map Pos A).
```

The proof uses decidability of *satis M B s*.

12.4 Demos

Given a clause C we would like an algorithm for determining whether or not C is satisfiable. In addition, if C is satisfiable, we would like to obtain a model satisfying C. A way to approach this is to analyze the subformulas of the signed formulas of C. Suppose M is a model, $A \in M$ and A satisfies C. Assume a positive implication $s \to t$ is in C. By reflexivity of $\preceq$, if A satisfies s, then A must

also satisfy t. Hence either A dissatisfies s or satisfies t. Consequently, A also satisfies the clause C, s^- or the clause C, t.

Next assume a negative implication $s \to t^-$ is in C. Since A dissatisfies $s \to t$, there must be some B such that $A \preceq B$, B satisfies s and B dissatisfies t. Also, by monotonicity, B must satisfy all the positive formulas in C. Hence B satisfies C^+, s, t^-.

We can try to construct a model satisfying C by extending C to include either s^- or t whenever $s \to t$ is in C. Furthermore, if a negative implication $s \to t^-$ is in C, then we can simultaneously attempt to include a state in the model satisfying C^+, s, t^-. Of course, if we reach a point in which $\bot$ is in C or there are conflicting signed formulas x and x^- in C, then we know C is unsatisfiable.

The procedure sketched above will allow us to construct special models called **demos** when the original clause is satisfiable. A **demo** is a model M such that for every state $A \in M$ we have the following conditions:

- $\bot \notin A$.
- If $x^- \in A$, then $x \notin A$.
- If $s \to t \in A$, then $s^- \in A$ or $t \in A$.
- If $s \to t^- \in A$, then there is some $B \in M$ such that $A \preceq B$, $s \in B$ and $t^- \in B$.

In Coq we can realize this definition using an auxiliary definition with a match corresponding to the conditions above.

```
Definition demo' (M : model) (A : clause) (S : sform) : Prop :=
  match S with
    | − Var x ⇒ ¬ +Var x ∈ A
    | + Imp s t ⇒ −s ∈ A ∨ +t ∈ A
    | − Imp s t ⇒ exists B, before M A B ∧ +s ∈ B ∧ −t ∈ B
    | + Fal ⇒ False
    | _ ⇒ True
  end.

Definition demo (M : model) : Prop :=
  ∀ A, A ∈ M → ∀ S, S ∈ A → demo' M A S.
```

We now turn to the **Demo Theorem** (*demo_satis*). If M is a demo and A is a state in M, then A satisfies all the positive formulas in A and dissatisfies all the negative formulas in A. We use an axiliary function *sign* to conveniently state and prove the result.

```
Definition sign (b : bool) :=
  if b then Pos else Neg.

Lemma demo_satis M A b s :
  demo M → A ∈ M → sign b s ∈ A →
  if b then satis M A s else ¬ satis M A s.
```

The demo theorem is proven by induction on s.

- Suppose $x \in A$. Clearly A satisfies x^+.
- Suppose $x^- \in A$. Since M is a demo, $x \notin A$. Hence A dissatisfies x.
- Suppose $s \to t \in A$. We prove A satisfies $s \to t$. Let B be such that $A \preceq B$ and B satisfies s. Since $A^+ \subseteq B$, we know $s \to t$ is in B. Since M is a demo, either $s^- \in B$ or $t \in B$. By the inductive hypothesis for s, if $s^- \in B$, then B dissatisfies s, contradicting our assumption about B. Hence $t \in B$. By the inductive hypothesis for t, B satisfies t, as desired.
- Suppose $s \to t^- \in A$. We prove A dissatisfies $s \to t$. Since M is a demo, there is some $B \in M$ such that $A \preceq B$, $s \in B$ and $t^- \in B$. By the inductive hypotheses, B satisfies s and dissatisfies t. This B witnesses that A dissatisfies $s \to t$.
- We cannot have $\bot \in A$ since M is a demo.
- Suppose $\bot^- \in A$. Clearly A dissatisfies $\bot$, as desired.

Here is a simple corollary of the Demo Theorem.

```
Lemma demo_satis' M A :
  demo M → A ∈ M → satis' M A A.
```

Example 12.4.1 Let x be a variable. In this example we construct a demo with a state satisfying the clause $\neg\neg x \to x^-$. Clearly such a demo must include a state satisfying $\neg\neg x^+$ and x^-. Since such a state cannot satisfy $\bot$, it must also satisfy $\neg x^-$. Combining these signed formulas, let A be the state

$$\neg\neg x \to x^-, \neg\neg x^+, \neg x^-, x^-.$$

Since $\neg x^-$ is in A, there must be another state with all the positive formulas in A, the positive formula x and the negative formula $\bot^-$. That is, we need a state satisfying

$$\neg\neg x^+, x^+, \bot^-.$$

To satisfy the demo condition for $\neg\neg x^+$, we also include $\neg x^-$. Let B be the state

$$\neg\neg x^+, x^+, \bot^-, \neg x^-.$$

Let M be the model with the two states A and B. It is easy to check that M is a demo. By the Demo Theorem, A dissatisfies $\neg\neg x \to x$. As as a consequence of soundness, $nil \not\Vdash \neg\neg x \to x$ in $\mathcal{N}$. □

Exercise 12.4.2 Use the demo theorem to prove the following.

```
Goal ∀ M A s, demo M → A ∈ M → +s ∈ A → ¬ −s ∈ A.
```

Exercise 12.4.3 For each of the following clauses, construct a demo with a state satisfying the clause. Assume x, y and z are distinct variables.

a) $x \to y,\ \neg x \to y, y^-$

b) $\neg x \to y,\ \neg\neg x \to y,\ y^-$

c) $(x \to y) \to z,\ (y \to x) \to z,\ z^-$

d) $\neg x \to y,\ x \to z,\ y \to z,\ z^-$

12.5 Intuitionistic Tableau System

All attempts to construct a demo for an unsatisfiable clause C must fail. If we analyze the choices one makes while trying to construct a demo, we arrive at a derivation system for unsatisfiable clauses. This is often called an intuitionistic tableau system. We first give the tableau system as rules.

$$\frac{}{C, \bot} \qquad \frac{}{C, s, s^-} \qquad \frac{C, s^- \qquad C, t}{C, s \to t} \qquad \frac{C^+, s, t^-}{C, s \to t^-} \qquad \frac{C}{C'}\ C \subseteq C'$$

We say a clause C is **tableau refutable** if it is derivable from these rules. For each of the tableau rules, if the conclusion is satisfiable, then one of the premises must be satisfiable. In Coq, we define a corresponding predicate as follows.

```
Inductive tab : clause → Prop :=
| tabF C :       tab (+Fal :: C)
| tabC x C :     tab (−Var x :: +Var x :: C)
| tabIP s t C :  tab (−s :: C) → tab (+ t :: C) → tab (+Imp s t :: C)
| tabIN s t C :  tab (+s :: −t :: pos C) → tab (−Imp s t :: C)
| tabW C C' :    C' ⊆ C → tab C' → tab C.
```

Note that in the rule to demonstrate refutability of $C, s \to t^-$ (i.e., a clause with a negative implication) the premise is C^+, s, t^-, not C, s, t^-.

We now prove we can translate from tableau refutations to natural deduction derivations. Since the tableau system works on clauses C while natural deduction works on contexts A and formulas s, it is not immediately clear what should be provable in the natural deduction system if C is tableau refutable. The obvious choice for a context A is $\{t | t^+ \in C\}$. This leaves the question of which formula s should follow from this context. A first approximation is to take some s such that $s^- \in C$. That is, some negative formula in C should follow from the positive formulas in C. However, it is possible that there is no negative formula in C. For this reason, we prove a disjunction: If C is tableau refutable, then $\{t | t^+ \in C\} \vdash \bot$ or there is some s such that $s^- \in C$ and $\{t | t^+ \in C\} \vdash s$. To represent $\{t | t^+ \in C\}$ we use the *uns* function from the previous chapter.

```
Definition uns (S : sform) : form :=
  match S with +s ⇒ s | −s ⇒ Not s end.
```

We can now represent the disjunction as the following definition.

```
Definition ndr (C : clause) : Prop :=
  nd (map uns (pos C)) Fal ∨ ∃ s, −s ∈ C ∧ nd (map uns (pos C)) s.
```

The translation result can now be stated in a compact manner.

```
Lemma tab_ndr C :
  tab C → ndr C.
```

The proof is by induction on the tableau refutation. We check only the two implication rules. The others are easy. To ease notation let $\langle C \rangle$ denote the context $\{t | t^+ \in C\}$ for a clause C.

- Assume two inductive hypothesis.

  (1) Either $\langle C, s^- \rangle \vdash \bot$ or $\langle C, s^- \rangle \vdash u$ for some u such that $u^- \in C, s^-$.

  (2) Either $\langle C, t \rangle \vdash \bot$ or $\langle C, t \rangle \vdash u$ for some u such that $u^- \in C, t$.

  If there is some $u^- \in C$ such that $\langle C \rangle \vdash u$, then we are done. Otherwise, the first inductive hypothesis implies $\langle C \rangle \vdash s$. Hence $\langle C, s \to t \rangle \vdash t$. If $\langle C, t \rangle \vdash \bot$, then we conclude $\langle C, s \to t \rangle \vdash \bot$. Otherwise, suppose u is such that $u^- \in C, t$ and $\langle C, t \rangle \vdash u$. Clearly $u^- \in C, s \to t$ and $\langle C, s \to t \rangle \vdash u$.
- Since t^- is the only negative formula in C^+, s, t^-, the inductive hypothesis implies $\langle C^+, s \rangle \vdash t$. Hence $\langle C \rangle \vdash s \to t$.

An easy consequence is that if A is a context and the clause A, s^- is tableau refutable, then $A \vdash s$. The previous result gives that either $A \vdash \bot$ (so that $A \vdash s$) or there is some u such that $u^- \in A, s^-$ such taht $A \vdash \bot$. Since s is the only negative formula in A, s^-, u must be s.

```
Lemma tab_nd A s :
  tab (−s :: map Pos A) → nd A s.
```

Exercise 12.5.1 Prove the following.

```
Lemma satis_sem M A C s :
  A ∈ M → satis' M A C → sem (map uns (pos C)) s → satis M A s.
```

Use the result *satis_sem*, *tab_nd* and *nd_sem* to prove soundness of tableau refutability.

```
Lemma tab_sound C :
  tab C → sat C → False.
```

12.6 Main Results

We assume the following property we call the **main lemma**. For every clause C we can decide whether C is satisfiable or C is tableau refutable.

```
Variable main : ∀ C, {sat C} + {tab C}.
```

From the main lemma it is easy to prove that for a context A and formula s, either $A \vdash s$ or A does not semantically entail s. One simply applies the main lemma to the clause A, s^-. If A, s^- is satisfiable, then A does not semantically entail s. If A, s^- is tableau refutable, then $A \vdash s$.

```
Lemma nd_plus_not_sem A s :
  {nd A s} + {¬ sem A s}.
```

We can now prove decidability of *nd*. By the lemma above, either $A \vdash s$ (so we are done) or A does not semantically entail s. Assume A does not semantically entail s. We will prove $A \nvdash s$. Assume $A \vdash s$. By soundness (*nd_sem*), A must semantically entail s, contradicting our assumption.

```
Lemma nd_dec A s :
  dec (nd A s).
```

Similarly, we can prove the equivalence of *nd* and semantic entailment.

```
Lemma nd_iff_sem A s :
  nd A s ↔ sem A s.
```

Half of the equivalence is soundness (*nd_sem*). The other half will be completeness. We assume A semantically entails s and must prove $A \vdash s$. Appying the *nd_plus_not_sem*, either $A \vdash s$ (so we are done) or A does not semantically entail s (contradicting our assumption).

12.7 Remarks

The first complete semantics for intuitionistic propositional logic was given by Saul Kripke. These are known as Kripke models. Kripke originally used such models as a semantics for various modal logics in 1959. Since intuitionistic logic can be embedded into a certain modal logic, this already provides a semantics for intuitionistic logic. In 1965 Kripke considered the intiuitionistic case in depth. The clausal models we have given here are essentially special cases of Kripke models.

The tableau system considered in this chapter is essentially the same as a tableau system given by Fitting in 1969.

Coq Summary

New Tactics

- *intuition* is an automation tactic simplifying goals by means of intros and propositional reasoning.

13 Intuitionistic Decidability

In this chapter we construct a procedure that given a clause decides whether the clause is intuitionistically satisfiable. The procedure will be realized as an informative test $\forall C.\ \{sat\ C\} + \{tab\ C\}$ proving the main lemma. The procedure relies on the so-called subformula property of the tableau rules. If the input clause is satisfiable, the procedure will construct a demo satisfying the clause, where the demo will only contain formulas that are subformulas of formulas occurring in the input clause.

13.1 Outline

When we look at the intuitionistic tableau rules, we see that the premises of each rule contain only subformulas of formulas occurring in the conclusion of the rule.

$$\frac{}{C, \bot} \qquad \frac{}{C, s, s^-} \qquad \frac{C, s^- \qquad C, t}{C, s \to t} \qquad \frac{C^+, s, t^-}{C, s \to t^-} \qquad \frac{C}{C'}\ C \subseteq C'$$

This property is known as **subformula property** and will be crucial for the decidability result.

In the following, we will see clauses as finite sets of signed formulas, and demos as a finite sets of clauses. This has the important consequence that a clause has only finitely many subclauses. In the Coq formalization clauses will still be represented as lists, but we will arrange things such that we can exploit the finiteness property coming with the finite set view.

We assume that U is a subformula-closed clause. By subformula closedness we mean that U satisfies the following properties:

- If $s \to t$ is in U, then s^- and t are in U.
- If $s \to t^-$ is in U, then s and t^- are in U.

The subformula property of the tableau rules tells us that a tableau derivation of a subclause of U contains only subclauses of U. Since there are only finitely many subclauses of U, we can argue that tableau refutability of subclauses of U is decidable. We will discuss this point in detail later.

Since a formula has only finitely many subformulas, we can compute for every clause C a subformula-closed clause U containing C. It follows that tableau refutability is decidable for all clauses.

We call a clause **consistent** if it is not tableau refutable. Since tableau refutability is decidable, consistency of clauses is decidable. From the tableau rules it follows that every consistent clause C satisfies the following properties:

(1) C contains neither $\bot$ nor a clashing pair s and s^-.

(2) If $s \to t$ is in C, then either C, s^- or C, t is consistent.

(3) If $s \to t^-$ is in C, then C^+, s, t^- is consistent.

We now look at the set M of all maximal consistent subclauses of U. A clause C is in M if and only if it is a consistent subclause of U such that $C = D$ for every consistent subclause D of U such that $C \subseteq D$. One can verify the following properties:

(4) Every consistent subclause of U is a subclause of a clause in M. This follows from the finiteness of U.

(5) M is a demo.

(6) M satisfies every consistent subclause of U.

We call M the **canonical demo** for U.

We now construct the function proving the main lemma. Given a clause C, we decide whether C is tableau refutable. If C is tableau refutable, we have a tableau refutation and we are done. If C is consistent, we compute a subformula-closed clause U extending C. Next we compute the canonical demo M for U as described above. Since M satisfies C, we have a oroof that C is satisfiable. Thus we are done.

Formalizing the outlined proof in Coq takes considerable effort (about 550 lines). Half of the effort goes into general-purpose results for finite sets represented as lists. There are two Coq files for this chapter:

- *LFS2* Extends *LFS* with a power set representation and theorems for finite fixpoints and finite maximal extensions.
- *Deci* Extends *Semi* with syntactic closures, canonical demos, decidability of tableau refutability, and the main lemma. Requires the file *LFS2*.

13.2 Subformula Closedness

We define subformula closedness of clauses as follows.

```
Definition sf_closed' (S : sform) (C : clause) : Prop :=
  match S with
    | + Imp s t ⇒ −s ∈ C ∧ +t ∈ C
    | − Imp s t ⇒ +s ∈ C ∧ −t ∈ C
    | _ ⇒ True
  end.
Definition sf_closed (C : clause) : Prop :=
  ∀ S, S ∈ C → sf_closed' S C.
```

Next we define a function *scl* that for every clause yields a subformula-closed superclause.

```
Lemma scl_correct C :  sf_closed (scl C) ∧ C ⊆ scl C.
```

We call *scl C* the **syntactic closure** of C. The function *scl* is defined with a function *scl′* that for a signed formula S yields a subformula-closed clause containing S. The function *scl* applies *scl′* to every signed formula in its argument clause and yields the concatenation of the resulting clauses. This yields the desired result since the concatenation of subformula-closed clauses is subformula-closed.

The correctness proofs for *scl* and *scl′* involve many subcases. Making clever use of Coq's automation features they still can be obtained with compact proof scripts.

13.3 Power Set Representation

Given a subformula-closed clause U, we need a representation of the power set of U. This will be a list of lists that contains all subclauses of U (up to equivalence).

We establish the power set representation for a general base type X with decidable equality.

```
Section PowerRep.
Variable X : Type.
Context {eq_X_dec : eq_dec X}.
```

We define a function that yields the **power list** of a list.

```
Fixpoint power (U : list X ) : list ( list  X) :=
  match U with
  |  nil ⇒ [nil]
  |  x :: U' ⇒ power U' ++ map (cons x) (power U')
  end.
```

By induction on U we show that every element of *power U* is a sublist of U.

```
Lemma power_incl A U :
  A ∈ power U → A ⊆ U.
```

For the other direction we have to work harder. First of all, the other direction is not literally true. What we can show is that *power U* contains every sublist of U up to equivalence. We define a function *rep* that for every sublist of U yields an equivalent list in *power U*.

```
Definition rep (A U : list X) : list X :=
  filter (fun x ⇒ x ∈ A) U.

Lemma rep_power A U :
  rep A U ∈ power U.

Lemma rep_equi A U :
  A ⊆ U → rep A U ≡ A.
```

Exercise 13.3.1 Assume that X is a type with decidable equality and U is a list over X. Prove the following facts about *power* and *rep*.

a) nil ∈ power U

b) rep A U ⊆ A

c) A ⊆ B → rep A U ⊆ rep B U

d) A ≡ B → rep A U = rep B U

e) A ⊆ U → B ⊆ U → rep A U = rep B U → A ≡ B

f) rep (rep A U) U = rep A U

g) dupfree U → dupfree (power U)

h) A ∈ power U → dupfree U → dupfree A

i) dupfree U → A ∈ power U → rep A U = A

j) dupfree U → A ∈ power U → B ∈ power U → A ≡ B → A = B

13.4 Step Predicates and Finite Fixpoint Theorem

An inductive predicate $X \to \mathit{Prop}$ can often be analyzed with a **step predicate** $\mathit{step} : \mathit{list}\ X \to X \to \mathit{Prop}$ modeling one-step derivability. The idea is that $\mathit{step}\ A\ x$ holds if x can be obtained from the elements of A with a single rule application (i.e., there is an instance of a rule where x is the conclusion and all premises are elements of A).

We will show the decidability of tableau refutability using a step predicate. We postpone the definition of the step predicate and first elaborate the decision technique in the abstract.

For the abstract development we assume a base type with decidable equality and a monotone and decidable step predicate. We also assume a list U over the base type.

```
Section LFP.
Variable X : Type.
Context {eq_X_dec : eq_dec X}.
Variable step : list X → X → Prop.
Variable step_mono : ∀ A B x, A ⊆ B → step A x → step B x.
Context {step_dec : ∀ A x, dec (step A x)}.
Variable U : list X.
```

We use the step predicate to define an inductive predicate *lfp*.[1] We only admit derivation steps whose conclusion is in U.

```
Inductive lfp : X → Prop :=
| lfpI  A x : (∀ a, a ∈ A → lfp a) → step A x → x ∈ U → lfp x.
```

When we apply the technique to tableau refutability, we will choose for U the power list of the syntactic closure of the initial clause for wich we want to decide tableau refutability. Since the tableau rules have the subformula property, all premises of a rule instance are in U if the conclusion is in U. Thus the predicate *lfp* will hold exactly for the tableau refutable elements of U.

We show in the abstract that the predicate *lfp* is decidable. To do so, we define a **step function** that for a list A yields a list containing the elements of U that are one-step derivable from A.

```
Definition fstep (A : list X) : list X :=
  filter (step A) U.
```

If we iterate the step function n-times on the empty list, we obtain a chain

$$nil = C_0 \subseteq C_1 \subseteq \cdots \subseteq C_n \subseteq U$$

where $C_0 := nil$ and $C_{i+1} := fstep\ C_i$. The chain property follows from the monotonicity of the step function, which follows from the monotonicity of the step predicate. The key insight now is that $C_i \equiv C_{i+1}$ implies $C_{i+1} \equiv C_{i+2}$ (by the monotonicity of the step function). Thus all jumps $C_i \subsetneq C_{i+1}$ precede the stationary steps $C_i \equiv C_{i+1}$. Since there can be at most one jump per element of U, we know that the chain is stationary after at most *card U* steps.

```
Definition limit := nat_iter (card U) fstep nil.
```

```
Lemma limit_fixpoint:
  fstep limit ≡ limit.
```

From what we have said it is clear that *limit* is a least fixpoint of the step function. We can now show that *limit* contains exactly those element of X for which the predicate *lfp* holds.

[1] *lfp* stands for least fixpoint. The motivation for this technical name will be explained later in this section.

```
Lemma lfp_limit x :
  lfp x ↔ x ∈ limit.
```

Thus *lfp* is a decidable predicate.

```
Lemma lfp_dec x :
  dec (lfp x).
End LFP.
```

We will refer to *lfp_dec* as the **finite fixpoint theorem**.

13.5 Decidability of Tableau Refutability

We are now well prepared for showing the decidability of tableau refutability. First we capture the tableau predicate *tab* with a step predicate *step*.

```
Definition sup (A : list clause) (C : clause) : Prop :=
  ∃ D, D ∈ A ∧ D ⊆ C.
Definition step' (A : list clause) (S : sform) (C : clause) : Prop :=
  match S with
    | + Fal ⇒ True
    | − Var x ⇒ +Var x ∈ C
    | + Imp s t ⇒ sup A (−s :: C) ∧ sup A (+t :: C)
    | − Imp s t ⇒ sup A (+s :: −t :: pos C)
    | _ ⇒ False
  end.
Definition step (A : list clause) (C : clause) : Prop :=
  ∃ S, S ∈ C ∧ step' A S C.
```

We read *sup A C* as "*A* supports *C*". Note that *sup A C* and *step A C* are defined such that *A* can be seen as a set of sets and *C* can be seen as a set (i.e., order and duplicates don't matter). It is not difficult to show that the step predicate is monotone and decidable.

The decidability of the tableau predicate will be obtained with the finite fixpoint theorem. It suffices to show that

```
C ∈ power U → (tab C ↔ lfp step (power U) (rep C U))
```

holds for every subformula-closed clause *U*. The decidability of *tab* can then be obtained with *U* := *scl C*. We start as follows.

```
Section Decidability.
Variable U : clause.
Variable sfcU : sf_closed U.
```

Given the definition of the step predicate, it is not difficult to show that *lfp* implies *tab*.

```
Lemma lfp_tab C :
  lfp  step (power U) C → tab C.
```

This yields the direction from right to left of the equivalence. For the other direction we prove

```
Lemma tab_lfp C :
  C ⊆ U → tab C → lfp step (power U) (rep C U).
```

by induction on *tab C*. Using the generalized assumption $C \subseteq U$ is crucial for the induction to go through. From this we obtain

```
Lemma tab_dec' C :
  C ⊆ U → dec (tab C).
End Decidability.
```

which yields the general decidability of *tab* with $U := scl\ C$.

```
Lemma tab_dec C :
  dec (tab C).
```

13.6 Quasi-Maximal Extensions

Think of clauses as finite sets of signed formulas. Suppose C is a consistent subclause of a clause U. Then it is intuitively clear that there exists a maximal subclause of U that extends A and is consistent. Proving this result in type theory will take some effort. We have to construct a function that given C computes a maximal extension in U that is consistent.

We first solve the problem in the abstract. We assume a type X with decidable equality and a decidable predicate p on lists over X. We also assume a list U.

```
Section QME.
Variable X : Type.
Context {eq_X_dec : eq_dec X}.
Variable p : list X → Prop.
Context {p_dec : ∀ A, dec (p A)}.
Variable U : list X.
```

Next we define the quasi-maximal sublists of U satisfying p.

```
Definition qmax (M : list X) : Prop :=
  M ⊆ U ∧ p M ∧ ∀ x, x ∈ U → p (x::M) → x ∈ M.
```

Note that *qmax M* holds if and only if M is a sublist of U satisfying p such that M contains x whenever $x :: M$ satisfies p. This notion of maximality will suffice for our purposes.

Let A be a sublist of U satisfying p. We show that A can be extended to a quasi-maximal sublist of U satisfying p.

```
Lemma qmax_exists A :
  A ⊆ U → p A → {M | A ⊆ M ∧ qmax M}.
```

The mathematical proof is straightforward. Either A is already quasi-maximal or there is an element $x \in U$ such that $x \notin A$ and p satisfies $x :: A$. In the second case we proceed recursively with $x :: A$. The recursion terminates since U is finite and each recursion steps adds a new element of U.

The first argument of the mathematical proof of *qmax_exists* can be justified with the lemma

```
Lemma qmax_or A :
  A ⊆ U → p A →  {x | x ∈ U ∧ p (x::A) ∧ ¬ x ∈ A} + {qmax A}.
```

The proof of this lemma uses the lemmas *sigma_forall_list* and *dec_DM_impl* from Section 8.

The second argument of the mathematical proof of *qmax_exists* can be justified with size recursion (see Section 11.6) using the size function $f\,A := \mathit{card}\,U - \mathit{card}\,A$. This form of size recursion is sometimes called *slack recursion*.

Exercise 13.6.1 Assume that X is a type with decidable equality and let U be a list over X. Prove the following slack recursion principle for sublists of U.

```
Lemma slack_recursion U (t : list X → Type) :
  (∀ A, A ⊆ U → (∀ B, B ⊆ U → card A < card B → t B) → t A) →
  ∀ A, A ⊆ U → t A.
```

13.7 Canonical Demos

We now come to the construction of the canonical demo. We say that a clause is **consistent** if it is not tableau refutable.

```
Definition con C := ¬ tab C.
```

Consistency is decidable since tableau refutability is decidable. Decidability of consistency is essential for the construction of the canonical demo. We assume a subformula-closed clause U and define a model CD as the list of maximal consistent clauses in *power* U.

```
Section CanonicalDemo.
Variable U : clause.
Variable sfcU : sf_closed U.
Context {tab_dec : ∀ C, dec (tab C)}.

Definition CD : model := filter (qmax con U) (power U).
```

We will show that *CD* is a canonical demo for *U*. The key lemma for doing this is the **Extension Lemma**

```
Lemma extension C :
  C ⊆ U → con C → {C' | C' ∈ power U ∧ C ⊆ C' ∧ qmax con U C'}.
```

which says that every consistent subclause of *U* can be extended to a maximal consistent subclause of *U* in *power U*.[2] The Extension Lemma follows with the quasi-maximal extension lemma *qmax_exists*.

We can now prove that *CD* is a demo.

```
Lemma CD_demo :  demo CD.
```

The proof uses the properties of consisten clauses stated in Section 13.1. For the proof of property (2) the decidability of tableau refutability is essential. For the proof of the demo requirement for negative implications (see Section 12.4) the Extension Lemma is needed.

It remains to show that *CD* satisfies every consistent subclause of *U*.

```
Lemma CD_con_satis C :
  C ⊆ U → con C → ∃ A, A ∈ CD ∧ C ⊆ A ∧ satis' CD A C.
```

The proof extends *A* to a clause $A \in CD$ using the extension lemma. That *A* satisfies *C* in *CD* follows with the corollary *demo_satis′* of the Demo Theorem (see Section 12.4).

13.8 Proof of Main Lemma

The proof of the main lemma is now straightforward. We need the correctness lemma for the syntactic closure function, the lemma for tableau decidability, and the canonical demo lemma.

```
Lemma main C :  {sat C} + {tab C}.
Proof.
  destruct (scl_correct C) as [E F].
  assert (T : ∀ C, dec (tab C)) by apply tab_dec.
  decide (tab C) as [D|D]. tauto.
  left . destruct (CD_con_satis E F D) as [A G]. ∃ (CD (scl C)), A. tauto.
Qed.
```

[2] An extension lemma with a propositional existential quantification would suffice for our results.

13.9 Discussion and Tableau Procedure

In this chapter we presented a proof of the main lemma for intuitionistic propositional logic. The ideas of the proof were presented at an abstract level where we talk about finite sets rather than lists. The abstract presentation is needed for simplicity and clarity. The formalization of finite sets as lists and the proofs of the necessary lemmas took considerable effort. The most complex part is the decidability of tableau refutability, which requires a power set representation and a finite fixpoint theorem.

There should be a simpler proof of the main lemma using a tableau procedure generalizing the tableau procedure for the classical case. This time the procedure works on a finite set of clauses we call **pool**. The procedure decides whether all clauses in the pool are satisfiable. A pool is **failed** if it contains a clause containing $\bot$ or a clash s/s^-. A pool is **solved** if it is not failed and satisfies the following **saturation conditions**.

1. If a positive implication $s \to t$ is in a clause C in the pool, the pool contains a clause containing either C, s^- or C, t.
2. If a negative implication $s \to t^-$ is in a clause C in the pool, the pool contains a clause containing C^+, s, t^-.

It is not difficult to see that the maximal clauses of a solved pool are a demo. Thus all clauses of a solved pool are satisfiable.

The procedure now works as follows. It starts with just the input clause in the pool. If the pool is solved, the procedure stops and returns a demo satisfying all clauses in the pool. If the pool is failed, the procedures stops and returns a proof that a clause in the pool is unsatisfiable. If the pool is neither solved nor failed, a saturation condition is violated. In this case the procedure adds a clause to the pool to satisfy the violated saturation condition and recurses. The termination of the procedure follows from the fact that added clauses do not contain new subformulas.

As in the classical case, it is possible to modify the procedure such that in the negative case it returns an abstract refutation proof. The requirements on the refutation predicate correspond again to the tableau rules.

Exercise 13.9.1 Prove the main lemma with a tableau procedure.