Introduction to Computational Logic

Lecture Notes SS 2012

July 18, 2012

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

Contents

1 Introduction

This course is an introduction to constructive type theory and interactive theorem proving. It also covers classical first-order logic. For most of the course we use the proof assistant Coq.

Constructive type theory provides a programming language for expressing mathematical and computational theories. Theories consist of definitions and theorems stating logical consequences of the 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 proofs can be checked automatically. Thus a computer program can check the correctness of theorems and theories.

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.

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 and the verification of a compiler for a subset of the programming language C.

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, and proof systems.

We also consider classical first-order logic. First-order logic matters in practice since it comes with powerful automated theorem provers. First-order logic can be seen as a fragment of constructive type theory that trades expressivity for automation. First-order logic comes with a natural set-theoretic semantics that provides a basis for arguing the soundness and completeness of proof systems.

1 Introduction

2 Types and Functions

In this chapter, we take a first look at Coq and its mathematical programming language. Using types and functions, we define basic data structures such as booleans, natural numbers, and lists. Based on our definitions, we prove equational theorems, constructing the proofs in interaction with the Coq interpreter. Frequently, the functions we define are recursive and the proofs we construct are inductive.

2.1 Booleans

We start with the boolean values true and false and the boolean operations negation and conjunction. We first define these objects in ordinary mathematical language. To start with, we fix two different values *true* and *false* and define the set $bool := \{true, false\}$. Next we define the operations negation and conjunction by stating their types and defining equations.

$$\begin{array}{ll} \neg : bool \to bool & \wedge : bool \to bool \to bool \\ \neg true = false & true \wedge y = y \\ \neg false = true & false \wedge y = false \end{array}$$

In general, there is more than one possibility to choose the defining equations of an operation. We require that for every application of an operation exactly one of the defining equations applies from left to right. For instance, given $true \wedge false$, the first defining equation of $\wedge$ applies and yields $true \wedge false = false$.

Our presentation of the booleans translates into three definitions in Coq.

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

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

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

The first definition (starting with the keyword *Inductive*) defines a type bool that has two members false and true. The remaining two definitions (starting with the keyword *Definition*) define two functions negb and andb representing the operations negation and conjunction. The defining equations of the operations are expressed with so-called **matches**. Altogether, the definitions introduce 5 identifiers, each equipped with a unique type:

$$\begin{aligned} \mathit{bool} &: \mathit{Type} \\ \mathit{true} &: \mathit{bool} \\ \mathit{false} &: \mathit{bool} \\ \mathit{negb} &: \mathit{bool} \to \mathit{bool} \\ \mathit{andb} &: \mathit{bool} \to \mathit{bool} \to \mathit{bool} \end{aligned}$$

It is time that you start a Coq interpreter. Enter the 3 definitions one after the other. Each time Coq checks the well-formedness of the definition. Once Coq has accepted the definitions, you can explore the defined objects by entering commands that check and evaluate terms (i.e., expressions).

```
Check negb true.
% negb true : bool
Compute negb true.
% false : bool
Compute negb (negb true).
% true : bool
Compute andb (negb false) true.
% true : bool
```

Note that functions are applied without writing parentheses and that multiple arguments are not separated by commas. Functions that take more than one argument can also be applied to a single argument.

```
Check andb (negb false).
% andb(negb false) : bool → bool
Compute andb (negb false).
% fun y : bool ⇒ y  :  bool → bool
```

The term *fun y* : *bool* ⇒ *y* decribes a function *bool* → *bool* that returns its argument. Terms that start with the keyword *fun* are called abstractions and can be used freely in Coq.

```
Compute (fun x : bool => andb x x) true
% true : bool
```

2.2 Proof by Case Analysis and Simplification

From our definitions it seems clear that the equation $\neg\neg x = x$ holds for all booleans x. To verify this claim, we perform a case analysis on x.

1. $x = true$. We have to show $\neg\neg true = true$. This follows with the defining equations of negation: $\neg\neg true = \neg false = true$.
2. $x = false$. We have to show $\neg\neg false = false$. This follows with the defining equations of negation: $\neg\neg false = \neg true = false$.

To carry out the proof with Coq, we state the claim as a lemma.

```
Lemma negb_negb (x : bool) :
negb (negb x) = x.
```

The identifier negb_negb serves as the name of the lemma. Once you enter the lemma, Coq switches to proof mode and you see the initial proof goal. Here is a **proof script** that constructs the proof of the lemma.

```
Proof. destruct x. simpl. reflexivity. simpl. reflexivity. Qed.
```

At this point, it is crucial that you step through the proof script with Coq. The script begins with the command *Proof* and ends with the command *Qed*. The commands between *Proof* und *Qed* are called **tactics**. The tactic *destruct x* does the case analysis and replaces the initial goal with two subgoals, one for $x = false$ and one for $x = true$. Once you have entered *destruct x*, you will see the first subgoal on the screen. The tactic simpl simplifies the equation we have to prove by applying the definition of negb. For the first subgoal, we are now left with the trivial equality $false = false$, which is established with the tactic reflexivity. The second subgoal is established analogously.

It is important that you step back and forth in the proof script with Coq and observe what happens. This way you can see how the proof advances. At each point in the proof you are confronted with a **proof goal**, which consists of some **assumptions** (possibly none) and a **claim**. Here is the sequence of proof goals you will see when you step through the proof script.

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

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

In each goal, the assumptions appear above and the claim appears below the rule. We can shorten the proof script by combining the tactics *destruct x* and *simpl* with the **semicolon operator**.

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

The semicolon operator applies *simpl* to each of the two subgoals generated by *destruct x*. Given the symmetry of the two subgoals, we can shorten the proof script further.

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

Since the tactic *reflexivity* first simplifies the equation it is applied to, we can shorten the proof script even further.

```
Proof. destruct x ; 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.

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 2.2.1 (Commutativity of conjunction) Prove $x \wedge y = y \wedge x$ in Coq.

Exercise 2.2.2 (Disjunction) 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 the de Morgan law $\neg(x \vee y) = \neg x \wedge \neg y$ in Coq.

2.3 Natural Numbers and Structural Recursion

Dedekind and Peano discovered that the natural numbers can be obtained with two constructors O and S. The idea is best expressed with the definition of a type nat in Coq.

```
Inductive nat : Type :=
| O : nat
| S : nat -> nat.
```

The constructor O represents the number 0, and the constructor S yields the **successor** of a natural number (i.e., $Sn = n + 1$). 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 elements of nat 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 from finitely many constructors.

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

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

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

Given the constructor represention of the natural numbers, we can define the operations addition and multiplication:

$$\begin{array}{ll} + : nat \to nat \to nat & \cdot : nat \to nat \to nat \\ 0 + y = y & 0 \cdot y = O \\ Sx + y = S(x + y) & Sx \cdot y = x \cdot y + y \end{array}$$

The defining equations become clear if one thinks of Sx as $x + 1$. Here is a computation that applies the defining equations for $+$:

$$S(S(SO)) + y = S(S(SO) + y) = S(S(SO + y)) = S(S(Sy))$$

One says that the operations $+$ and $\cdot$ are defined by **structural recursion** over the first argument. The recursion comes from the second defining equation where the operation to be defined also appears on the right. Since each recursion step strips off a constructor S, the recursion must terminate. The mathematical definitions of addition and multiplication carry over to Coq:

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

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

We use the keyword *Fixpoint* in place of the keyword *Definition* to enable recursion. Coq permits only structural recusion. This way Coq makes sure that the evaluation of recursive functions always terminates. Structural recursion always happens on an argument taken from an inductive type (a type defined with the keyword *Inductive*). Each recursion step in the definition of a recursive function must take off at least one constructor.

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. One says that the short form is syntactic sugar for the long form. 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. Without the order sensitivity the second rule in the sugared match would be incorrect.

You cannot define the same identifier twice in a Coq session. Thus you can enter either the long or the short definition of leb, but not both. If you want to have both definitions, choose a different name for the second definition you enter.

Exercise 2.3.1 Define functions as follows.

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 $mod3 : nat \to nat$ that yields the remainder of x on division by 3.

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*.

2.4 Proof by Structural Induction and Rewriting

Consider the proof goal

$$\frac{x : nat}{px}$$

where px is a claim that depends on x. By **structural induction on** x we can reduce the goal to two subgoals.

$$\frac{}{pO} \qquad \frac{\begin{array}{c} x : nat \\ IHx : px \end{array}}{p(Sx)}$$

This reduction is like a case analysis on the structure of x, but has the added feature that the second subgoal comes with an extra assumption *IHx* known as **inductive hypothesis**. We think of *IHx* as a proof of px. If we can prove both subgoals, we have established the initial claim px for all $x : nat$. This can be seen as follows.

1. The first subgoal gives us a proof of pO.
2. The second subgoal gives us a proof of $p(SO)$ from the proof of pO.
3. The second subgoal gives us a proof of $p(S(SO))$ from the proof of $p(SO)$.
4. After finitely many steps we arrive at a proof of px.

This reasoning is valid since the proof of the second subgoal is a function that given an x and a proof of px yields a proof of $p(Sx)$. Here is our first inductive proof in Coq.

```
Lemma plus_O (x : nat) :  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.

$\dfrac{x : nat}{plus\ x\ O = x}$	$\dfrac{}{O = O}$	$\dfrac{\begin{array}{c} x : nat \\ IHx : plus\ x\ O = x \end{array}}{S(plus\ x\ O) = Sx}$	$\dfrac{\begin{array}{c} x : nat \\ IHx : plus\ x\ O = x \end{array}}{Sx = Sx}$
induction x ; simpl	reflexivity	rewrite IHx	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 such proofs. Here is our next example.

```
Lemma plus_S (x y : nat) :  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.

Note that the lemmas plus_O and plus_S provide the symmetric versions of the defining equations of plus. Using the lemmas, we can prove that addition is commutative.

```
Lemma plus_com (x y : nat) :  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. Given that the definition of *plus* is not symmetric, the commutativity of *plus* is an interesting result. Next we prove that addition is associative.

```
Lemma plus_asso (x y z: nat) :  plus (plus x y) z = plus x (plus y z).

Proof. induction x ; simpl. reflexivity. rewrite IHx. reflexivity.  Qed.
```

Rewriting with plus_com can be tricky since the lemma applies to every sum. This can be resolved by instantiating the lemma. Here is an example.

```
Lemma plus_AC (x y z : nat) :
plus  y (plus x z)  = plus (plus z y) x.

Proof. rewrite (plus_com z). rewrite (plus_com x). rewrite plus_asso. reflexivity. Qed.
```

Note that the instantiated lemma plus_com z can only rewrite terms of the form plus z _. Here is a more involved example using the tactic f_equal and (partially) instantiated lemmas.

```
Lemma plus_AC' (x y z : nat) :
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.
rewrite (plus_com _ (plus _ _)). rewrite plus_asso. f_equal.
rewrite plus_com. reflexivity. Qed.
```

Run the proof script to see the effects of the tactics. The tactic f_equal reduces a claim $st = su$ to $t = u$. The first rewrite with plus_com requires that the second argument of plus is of the form plus _ _.

Exercise 2.4.1 Prove Lemma plus_com by induction on y.

Exercise 2.4.2 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).
```

Exercise 2.4.3 Often a claim must be generalized before it can be proven by induction. For instance, it seems impossible to prove $plus\ (plus\ x\ x)\ x = plus\ x (plus\ x\ x)$ without using lemmas. However, a more general claim expressing the associativity of addition with three variables has a straightforward inductive proof (see lemma plus_asso).

2.5 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 functions

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

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

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

One says that *pair* is a **polymorphic function**. 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 O true* for the pair $(O, true)$. Fortunately, Coq comes with a **type inference feature** making it possible to just write *pair O true* and leave it to the interpreter to insert the missing arguments. One speaks of **implicit arguments**. With the command

```
Set Implicit Arguments.
```

we tell Coq to make all arguments implicit that can be derived from other arguments. If the definition of products and pairs appears *after* this command, the first two arguments of *pair* are automatically treated as implicit arguments.

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

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

```
Check @pair nat.
% @pair nat : forall Y : Type, nat → Y → prod nat Y

Check @pair nat bool O.
% @pair nat bool O : bool → prod nat bool
```

We can see that Coq automatically inserts implicit arguments by telling Coq to not use notational conveniences when displaying terms.

```
Set Printing All.

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

Unset Printing All.
```

Here are functions that yield the first and the second component of a pair.

```
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. reflexivity. Qed.
```

Here is a function that swaps the components of a pair:

```
Definition swap (X Y : Type) (p : prod X Y) : prod Y X := pair (snd p) (fst p).

Compute swap (pair O true).
% pair true nat : prod bool nat

Lemma swap_swap (X Y : Type) (p : prod X Y) :
swap (swap p) = p.

Proof. destruct p. unfold swap. simpl. reflexivity. Qed.
```

Note the use of the tactic *unfold*. We use it since *simpl* does not simplify applications of functions that do not involve a match. Since *reflexivity* does all the required simplification automatically, we may omit the unfold and simplification step.

Exercise 2.5.1 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 functions

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

that translate between the cascaded and cartesian representation and prove the following lemmas.

```
Lemma car_P (X Y Z :Type) (f : X -> Y -> Z) (x :X) (y :Y) : car f (pair x y) = f x y.

Lemma cas_P (X Y Z :Type) (f : prod X Y -> Z) (x :X) (y :Y) : cas f x y = f (pair x y).
```

The type arguments of *car* and *cas* are assumed to be implicit.

2.6 Iteration

We now define a function *iter* that takes a natural number n, a type X, a function $f : X \to X$, and a value $x : X$, and yields the value obtained by applying the function f n times to x. The defining equations for *iter* are as follows (type argument suppressed):

$$iter\ 0\ f\ x = x$$
$$iter\ (Sn)\ f\ x = f(iter\ n\ f\ x)$$

The Coq definition is now straightforward:

```
Fixpoint iter (n : nat) (X : Type) (f : X -> X) (x : X) : X :=
match n with
| O => x
| S n' => f (iter n' f x)
end.
```

With *iter* we can give non-recursive definitions of addition and multiplication.

```
Definition plusi (x y : nat) : nat := iter x S y.

Definition multi (x y : nat) : nat := iter x (plusi y) O.
```

The function *plusi* obtains $x + y$ by iterating the function $S\ x$ times on y. The function *multi* obtains $x \cdot y$ by iterating the function $+y\ x$ times on 0.

```
Lemma iter_plus (x y : nat) :
plus x y = iter x S y.

Proof. induction x ; simpl. reflexivity. rewrite IHx. reflexivity. Qed.
```

We can see *iter* n as a functional representation of the number n that carries with it the structural recursion coming with n. The following definitions implement this idea.

```
Definition Nat := forall X : Type, (X -> X) -> X -> X.

Definition encode : nat -> Nat := iter.

Definition decode : Nat -> nat := fun f => f nat S O.

Compute decode (encode (S (S O))).
% S(S O) : nat

Lemma iter_coding (x : nat) :
decode (encode x) = x.

Proof. unfold encode. unfold decode. induction x ; simpl.
reflexivity. rewrite IHx. reflexivity. Qed.
```

A **higher-order function** is a function that takes a function as argument. The function *iter* is our first example of a higher-order function. It formulates a recursion scheme known as *iteration* or *primitive recursion*. Note that *iter* is also polymorphic.

Exercise 2.6.1 Prove *mult x y* = *iter x* (*plus y*) *O* for all numbers x and y.

Exercise 2.6.2 Define a function *power* recursively (see Exercise 2.3.1) and prove power x n = iter n (mult x) (S O) for all $x, n : nat$.

Exercise 2.6.3 Prove the following lemma.

```
Lemma iter_move (X : Type) (f : X -> X) (x : X) (n : nat) :
iter (S n) f x = iter n f (f x).
```

Exercise 2.6.4 (Subtraction with Iteration) Prove the following lemmas about a subtraction function defined with iter.

```
Definition minus (x y : nat) : nat := iter y pred x.

Lemma minus_O (y : nat) : minus O y = O.

Lemma minus_O' (x : nat) : minus x O = x.

Lemma minus_SS (x y : nat) : minus (S x) (S y) = minus x y.

Lemma minus_SP (x y : nat) :  minus x (S y) = pred (minus x y).

Lemma minus_SP' (x y : nat) :  minus x (S y) = minus (pred x) y.

Lemma minus_PS (x y : nat) :  minus x y = pred (minus (S x) y).
```

Hint: Do unfold minus as first step in your proofs.

2.7 Factorials with Iteration

We define the factorial $n!$ of a natural number n by a recursive function:

```
Fixpoint fac (n : nat) :  nat :=
match n with
| O => S O
| S n' => mult n (fac n')
end.
```

We can compute factorials with iter if we iterate on pairs:

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

We realize the idea with two definitions.

```
Definition step (p : prod nat nat) : prod nat nat :=
match p with pair n f => pair (S n) (mult (S n) f) end.

Definition ifac (n :  nat) :  nat := snd (iter  n step (pair O (S O))).
```

To verify the correctness of the iterative computation of factorials, we would like to prove $ifac\ n = fac\ n$ for $n : nat$. An attempt to prove the claim directly fails miserably. The problem is that we need to account for both components of the pairs computed by *iter*. To do so, we prove the following lemma.

```
Lemma iter_fac (n : nat) :
pair n (fac n) = iter  n step (pair O (S O)).

Proof.
induction n. reflexivity.
simpl iter.   rewrite <- IHn. unfold step. reflexivity.
Qed.
```

To avoid large and unreadable terms, the proof simplifies only the application of *iter*. The command *unfold step* can be omitted; it is included to help your understanding when you step through the proof.

It is now straightforward to prove that *ifac* and *fac* agree on all arguments.

Exercise 2.7.1 Prove the following lemmas.

```
Lemma ifac_fac (n : nat) : ifac n = fac n.
```

```
Lemma ifac_step (n : nat) : step (pair n (fac n)) = pair (S n) (fac (S n)).
```

2.8 Lists

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

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

All elements of a list must be taken from the same type. The constructor nil represents the empty sequence, and the constructor cons represents nonempty sequences.

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

Coq does not automatically treat the type argument of *nil* as implicit argument since there is no other argument where the type can be obtained from. So we force Coq to treat the argument of *n* as implicit:

```
Implicit Arguments nil [X].
```

Now Coq will try to derive the argument of *nil* from the context surrounding an occurrence of *nil*. Here are functions defining the length, the concatenation, and the reversal of lists.

```
Fixpoint length (X : Type) (xs : list X) : nat :=
match xs with
| nil => O
| cons _ xr => S (length xr)
end.
```

```
Fixpoint app (X : Type) (xs ys : list X) : list X :=
match xs with
| nil => ys
| cons x xr => cons x (app xr ys)
end.

Fixpoint rev (X : Type) (xs : list X) : list X :=
match xs with
| nil => nil
| cons x xr => app (rev xr) (cons x nil)
end.
```

Using informal notation for lists, we have the following.

$$\begin{aligned} \mathit{length}\,[x_1,\ldots,x_n] &= n \\ \mathit{app}\,[x_1,\ldots,x_m]\,[y_1,\ldots,y_n] &= [x_1,\ldots,x_m,y_1,\ldots,y_n] \\ \mathit{rev}\,[x_1,\ldots,x_n] &= [x_n,\ldots,x_1] \end{aligned}$$

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) (xs : list X) : app xs nil = xs.

Proof. induction xs ; simpl. reflexivity. rewrite IHxs. reflexivity. Qed.
```

Exercise 2.8.1 Prove the following lemmas.

```
Lemma app_asso (X : Type) (xs ys zs : list X) :
app (app xs ys) zs = app xs (app ys zs).

Lemma length_app (X : Type) (xs ys : list X) :
length (app xs ys) = plus (length xs) (length ys).

Lemma rev_app (X : Type) (xs ys : list X) :
rev (app xs ys) = app (rev ys) (rev xs).

Lemma rev_rev (X : Type) (xs : list X) :
rev (rev xs) = xs.
```

2.9 Linear List Reversal

We will now see inductive proofs where the inductive hypothesis carries a universal quantification. Such proofs are needed for the verification of the correctness of tail-recursive procedures for list reversal and list length. The proofs will employ the tactics revert and intros.

If you are familiar with functional programming, you will know that the function rev takes quadratic time to reverse a list since 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 accumultate the elements of the main list in an extra argument.

```
Fixpoint revi (X : Type) (xs ys : list X) : list X :=
match xs with
| nil => ys
| cons x xr => revi xr (cons x ys)
end.
```

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

```
Lemma revi_rev (X : Type) (xs ys : list X) :
revi xs ys = app (rev xs) ys.
```

We prove this lemma by induction on xs. For the induction to go through, the inductive hypothesis must hold for all ys. To get this property, we move the universal quantification for ys from the assumptions to the claim before we issue the induction. We do this with the tactic revert ys.

```
Proof. revert ys. induction xs ; simpl.
intros ys. reflexivity.
intros ys. rewrite IHxs. rewrite app_asso. reflexivity. Qed.
```

Step through the proof script to see how it works. The tactic intros ys moves the universal quantification for ys from the claim back to the assumptions.

Exercise 2.9.1 Prove the following lemma.

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

The lemma tells us how we can reverse lists with revi.

Exercise 2.9.2 Here is a tail-recursive function that obtains the length of a list with an accumulator argument.

```
Fixpoint lengthi (X : Type) (xs : list X) (a : nat) :=
match xs with
| nil => a
| cons _ xr => lengthi xr (S a)
end.
```

Proof the following lemmas.

```
Lemma lengthi_length (X : Type) (xs : list X) (a : nat) :
lengthi xs a = plus (length xs) a.

Lemma length_lengthi (X : Type) (xs : list X) :
length xs = lengthi xs O.
```

Exercise 2.9.3 Define a tail-recursive function *faci* that computes factorials. Prove $fac\ n = faci\ n\ O$ for $n : nat$. Hint: First you need a lemma that characterizes *faci* non-recursively using *fac*.

2.10 Options and Finite Types

An empty type can be defined as an inductive type that has no constructors.

```
Inductive void : Type := .
```

Computationally, *void* seems useless. Logically, however, *void* is dynamite. If we assume that *void* has a member, we can prove that every equation holds. In other words, if we assume that void is inhabited, logical reasoning crashes.

```
Lemma void_vacuous (v : void) (X : Type) (x y : X) :  x=y.
```

```
Proof. destruct v. Qed.
```

The proof is by case analysis on the assumed member v of void. To prove a claim by case analysis on a member of an inductive type, we need to prove the claim for every constructor of the type. Since void has no constructor, the claim follows vacuously.[1] Vacuous reasoning is a basic logical principle.

Next we consider a type constructor *option* that adds a new element to a type.

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

The constructor *Some* yields the old elements and the constructor *None* yields the new element (none of the old elements). The elements of an option type are called *options*.

Option types can be used to represent partial functions. Here is such a representation of the subtraction function.

```
Implicit Arguments None [X].
```

```
Fixpoint subopt (x y : nat) : option nat :=
match x, y with
| _, O => Some x
| O, _ => None
| S x', S y' => subopt x' y'
end.
```

[1] 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.

If one iterates the type constructor *option* on *void* n times, one obtains a type with n elements.

```
Definition fin (n : nat) : Type := iter n option void.
```

Here are definitions naming the elements of the types $fin(S\,O)$, $fin(S(S\,O))$, and $fin(S(S(S\,O)))$.

```
Definition a11 : fin (S O) := None.
Definition a21: fin (S (S O))  := Some a11.
Definition a22 : fin (S (S O)) := None.
Definition a31: fin (S (S (S O))) := Some a21.
Definition a32 : fin (S (S (S O))) := Some a22.
Definition a33 : fin (S (S (S O))) := None.
```

Exercise 2.10.1 Define a predecessor function $nat \to option\ nat$.

Exercise 2.10.2 Prove the following lemma.

```
Lemma fin_SO (x : fin (S O)) :  x = None.
```

Exercise 2.10.3 One can define a bijection between *bool* and $fin(S(S\,O))$. Show this fact by completing the definitions and proving the lemmas shown below.

```
Definition tofin (x : bool) :  fin  (S(S O)) :=
Definition fromfin (x : fin (S(S O))) :  bool :=
Lemma bool_fin (x : bool) :  fromfin (tofin x) = x.
Lemma fin_bool (x : fin (S(S O))) :  tofin (fromfin x) = x.
```

Exercise 2.10.4 One can define a bijection between *nat* and *option nat*. Show this fact by completing the definitions and proving the lemmas shown below.

```
Definition tonat (x : option nat) : nat :=
Definition fromnat (x : nat) : option nat :=
Lemma opnat_nat (x : option nat) : fromnat (tonat x) = x.
Lemma nat_opnat (x : nat) : tonat (fromnat x) = x.
```

2.11 Fun and Fix

Coq has terms describing functions. The definitions of functions we have seen so far all reduce to plain declarations

```
Definition x : t := s.
```

where x is the name of the function, t is the type of the function, and s is a term describing the function. Here are two examples.

```
Definition plus2 : nat -> nat :=
fun x : nat => S (S x).

Definition double : nat -> nat :=
fix f (x : nat) : nat :=
match x with
| O => O
| S x' => S (S ( f x'))
end.
```

Terms starting with the keyword *fun* are used to describe non-recursive functions, and terms starting with the keyword *fix* are used to describe recursive functions. While terms with *fun* do not specify a name for the function described, terms with *fix* do so that recursion can be expressed. Terms with *fix* also specify the return type of the function described.

Coq's print command displays the definition of a name. When you print the previously defined names *andb* and *plus*, you will observe that Coq has translated our definitions to plain definitions.

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

Print plus.
plus =
fix plus (x y : nat) : nat :=
  match x with
  | O => y
  | S x' => S (plus x' y)
  end
: nat -> nat -> nat
```

2.12 Standard Library

Coq comes with an extensive library that provides all the data types we have seen in this chapter. So there is no need to define these types. You may use the print command to display the definition of predefined names. When using *Print*, you will see a few things we have not explained so far. For instance, Coq may say *Set* where we have written *Type*. For now, just think of *Set* as a synonym for *Type*.

For natural numbers, Coq's library provides the usual notations. For instance, you may write $2 + 3 * 2$ for *plus* $(S(S\ O))$ $(mult\ (S(S(S\ O)))\ (S(S\ O)))$.

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

```
Definition andb (x y : bool) : bool :=
 if  x then y else false.
```

You may use the command *Set Printing All* to get rid of the notational sugar.

```
Set Printing All.

Check 2+3*2.
% plus (S(S O)) (mult (S(S(S O))) (S(S O))) : nat

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

If you execute the above commands in an environment 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.

2.13 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 constructor representation of the natural numbers employed in the Peano axioms. Inductive types are also a basic feature of functional programming languages (e.g., ML, Haskell).

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 higher-order function *iter* that formulates 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 strip off at least one constructor of a given argument.

Coq's language is very regular. Both functions and types are first-class values, and functions can take types and functions as arguments.

Coq provides for the formulation and proof of theorems. So far we have seen equational theorems. As it comes to proof techniques, we have used simplification, case analysis, induction, and rewriting. Proofs are constructed by proof scripts, which are obtained with commands called tactics. A tactic either resolves a trivial 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 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, use the command *Print L* to see the term serving as the proof of a lemma *L*.

2.14 Tactics Summary

destruct x	Do case analysis on x
induction x	Do induction on x
rewrite [<-] *s*	Rewrite claim with an equation obtained from s
f_equal	Reduce claim $st = su$ to $t = u$
simpl [x | t]	Simplify [applications of x | subterm t] in claim
unfold x	Unfold definition of x in claim
intros x	Move universal quantification from claim to assumptions
revert x	Move universal quantification for x from assumptions to claim
reflexivity	Establish the goal by computation and reflexivity of equality
t_1 ; t_2	Combines tactics t_1 and t_2 into one tactic

3 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.

3.1 Logical Operations

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 truth values. Coq and our presentation of logic follow the constructive approach. The cornerstone of the constructive approach is the **BHK-scheme**,[1] wich 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.
- A proof of $\forall x : T.A$ is a function that for every $x : T$ yields a proof of A.

[1] The name BHK-scheme 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 $\exists x : T.A$ 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 the other operations:

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

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.

3.2 Implication and Universal Quantification

Example: Symmetry of Equality

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

```
Goal forall (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 3.2.1 Prove the following goal.

```
Goal forall 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 forall 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 forall X Y Z : Prop, (X -> Y) -> (Y -> Z) -> X -> Z.
```

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

Exercise 3.2.2 Prove that equality is transitive.

3.3 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 forall p q : nat -> Prop,
p 7 -> (forall x, p x -> q x) -> q 7.
```

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

[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 3.1 that proofs of implications are functions.

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 $p\ 7$.

3.4 The Apply Tactic

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

```
Goal forall 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 forall p q : nat -> Prop, p 7 -> (forall 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 3.4.1 Prove the following goals.

```
Goal forall X Y,
(forall Z, (X -> Y -> Z) -> Z) -> X.
```

```
Goal forall X Y,
(forall Z, (X -> Y -> Z) -> Z) -> Y.
```

Exercise 3.4.2 Prove the following goals, which express essential properties of booleans, numbers, and lists.

```
Goal forall (p : bool -> Prop) (x : bool),
p true -> p false -> p x.
```

```
Goal forall (p : nat -> Prop) (x : nat),
p O -> (forall n, p n -> p (S n)) -> p x.
```

```
Goal forall (X : Type) (p : list X -> Prop) (xs : list X),
p nil -> (forall x xs, p xs -> p (cons x xs)) -> p xs.
```

Hint: Use case analysis and induction.

3.5 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 forall (X : Type) (x y : X),
(forall  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 3.5.1 Prove the following goals.

```
Goal forall (X : Type) (x y : X),
x=y -> forall p : X -> Prop, p x -> p y.

Goal forall (X : Type) (x y : X),
(forall  p : X -> Prop, p x -> p y) ->
forall  p : X -> Prop, p y -> p x.
```

3.6 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 3.1). 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*.

An 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 3.6.1 Prove the following goals in Coq. Explain what you see.

```
Goal forall X : Type,
(fun x : X => x) = (fun y : X => y)

Goal forall X Y : Prop,
(X -> Y) -> forall x : X, Y.

Goal forall X Y : Prop,
(forall x : X, Y) -> X -> Y.

Goal forall X Y : Prop,
(X -> Y) = (forall x : X, Y).
```

3.7 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 forall 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 forall 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 forall X:Prop,
~~X -> (X -> ~X) -> X.
```

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

Exercise 3.7.1 Prove the following goals.

```
Goal forall X : Prop, ~~~X -> ~X.
```

```
Goal forall X Y : Prop, (X -> Y) -> ~Y -> ~X.
```

3.8 Conjunction, Disjunction, and Equivalence

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

```
Goal forall 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 forall 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 forall X Y : Prop, X /\ Y -> Y /\ X.
Proof. intros X Y [x y]. split. exact y. exact x. Qed.

Goal forall 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 forall 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.
```

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.

```
Goal forall 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.

Goal forall 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.
```

Exercise 3.8.1 Prove the following goals.

```
Goal forall X Y : Prop,
X /\ (X \/ Y) <-> X.

Goal forall X Y : Prop,
X \/ (X /\ Y) <-> X.

Goal forall X : Prop,
~(X \/ ~X) -> X \/ ~X.

Goal forall X : Prop,
(X \/ ~X -> ~(X \/ ~X)) -> X \/ ~X.

Goal forall X:Prop,
(X -> ~X) -> X <-> ~~X.

Goal forall X Y Z W : Prop,
(X -> Y) \/ (X -> Z) -> (Y -> W) /\ (Z -> W) -> X -> W.
```

Exercise 3.8.2 (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 <-> forall Z : Prop, Z.

Goal forall X : Prop,
~X <-> forall Z : Prop, X -> Z.

Goal forall X Y : Prop,
X /\ Y <-> forall Z :Prop, (X -> Y -> Z) -> Z.

Goal forall X Y : Prop,
X \/ Y <-> forall Z : Prop, (X -> Z) -> (Y -> Z) -> Z.

Goal forall (X : Type) (x y : X),
x=y <-> forall p : X -> Prop, p x -> p y.
```

3.9 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 forall 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 forall (X : Type) (p : list X -> Prop) (xs : list X),
p nil -> (forall 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 forall X : Prop, ~ (X <-> ~X).

Proof. tauto. Qed.
```

3.10 Existential Quantification

The tactics for existential quantifications are *destruct* and *exists*.

```
Goal forall (X : Type) (p q : X -> Prop),
(exists x, p x /\ q x) -> exists x, p x.

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

Run the proof scripts with Coq to understand.

Russell's law is a simple fact about nonexistence that has amazing consequences.[5] One such consequence is the undecidability of the halting problem. We state Russell's law as follows:

```
Definition Russell : Prop := forall (X : Type) (p : X -> X -> Prop),
~ exists x, forall 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, Russell's law says that there is no Turing machine x such that x halts on a Turing machines y if and only if y does not halt on its string representation.

The proof of Russell's law is not difficult.

```
Lemma circuit (X : Prop) : ~ (X <-> ~X).

Proof. tauto. Qed.

Goal Russell.

Proof. intros X p [x A]. apply (@circuit (p x x)). exact (A x). Qed.
```

We can prove Russell's law without a lemma if we use the tactic *specialize*.

```
Goal Russell.

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

A **disequation** $s \neq t$ is a negated equation $\neg(s=t)$. Coq's notation for disequations is $s<>t$. We prove the correctness of a characterization of disequality that employs existential quantification.

```
Goal forall (X : Type) (x y : X),
x <> y <-> exists p : X -> Prop, p x /\ ~p y.

Proof. split.
intros A. exists (fun z => x = z). now auto.
intros [p [A B]] C. apply B. rewrite <- C. apply A. Qed.
```

[5] One amazing consequence of Russell's law is that naive set theory is inconsistent. In naive set theory the Russell set $\{x | x \notin x\}$ violates Russell's law. If one can form such a Russell set in a theory, the inconsistency result is known as Russell's paradox.

The proof uses two features of Coq's tactic language we have not seen so far. First, note that *split* tacitly introduces X, x, and y. Second, note the *now* in front of the *auto*. The proof will go through if we omit the *now*, but with the *now* we make explicit that *auto* must solve the current goal. The *now* modifies *auto* into a **solve-or-fail** tactic like *reflexivity*, *exact*, *assumption*, or *tauto*. The solve-or-fail tactics appear in red color in our typesetting of Coq code.

Exercise 3.10.1 Prove the De Morgan law for existential quantification.

```
Goal forall (X : Type) (p : X -> Prop),
~(exists x, p x) <-> forall x, ~ p x.
```

Exercise 3.10.2 Prove the exchange rule for existential quantifications.

```
Goal forall (X Y : Type) (p : X -> Y -> Prop),
(exists x, exists y, p x y) <-> exists y, exists x, p x y.
```

Exercise 3.10.3 (Impredicative Characterization) Prove the following goal. It shows that existential quantification can be expressed with implication and universal quantification.

```
Goal forall (X : Type) (p : X -> Prop),
(exists x, p x) <-> forall Z : Prop, (forall x, p x -> Z) -> Z.
```

Exercise 3.10.4 Below are characterizations of equality and disequality based on reflexive relations. Prove the correctness of the characterizations.

```
Goal forall (X : Type) (x y : X),
x = y <-> forall r : X -> X -> Prop, (forall z : X, r z z) -> r x y.
```

```
Goal forall (X : Type) (x y : X),
x <> y <-> exists r : X -> X -> Prop, (forall 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 3.10.5 Prove the following goal.

```
Goal forall (X: Type) (x : X) (p : X -> Prop), exists q : X -> Prop,
q x /\ (forall y, p y -> q y) /\ forall y, q y -> p y \/ x = y.
```

Exercise 3.10.6

a) Prove the following goal.

```
Goal forall (X : Type) (Y : Prop) ,
X -> Y <-> (exists x : X, True) -> Y.
```

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

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

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

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

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

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

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

Figure 3.1: Basic proof rules

3.11 Basic Proof Rules

By now we have conducted many proofs in Coq. In this chapter we mostly proved general properties of the logical operations. 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 3.1. 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 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^x_t 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*.

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

Each row in Figure 3.1 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 3.1 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 operations are defined on top of the basic logical operations.

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

The proof rules in Figure 3.1 were first formulated and studied by Gerhard Gentzen in 1935. They are known as *intuitionistic natural deduction rules*.

Exercise 3.11.1 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 3.1 in this fashion.

3.12 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 forall 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 above proofs, 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 -> exists x, p x.

Proof. intros x A. exists x. exact A. Qed.

Lemma ExE (X : Type) (p : X -> Prop) (U : Prop) :
(exists x, p x) -> (forall 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 forall (X : Type) (p q : X -> Prop),
(exists x, p x /\ q x) -> exists x, p x.
```

```
Proof. intros X p q A.
apply (ExE _ A). intros x B.
apply (AndE B). intros C _.
apply (ExI _ x).  exact C. Qed.
```

Note the underlines in the applications of *ExE* and *ExI*. They delegate it to Coq to fill in the predicates needed as arguments.

Exercise 3.12.1 Fill in the underlines in the above proof script.

Exercise 3.12.2 Formulate the introduction and elimination rules for disjunctions as lemmas and use the lemmas to prove the commutativity of disjunction.

Exercise 3.12.3 The tactics *reflexivity* and *rewrite* implement the following proof rules for equations.

$$\frac{}{s = s} \qquad\qquad \frac{s = t \qquad u_t^x}{u_s^x}$$

a) Formulate a lemma *EqI* expressing the introduction rule for equations.

b) Formulate a lemma *EqE* expressing the elimination rule for equations.

c) Prove symmetry and transitivity of equality using the lemmas *EqI* and *EqE*. Do not use the tactics *reflexivity* and *transitivity*.

3.13 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.[6]

```
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.

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

```
Goal forall 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 forall 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 2.10 where we considered the type *void*.

Coq defines conjunction and disjunction as inductive predicates (i.e., inductive type constructors into *Prop*).[7]

```
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-scheme: 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 2.5.

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

```
Inductive ex (X : Type) (p : X -> Prop) : Prop :=
| ex_intro : forall 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 3.12).

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).
```

[7] Use the commands *Set Printing All* and *Print* to find out the definitions of the infix notations "$\wedge$" and "$\vee$".

There is only one family of propositions left for which we have not said how it is represented in Coq: Equations. Coq represents equations $s = t$ as inductive propositions *eq s t* where only trivial equations *eq s s* have proofs.

```
Inductive eq (X : Type) (x : X) : X -> Prop :=
| eq_refl : eq x x.
```

The tactic *reflexivity* is realized as application of the proof constructor *eq_refl*, and the rewriting tactic is realized by case analysis on the proofs of equations. Study the following example to understand.

```
Goal forall (X : Type) (x y : X), x=y -> y=x.
Proof. intros X x y A. destruct A. exact (eq_refl x). Qed.
```

Keep the following facts in mind when stepping through the above proof.

1. By the inductive definition of *eq* only trivial equations $s{=}s$ have proofs.
2. Given a proof of an equation $s{=}t$, we thus know that the terms s and t are equal up to simplification and unfolding of definitions.
3. When you destructure a proof of an equation $s{=}t$ when proving a goal, Coq replaces every occurrence of the term t in the claim and the assumptions of the goal with the term s. This is justified since s and t are equal up to simplification and unfolding of definitions.

Exercise 3.13.1 Prove $\mathit{True} \neq \mathit{False}$.

Exercise 3.13.2 Prove the commutativity of disjunction without using the tactics *left* and *right*.

Exercise 3.13.3 Prove the transitivity of equality without using the tactics *reflexivity* and *rewrite*.

Exercise 3.13.4 Define you 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 3.13.5 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 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.

3.14 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 proof 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 3.8.2 and 3.10.3. 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. now auto.
intros [x y] Z A. apply A ; assumption. Qed.
```

Exercise 3.14.1 Define disjunction with a plain definition based on the impredicative characterization in Exercise 3.8.2. Prove an introduction, an elimination, and an agreement lemma for your disjunction. Carry out the same program for the existential quantifier.

3.15 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 := forall X : Prop, X \/ ~X.
```

Coq can neither prove *XM* nor $\neg XM$. 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 3.15.1 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 forall X Y : Prop,
XM -> ~(X /\ Y) -> ~X \/ ~Y.

Goal forall (X : Type) (p : X -> Prop),
XM -> ~(forall x, p x) -> exists x, ~p x.
```

Exercise 3.15.2 Prove that the following propositions are equivalent. There are short proofs if you use *tauto*.

```
Definition XM : Prop := forall X : Prop, X \/ ~X.          (* excluded middle *)
Definition DN : Prop := forall X : Prop, ~~X -> X.         (* double negation *)
Definition CP : Prop := forall X Y : Prop, (~Y -> ~X) -> X -> Y.   (* contraposition *)
Definition Peirce : Prop := forall X Y : Prop, ((X -> Y) -> X) -> X.   (* Peirce's Law *)
```

Exercise 3.15.3 (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 -> (exists x : X, True) -> exists x, d x -> forall x, d x.
```

Exercise 3.15.4 (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 $XM \to s$ is provable in Coq if and only if $\neg\neg s$ is provable in Coq. This tells us that *tauto* can prove the following goals.

```
Goal forall X : Prop,
~~(X \/ ~X).
```

```
Goal forall X Y : Prop,
~~(((X -> Y) -> X) -> X).

Goal forall X Y : Prop,
~~(~(X /\ Y) <-> ~X \/ ~Y).

Goal forall X Y : Prop,
~~((X -> Y) <-> (~Y -> ~X)).
```

Do the proofs by hand and try to find out why the outer double negation can replace excluded middle.

Exercise 3.15.5 (Decidable Propositions) A proposition s is **decidable** if the proposition $s \vee \neg s$ is provable. Show that the following propositions are decidable.

a) `forall X : Prop, ~(X \/ ~X)`

b) `exists X : Prop, ~(X \/ ~X)`

c) `forall P : Prop, exists f : Prop -> Prop, forall X Y : Prop,`
`(X /\ P -> Y) <-> (X -> f Y)`

d) `forall P : Prop, exists f : Prop -> Prop, forall X Y : Prop,`
`(X -> Y /\ P) <-> (f X -> Y)`

3.16 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-scheme, which identifies the proofs of implications and quantifications as functions. The BHK-scheme 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-scheme (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-scheme 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-scheme, the correctness of the proof rules can be argued. Given the proof rules, the correctness of the BHK-scheme can be argued.

A formal model providing functions as assumed by the BHK-scheme 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.

3.17 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
now t	makes tactic t a solve-or-fail tactic

4 Untyped Lambda Calculus

Type theories are defined as syntactic systems. In this chapter we study a prototypical syntactic system known as untyped lambda calculus. Untyped lambda calculus was invented by Alonzo Church in the 1930's as a model of functional computation. It features lambda abstractions, bound variables, substitution, and beta reduction. Modern type theories can be seen as typed elaborations of untyped lambda calculus.

4.1 Outline

The **terms** of untyped lambda calculus are described by the following grammar where the letter x ranges over symbols called **variables**.

$$s, t ::= x \mid \lambda x.s \mid st$$

Terms of the form $\lambda x.s$ are called **abstractions** and describe functions. The argument variable x is local to the abstraction. Terms of the form st are called **applications** and describe the application of the function described by the term s to the object described by the term t. There are a few notational conventions.

$$\begin{aligned} stu &\rightsquigarrow (st)u \\ \lambda x.st &\rightsquigarrow \lambda x.(st) \\ \lambda xy.s &\rightsquigarrow \lambda x.\lambda y.s \\ \lambda xyz.s &\rightsquigarrow \lambda x.\lambda y.\lambda z.s \end{aligned}$$

Terms of the form $(\lambda x.s)t$ are called **beta redexes**.[1] A beta redex $(\lambda x.s)t$ can be reduced to the term s^x_t. Here is a sequence of **beta reductions**.

$$\begin{aligned} & (\lambda fx.f(fx))\,(\lambda x.x) \\ \to_\beta\ & \lambda x.(\lambda x.x)((\lambda x.x)x) \\ \to_\beta\ & \lambda x.(\lambda x.x)x \\ \to_\beta\ & \lambda x.x \end{aligned}$$

[1] Redex is an artificial word introduced by Church. It stands for "reducible expression".

A term is **normal** if it contains no beta redex. We write $s \to_\beta^* t$ if t can be obtained from the term s by $n \geq 0$ beta reductions. A term t is a **normal form** of a term s if t is normal and $s \to_\beta^* t$. The above series of beta reductions reduces a term to a normal form.

Modern type theories are arranged such that every term has a unique normal form. This is not the case for untyped lambda calculus since there are terms for which beta reduction does not terminate. Here is the canonical example:

$$(\lambda x.xx)\,(\lambda x.xx) \;\to_\beta\; (\lambda x.xx)\,(\lambda x.xx)$$

The term $\lambda x.xx$ is known as ω. Note that we have $\omega\omega \to_\beta \omega\omega$. A term is **diverging** if there is an infinite sequence of beta reductions starting from it. There are diverging terms where the term grows larger with each reduction step.

$$(\lambda x.f(xx))\,(\lambda x.f(xx)) \;\to_\beta\; f\,((\lambda x.f(xx))\,(\lambda x.f(xx)))$$

There are diverging terms that have a normal form.

$$(\lambda xy.y)(\omega\omega) \to_\beta (\lambda xy.y)(\omega\omega) \to_\beta (\lambda xy.y)(\omega\omega) \to_\beta \cdots$$
$$(\lambda xy.y)(\omega\omega) \to_\beta \lambda y.y$$

Untyped lambda calculus has the property that a term has at most one normal form. The uniqueness of normal forms follows from a property known as **confluence**: If we have $s \to_\beta^* s_1$ and $s \to_\beta^* s_2$, then there always exists a term t such that $s_1 \to_\beta^* t$ and $s_2 \to_\beta^* t$.

A variable x is **free** in a term s if it occurs in s as a subterm that is not in the scope of a **binder** λx. For instance, x is free in the term $(\lambda x.x)x$. We say that x has a **bound** and a **free occurrence** in the term $(\lambda x.x)x$. A term t is **open** if there is a variable that is free in t, and **closed** if it is not open.

Exercise 4.1.1 Use beta reduction to derive normal forms of the following terms.

a) $(\lambda xy.fyx)ab$

b) $(\lambda fxy.fyx)(\lambda xy.yx)ab$

c) $(\lambda x.xx)((\lambda xy.y)((\lambda xy.x)ab))$

d) $(\lambda xy.y)((\lambda x.xx)(\lambda x.xx))a$

e) $(\lambda xx.x)yz$

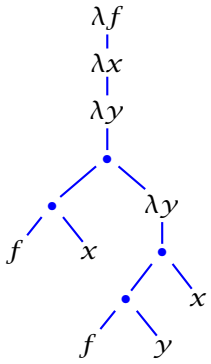
Exercise 4.1.2 Find terms *true*, *false*, and *if* such that *if true* $x\,y \to_\beta^* x$ and *if false* $x\,y \to_\beta^* y$. Hint: Represent *true* and *false* as two-argument functions returning their first and second argument, respectively.

Exercise 4.1.3 Find terms *pair*, *fst*, and *snd* such that *fst* $(pair\ x\ y) \to_\beta^* x$ and *snd* $(pair\ x\ y) \to_\beta^* y$. Hint: Represent a pair (x, y) as the function $\lambda f.fxy$.

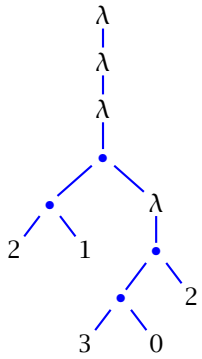
4.2 What Exactly Is A Term?

Modern type theories work with **nameless terms** where the concrete names of bound variables do not matter.[2] For instance, $\lambda xy.yx$ and $\lambda ab.ba$ are seen as two different notations for the same term. Related is the requirement that substitution must be **capture-free**. For instance, $(\lambda x.y)^y_x = \lambda z.x$ but $(\lambda x.y)^y_x \neq \lambda x.x$. What is lacking is a precise definition of terms that distinguishes between notation and the real syntactic object.

Consider the notation $\lambda fxy.fx(\lambda y.fyx)$. From what we have said in the previous section it is clear that this notation parses into the following tree (bullets represent applications).



If we replace the argument variables with numeric backward references, we obtain a faithful representation of the term described by the notation.



An argument reference n says that the λ responsible for this argument can be found as the n+1th λ one encounters on the unique path to the root of the tree.

[2] Church worked with a term representation where the concrete names of bound variables did matter. In such a representation two terms are *alpha equivalent* if they are equal up to renaming of bound variables. In the nameless representation alpha equivalent terms appear as identical terms.

The nameless representation of terms was invented by Nicolaas de Bruijn in the late 1960's in the context of the Automath project, which designed and implemented a type-theoretic language and system for representing and checking mathematical proofs. One speaks of the de Bruijn representation of terms. Argument references are sometimes called de Bruijn indices.

Exercise 4.2.1 Draw the nameless tree representations of the following terms.

a) $\lambda xyz.x$

b) $\lambda xyz.z(\lambda x.xz)$

4.3 Formalization of Terms and Substitution

We now see that the real grammar for terms is

$$s,t ::= n \mid \lambda s \mid st$$

where n ranges over natural numbers. Based on this grammar we formalize terms in Coq with the following inductive definition.

```
Inductive ter : Type :=
| R : nat -> ter
| L : ter -> ter
| A : ter -> ter -> ter.
```

In Coq's notation the term $\lambda fx.fxx$ can now be written as

```
L (L (A (A (R 1) (R 0)) (R 0)))
```

Now that we have a rigorous definition of terms, our next goal is the precise definition of substitution and beta reduction. In the fancy notation beta reduction and substitution look as follows:

$$(\lambda x.s)t \rightarrow_\beta^* s_t^x$$

We aim at a function $subst : nat \to ter \to ter \to ter$ such that

$$A\ (L\ s)\ t \rightarrow_\beta subst\ 0\ s\ t$$

Finding such a function is a rather interesting puzzle to solve. We note the following.

1. $subst\ d\ s\ t$ must replace the reference d in s with the term t.
2. $subst\ d\ s\ t$ must decrement all references $n > d$ in s by 1. This is necessary since a reference $n > d$ is free in the original term $A\ (L\ s)\ t$ and beta reduction removes the lambda above s.

3. Whenever *subst* descends into the body of an abstraction, the reference d to be replaced must be incremented by 1. Since *subst* starts with $d = 0$, the value of d says how many lambda nodes are above the current subterm.
4. When *subst* replaces the reference d with the term t, all free references in t must be incremented by d since they must skip the lambdas *subst* has descended through. This operation is called **shifting**.

For *subst* and also for shifting we need a comparison operation for numbers.

```
Inductive order : Type :=
| Less : order
| Equal : order
| Greater : order.

Fixpoint order_nat (m n : nat) : order :=
match m,n with
| O, O => Equal
| O, S _ => Less
| S _, O => Greater
| S m, S n => order_nat m n
end.
```

We first write the shifting operation $subst : nat \to nat \to ter \to ter$. An application *shift d k s* yields the term obtained from s by incrementing every free reference $n \geq d$ in s by k.

```
Fixpoint shift (d k : nat) (s : ter) :=
match s with
| A s1 s2 => A (shift d k s1) (shift d k s2)
| L s' => L (shift (S d) k s')
| R n => match order_nat n d with
           | Less => R n
           | _ => R (n + k)
         end
end.
```

Note that a reference n is free in the original term if and only if $n \geq d$. We can now write the substitution operation.

```
Fixpoint subst (d : nat) (s t : ter) :=
match s with
| A s1 s2 => A (subst d s1 t) (subst d s2 t)
| L s' => L (subst (S d) s' t)
| R n => match order_nat n d with
           | Less => R n
           | Equal => shift 0 d t
           | Greater => R (pred n)
         end
end.
```

For the beta reduction

$$(\lambda xy.xy)\,(\lambda x.x) \;\to_\beta\; \lambda y.(\lambda x.x)y$$

we obtain the following term.

```
Compute subst 0 (L (A (R 1) (R 0))) (L (R 0)).
% L(A (L (R 0)) (R 0))
```

Exercise 4.3.1 Write a function $free : nat \to nat \to ter \to bool$ such that $free\ 0\ n\ t$ yields $true$ if and only if the reference n is free in t.

4.4 Formalization of Beta Reduction

We now formalize beta reduction. In mathematical notation we may define beta reduction by the following rules.

$$\frac{}{(\lambda x.s)t \;\to_\beta\; s^x_t} \qquad \frac{s \;\to_\beta\; s'}{\lambda x.s \;\to_\beta\; \lambda x.s'} \qquad \frac{s \;\to_\beta\; s'}{st \;\to_\beta\; s't} \qquad \frac{t \;\to_\beta\; t'}{st \;\to_\beta\; st'}$$

The rules with premises make it possible to descent to beta redexes appearing as subterms. In Coq, we formalize the mathematical definition with a test $beta : ter \to ter \to bool$ such that $beta\ s\ t$ yields $true$ if and only if $s \to_\beta t$. We start with equality tests for numbers and terms.

```
Definition eq_nat (m n : nat) : bool :=
match order_nat m n with Equal => true | _ => false end.

Fixpoint eq_ter (s t : ter) : bool :=
match s, t with
| R m, R n => eq_nat m n
| A s1 s2, A t1 t2 => andb (eq_ter s1 t1) (eq_ter s2 t2)
| L s', L t' => eq_ter s' t'
| _, _ => false
end.

Definition beta_top (s u : ter) : bool :=
match s with
| A (L s) t => eq_ter (subst 0 s t) u
| _ => false
end.

Fixpoint beta (s t : ter) : bool :=
orb (beta_top s t)
match s, t with
| R _ , _ => false
```

```
| L s', L t' => beta s' t'
| A s1 s2, A t1 t2 => orb (andb (beta s1 t1) (eq_ter s2 t2))
                          (andb (beta s2 t2) (eq_ter s1 t1))
| _, _ => false
end.
```

We can now prove that $\omega\omega$ beta reduces to itself.

```
Definition omega : ter := L (A (R 0) (R 0)).

Goal beta (A omega omega) (A omega omega) = true.

Proof. reflexivity. Qed.
```

Our next goal is the definition of a normal form predicate. We first write a test that checks whether a term is normal.

```
Fixpoint normal (s : ter) : bool :=
match s with
| R _ => true
| L s' => normal s'
| A (L _) _ => false
| A s1 s2 => andb (normal s1) (normal s2)
end.
```

Next we need a formalization of the reduction relation $s \to_\beta^* t$. Since this relation is known to be computationally undecidable, we cannot work with a boolean test here. Instead we define an inductive predicate $red : ter \to ter \to Prop$ such that $red\ s\ t$ is provable if and only if $s \to_\beta^* t$.

```
Inductive red : ter -> ter -> Prop :=
| redR : forall s, red s s
| redS : forall t s u, beta s t = true -> red t u -> red s u.
```

The proof constructor *redR* gives us proofs for all propositions $red\ s\ s$. The proof constructor *redS* gives us proofs for all propositions $red\ s\ u$ such that there is some term t such that $beta\ s\ t = true$ and $red\ t\ u$ are provable. In mathematical notation the inductive definition of *red* may look as follows.

$$\frac{}{s \to_\beta^* s} \qquad \frac{s \to_\beta t \qquad t \to_\beta^* u}{s \to_\beta^* u}$$

```
Lemma beta_red s t :
beta s t = true -> red s t.

Proof. intros A. apply (@redS t). exact A. apply redR. Qed.
```

The definition of the normal form predicate is now routine.

```
Definition normal_form (s t : ter) : Prop :=
normal t = true /\ red s t.
```

Consider the following derivation of a normal form where the free variable x is represented as the reference 7.

$$\begin{array}{ll} & (\lambda f.f(fx))\,(\lambda y.y) \qquad\qquad (\lambda(0(0\ 8)))\,(\lambda 0) \\ \to_\beta & (\lambda y.y)\,((\lambda y.y)\,x) \qquad\qquad (\lambda 0)\,((\lambda 0)7) \\ \to_\beta & (\lambda y.y)\,x \qquad\qquad (\lambda 0)7 \\ \to_\beta & x \qquad\qquad 7 \end{array}$$

In Coq, we can verify the derivation as follows.

```
Goal normal_form
(A (L (A (R 0) (A (R 0) (R 8)))) (L (R 0)))
(R 7).

Proof. split. reflexivity.
apply (@redS (A (L (R 0)) (A (L (R 0)) (R 7)))). reflexivity.
apply (@redS (A (L (R 0)) (R 7))). reflexivity.
apply beta_red. reflexivity. Qed.
```

Exercise 4.4.1 For each of the following terms s find a normal form t and prove *normal_form s t* in Coq. Use the references 0 and 1 for the free variables.

a) $(\lambda f.fxy)(\lambda xy.x)$

b) $(\lambda fxy.fyx)(\lambda xy.yx)ab$

4.5 Church-Girard Programming

We will now see an extreme form of functional programming where numbers are represented as functions. The idea is due to Church who came up with the following functional representations of booleans and numbers in the untyped lambda calculus.

$$\begin{array}{rcl} \mathit{false} & := & \lambda xy.\,y \\ \mathit{true} & := & \lambda xy.\,x \\ 0 & := & \lambda xf.\,x \\ 1 & := & \lambda xf.\,fx \\ 2 & := & \lambda xf.\,f(fx) \\ 3 & := & \lambda xf.\,f(f(fx)) \end{array}$$

It turns out that all computable functions on natural numbers can be expressed as closed terms in the untyped lambda calculus. The terms coding the numbers are called **Church numerals**.

Girard showed that Church's codings carry over to constructive type theory.[3] The boolean values and the numbers now appear as values of function types in the universe *Prop*.[4]

```
Definition Bool : Prop := forall X : Prop, X -> X -> X.
Definition true : Bool := fun X x y => x.
Definition false : Bool := fun X x y => y.

Definition Nat : Prop := forall X : Prop, X -> (X -> X) -> X.
Definition O : Nat := fun X x f => x.
Definition one : Nat := fun X x f => f x.
Definition two : Nat := fun X x f => f (f x).
```

Note that a number n is represented as a function that applies a given function n-times to a given value. It is easy to write a successor function and an addition function.

```
Definition S (n : Nat) : Nat := fun X x f => n X (f x) f.

Definition plus (m n : Nat) : Nat := m Nat n S.

Compute plus two (S two).
% fun X x f => f (f (f (f (f x))))
```

A multiplication function is also not difficult.

```
Definition times (m n : Nat) : Nat := m Nat O (plus n).
```

Exercise 4.5.1 Write a function $power : Nat \to Nat \to Nat$ that for two numbers m and n yields the power m^n (numbers represented as Church numerals).

Exercise 4.5.2 The following type represents pairs as functions:

```
Definition Prod (X Y : Prop) : Prop :=
forall Z : Prop, X -> Y -> Z.
```

Write functions *pair*, *fst*, and *snd* that construct and decompose pairs.

Exercise 4.5.3 Write a function $fac : Nat \to Nat$ that computes the factorial $n!$ of a number n (numbers represented as Church numerals). Hint: Iterate on pairs $(n, n!)$ starting with $(0, 0!)$.

Exercise 4.5.4 Write a function $pred : Nat \to Nat$ that computes the predecessor of a number (numbers represented as Church numerals). Hint: Iterate on pairs $(n, n-1)$ starting with $(0, 0)$.

[3] In fact, Girard constructed the polymorphic lambda calculus, the first type theory that can express Church's codings.

[4] There is a reason that we work in the universe *Prop* and not in the universe *Type*. We will explain this issue later.

5 Basic Dependent Type Theory

A type theory is a syntactic system that features functions and types. Functions are described with typed lambda abstractions carrying a term describing the type of the argument. There is a typing relation distinguishing well-typed terms from ill-typed terms. Things are arranged such that beta reduction always terminates.

We look at a subsystem of the type theory underlying Coq. This system come with dependent function types and accommodate types as first-class objects. We first study a basic system and then add a universe for propositions.

5.1 Terms

The terms of our basic type theory provide for functions, dependent function types, and universes. Universes serve as types of function types and universes. Terms are obtained with the grammar

$$s, t \ ::= \ x \ \mid \ \lambda x{:}s.t \ \mid \ s\,t \ \mid \ \forall x{:}s.t \ \mid \ U$$

where x ranges over **variables** and U ranges over **universes**. Syntactically, universes can be seen as atomic symbols. Bound variables are introduced by **lambda abstractions** (terms of the form $\lambda x{:}s.t$) and **function types** (terms of the form $\forall x{:}s.t$). Both constructs carry a term s describing the type of the argument. Terms of the form st are called **applications**. We have already seen the Coq notations for lambda abstractions and function types:

$$\begin{aligned} \lambda x{:}s.t &\rightsquigarrow \mathit{fun}\ x{:}s \Rightarrow t \\ \forall x{:}s.t &\rightsquigarrow \mathit{forall}\ x{:}s,\ t \end{aligned}$$

We adopt the following notations.

$$\begin{aligned} s\,t\,u &\rightsquigarrow (s\,t)u && \\ s \to t &\rightsquigarrow \forall x{:}s.t && \text{provided } x \text{ does not occur in } t \\ s \to t \to u &\rightsquigarrow s \to (t \to u) && \\ \lambda xy{:}s.t &\rightsquigarrow \lambda x{:}s.\lambda y{:}s.t && \text{analogous for 3 and more variables} \\ \forall xy{:}s.t &\rightsquigarrow \forall x{:}s.\forall y{:}s.t && \text{analogous for 3 and more variables} \end{aligned}$$

Note that the familiar **arrow types** $s \to t$ are function types where the result type does not depend on the actual argument.

```
Check forall X : Type, Type.
% Type → Type : Type

Check forall X Y : Type, Type.
% Type → Type → Type : Type

Check forall X Y : Type, Y.
% Type → forall Y : Type : Y
```

A function type $\forall x{:}s.t$ is called **dependent** if x is free in t. A dependent function type of the form $\forall x{:}U.t$ is called **polymorphic**. Two straightforward examples of polymorphic function types are $\forall x{:}U.x$ and $\forall x{:}U.x \to x$. For more practical examples consider the types of the pair constructor *pair* and the projection functions fst and snd in Section 2.5.

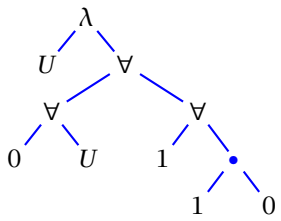
The formalization of terms in Coq follows the ideas we have seen for the untyped lambda calculus.

```
Inductive ter : Type :=
| R : nat -> ter
| L : ter -> ter -> ter
| A : ter -> ter -> ter
| F : ter -> ter -> ter
| U : uni -> ter.
```

The definition assumes that the universes appear as members of some type *uni*. Arguments of function types and lambda abstraction are referred to by numeric references. The argument x of a lambda abstraction $\lambda x{:}s.t$ or a function type $\forall x{:}s.t$ is invisible in the argument type s but visible in the body t. Thus the nameless tree representation of the term

$$\lambda X{:}U.\forall f{:}X{\to}U.\forall x{:}X.fx$$

looks as follows.



A formal account of the binding structure of terms can be given with a free reference test.

```
Fixpoint free (d n : nat) (s : ter) : bool :=
match s with
| R k => eq_nat k (d + n)
| L s1 s2 => orb (free d n s1) (free (S d) n s2)
| A s1 s2 => orb (free d n s1) (free d n s2)
| F s1 s2 => orb (free d n s1) (free (S d) n s2)
| U _ => false
end.
```

The test is defined such that *free d n s* yields *true* if and only if n is a free reference in the term s. Note that *free* increments the binder depth d for the right constituents (the bodies) of lambda abstractions and function types but not for the left constituents (the argument types).

Exercise 5.1.1 Draw the nameless tree representation of the following term. $\lambda xyz{:}U.\, x \to (\forall u{:}x.\, z \to y)$

Exercise 5.1.2 Decide for each pair of term notations whether the two terms are identical.

a) $\forall x{:}U.x \to x$ and $\forall y{:}U.y \to y$

b) $\lambda xy{:}U.x \to y \to x$ and $\lambda yx{:}U.y \to x \to y$

c) $\lambda xyz{:}U.x \to (\forall u{:}x.z \to y)$ and $\lambda yxz{:}U.y \to (\forall u{:}x.z \to x)$

d) $\lambda x{:}U.x$ and $\forall x{:}U.x$

e) $(\lambda xy{:}U.y)x$ and $(\lambda x{:}U.\lambda z{:}U.z)x$

Exercise 5.1.3 Determine the free variables of the following terms.

a) $\lambda x : y.\lambda z{:}x.\, zx$

b) $\lambda x : y.\forall z{:}x.\, x \to x'$

Exercise 5.1.4 Write a substitution function $subst : nat \to ter \to ter \to ter$. Follow the definition of the substitution function for untyped lambda calculus (Section 4.3) and the definition of the free variable test above.

5.2 Reduction

Beta reduction is defined as in the untyped lambda calculus

$$\begin{aligned}(\lambda x{:}\,u.s)t &\to_\beta s_t^x \\ (\lambda u.s)t &\to_\beta subst\ 0\ s\ t\end{aligned}$$

where a beta redex can be replaced in any subterm position.

Later we will add constructs to the type theory that require additional reduction rules. In anticipation of such extensions we will write $s \rhd_1 t$ for a one-step reduction and $s \rhd t$ for a $n \geq 0$ step reduction. For the basic type theory we are considering now $\rhd_1$ is exactly $\to_\beta$ and $\rhd$ is exactly $\to_\beta^*$.

A term s is **normal** or **irreducible** if there is no term t such that $s \rhd t$. A term t is a **normal form** of a term s if $s \rhd t$ and t is normal. A term s **terminates** if there is no infinite reduction sequence $s \rhd_1 s_1 \rhd_1 s_2 \rhd_1 \cdots$. Note that every terminating term has a normal form.

A basic design principle for type theories is that reduction must be **confluent**: If we have $s \rhd s_1$ and $s \rhd s_2$, then there always exists a term t such that $s_1 \rhd t$ and $s_2 \rhd t$. All the type theories we will consider are confluent. If reduction is confluent, then a term has at most one normal form.

One says that two terms s and t are **convertible** if $s \rhd u$ and $t \rhd u$ for some term u. We write $s \approx t$ to say that s and t are convertible. It follows from the confluence of reduction that convertibility is an equivalence relation.

Exercise 5.2.1 Determine the normal forms of the following terms.

a) $(\lambda x{:}U.\lambda g{:}U \to U \to U.\ (\lambda f{:}U \to U.\forall x{:}U.fx)(gx))\,U$

b) $\lambda x{:}U.(\lambda f{:}x \to x \to x.\lambda yz{:}x.f(fyz)(fzy))(\lambda yz{:}x.z)$

Exercise 5.2.2 Explain why terms have at most one normal form.

Exercise 5.2.3 Explain why convertibility of terms is an equivalence relation.

Exercise 5.2.4 Give a term of the untyped lambda calculus that has a normal form but does not terminate.

5.3 Typing

A type theory can be defined in three steps:

1. Define the terms of the theory.
2. Define a confluent reduction relation on terms.
3. Define a **typing relation** "$\Gamma \vdash s : t$".

The typing relation relates a context Γ with two terms s and t. A statement "$\Gamma \vdash s : t$" says that s has type t in Γ. A **context** is a sequence of typing assumptions $x : s$ for variables. Contexts are defined with the grammar

$$\Gamma \ ::= \ \emptyset \ \mid \ \Gamma, x : s$$

where $\emptyset$ is called the **empty context**. We impose the side condition that a context contains at most one assumption per variable.[1]

Given a typing relation, we can make some basic definitions. If $\Gamma \vdash s : t$, we say that s **has type** t **in** Γ, and that s **is a member of** t **in** Γ. Let Γ be a context. A term s is **well-typed in** Γ if there exists t such that $\Gamma \vdash s : t$. A term s is a **type in** Γ if there exists a universe U such that $\Gamma \vdash s : U$. We say that a term is **well-typed** if it is well-typed in some context, and that a term is a **type** if it is a type in some context. We say that a type t is **inhabited** if there is term s such that $\emptyset \vdash s : t$. We say that a type t in $\emptyset$ is **empty** if there is no term s such that $\emptyset \vdash s : t$.

We now list important properties one requires for typing relations in general. We will define several concrete typing relations satisfying these properties.

Propagation

1. If $\Gamma \vdash s : t$, then there exists a universe U such that $\Gamma \vdash t : U$.
2. If $\Gamma, x : u \vdash s : t$, then there exists a universe U such that $\Gamma \vdash u : U$.

Slogan: Only types can appear on the right.

Preservation

If $\Gamma \vdash s : t$ and $s \rhd s'$, then $\Gamma \vdash s' : t$.
Slogan: Typing respects reduction on the left.

Convertibility

If $\Gamma \vdash s : t$ and $t \approx t'$ and $\Gamma \vdash t' : U$, then $\Gamma \vdash s : t'$.
Slogan: Typing respects conversion on the right.

Strong Normalization

If $\Gamma \vdash s : t$, then s terminates.
Slogan: Reduction of well-typed terms terminates.

Decidability

There is an algorithm that given Γ and s decides whether there is a term t such that $\Gamma \vdash s : t$.
Slogan: Type checking is decidable.

Consistency

For every universe U there is an empty type t in U. That is, for every universe U there is a term t such that $\emptyset \vdash t : U$ and there is no s such that $\emptyset \vdash s : t$.
Slogan: All universes have an empty type.

Exercise 5.3.1 Explain the following.

a) If $\Gamma \vdash s : t$, then t terminates.

[1] In a formal account, numeric references would take the place of variables, and contexts would be represented as stacks of terms where a reference indicates a position in the stack.

b) If $\Gamma, x : u, \Gamma' \vdash s : t$, then u terminates.

Exercise 5.3.2 (Oracle Functions) Given a universe U, the members of the type $\forall x : U.x$ are functions that yield a member for every type in U. We call such functions *oracle functions*. Explain why a consistent type theory does not admit oracle functions (i.e., no type $\forall x : U.x$ is inhabited).

5.4 Basic Dependent Type Theory Without Prop

We now return to our concrete type theory. Before we define the typing relation, we first fix **infinitely many universes** $\mathsf{T}_0, \mathsf{T}_1, \mathsf{T}_2, \ldots$. The typing relation will be defined such that we obtain the hierarchy $\mathsf{T}_0 : \mathsf{T}_1 : \mathsf{T}_2 : \cdots$. The infinite hierarchy is needed since every universe needs to be a member of some universe and a cycle $U : \cdots : U$ would result in an inconsistent type theory.

The typing relation is defined with a proof system that derives syntactic objects called **judgements**. Judgements take the form $\Gamma \Rightarrow s : t$. The statement $\Gamma \vdash s : t$ now means that the judgement $\Gamma \Rightarrow s : t$ can be derived with the proof system.

Figure 5.1 shows the rules of the proof system for our basic dependent type theory. We refer to the rules as basic typing rules. The first two rules derive typings for universes and variables. The **weakening rule** Weak makes it possible to add further assumptions to the context. Keep in mind that there is the tacit assumption that a context contains at most one assumption per variable. The rules Lam, App, and Fun derive typings for lambda abstractions, applications, and function types. Note that the rule for function types ensures that every universe is closed under taking function types. The **conversion rule** Conv establishes convertibility of types. The **subsumption rule** makes the universe hierarchy cumulative: $\mathsf{T}_0 \subseteq \mathsf{T}_1 \subseteq \mathsf{T}_2 \subseteq \cdots$. The cumulativity extends to function types that target universes.

Basic dependent type theory is fully implemented in Coq. The lowest universe T_0 is written *Set* in Coq. The remaining universes $\mathsf{T}_1, \mathsf{T}_2, \mathsf{T}_3, \ldots$ are all summarized under the keyword *Type*. Behind the curtain Coq assigns levels to the occurrences of *Type* and makes sure that no cycle $\mathsf{T}_k : \cdots : \mathsf{T}_k$ occurs. Which levels are exactly assigned does not matter.

The definition of the typing relation is such that basic dependent type theory satisfies all the requirements we have listed for the typing relation in Section 5.3. Furthermore, the following canonical form property holds.[2]

Canonical Form

If $\emptyset \vdash s : t$ and s and t are both normal, then one of the following holds:

[2] In a type theory with inductive types the canonical form property needs to be adapted.

$$\textbf{Uni}\ \frac{}{\emptyset \Rightarrow \mathsf{T}_n : \mathsf{T}_{n+1}} \qquad \textbf{Var}\ \frac{\Gamma \Rightarrow s : U}{\Gamma, x : s \Rightarrow x : s} \qquad \textbf{Weak}\ \frac{\Gamma \Rightarrow s : t \qquad \Gamma \Rightarrow u : U}{\Gamma, x : u \Rightarrow s : t}$$

$$\textbf{Lam}\ \frac{\Gamma, x : s \Rightarrow t : u}{\Gamma \Rightarrow \lambda x : s.t : \forall x : s.u} \qquad \textbf{App}\ \frac{\Gamma \Rightarrow s : \forall x : u.v \qquad \Gamma \Rightarrow t : u}{\Gamma \Rightarrow s\,t : v_t^x}$$

$$\textbf{Fun}\ \frac{\Gamma \Rightarrow s : U \qquad \Gamma, x : s \Rightarrow t : U}{\Gamma \Rightarrow \forall x : s.t : U}$$

$$\textbf{Conv}\ \frac{\Gamma \Rightarrow s : t \qquad \Gamma \Rightarrow t' : U}{\Gamma \Rightarrow s : t'} \quad t \approx t'$$

$$\textbf{Sub}\ \frac{\Gamma \Rightarrow s : \forall x_1 : t_1 \cdots \forall x_k : t_k.\mathsf{T}_m}{\Gamma \Rightarrow s : \forall x_1 : t_1 \cdots \forall x_k : t_k.\mathsf{T}_n} \quad k \geq 0 \text{ and } m < n$$

Figure 5.1: Basic typing rules

1. s and t are both universes and s is lower than t.
2. s is a function type and t is a universe.
3. s is a lambda abstraction and t is a function type.

Exercise 5.4.1 Derive the following judgements.

a) $\emptyset \Rightarrow \mathsf{T}_3 : \mathsf{T}_5$

b) $\emptyset \Rightarrow \lambda x : \mathsf{T}_0.\, x : \forall x : \mathsf{T}_0.\mathsf{T}_0$

c) $\emptyset \Rightarrow \lambda x : \mathsf{T}_0.\, x : \mathsf{T}_0 \to \mathsf{T}_0$

d) $x : \mathsf{T}_0 \Rightarrow (\lambda x : \mathsf{T}_0.\, x)x : \mathsf{T}_0$

e) $\emptyset \Rightarrow \mathsf{T}_3 : (\lambda x : \mathsf{T}_6.x)\mathsf{T}_5$

Exercise 5.4.2 Make sure that you can reconstruct and explain the typing rules Lam, App, and Fun.

Exercise 5.4.3 Find terms s, s', t, t' such that $\emptyset \vdash s : t$, $\emptyset \vdash s' : t'$, $s \rhd s'$ and $\emptyset \not\vdash s : t'$.

Exercise 5.4.4 Answer the following questions and explain your answers.

a) Is there a nonterminating term that reduces to a terminating term?
b) Is there an ill-typed term that reduces to a well-typed term?
c) Is there a well-typed term that reduces to an ill-typed term?
d) Suppose $\Gamma \vdash s : t$ and $t \triangleright u$. Does this imply $\Gamma \vdash s : u$?
e) Is $\emptyset \Rightarrow \mathsf{T}_1 : \mathsf{T}_0$ derivable?
f) Is $\forall x : \mathsf{T}_0. \forall y : x. y$ well-typed?

Exercise 5.4.5 Explain the following type error.

```
Definition T : Type := forall X : Type, X -> X.
Check fun f : T => f T.
% Error : Universe inconsistency.
```

5.5 Adding Propositions

We now add an additional universe *Prop* to our basic type theory satisfying a stronger closure property than the ordinary universes. This property is established with the additional typing rule

$$\mathbf{Fun}_{Prop} \quad \frac{\Gamma, x : s \Rightarrow t : Prop}{\Gamma \Rightarrow \forall x : s. t : Prop}$$

Speaking logically, the rule Fun_{Prop} closes *Prop* under quantification over all types.[3] One says that the universe *Prop* is **impredicative**, and that the ordinary universes T_n are **predicative**. The impredicative characterizations of the logical operations (Exercise 3.8.2) depend on the impredicativity of *Prop*. Church-Girard programming (Section 4.5) also exploits the impredicativity of *Prop*.

There are two additional typing rules for *Prop* making *Prop* a member of T_1 and a subtype of T_0.

$$\mathbf{Uni}_{Prop} \quad \frac{}{\emptyset \Rightarrow Prop : \mathsf{T}_1} \qquad \mathbf{Sub}_{Prop} \quad \frac{\Gamma \Rightarrow s : \forall x_1 : t_1 \cdots \forall x_k : t_k. Prop}{\Gamma \Rightarrow s : \forall x_1 : t_1 \cdots \forall x_k : t_k. \mathsf{T}_0} \quad k \geq 0$$

The rule Sub_{Prop} makes *Prop* a subuniverse of all other universes. In fact, we have $Prop \subseteq \mathsf{T}_0 \subseteq \mathsf{T}_1 \subseteq \mathsf{T}_2 \subseteq \cdots$. Things are arranged such that *Prop* is not a member of T_0. The canonical form property for the basic dependent type theory carries over unchanged if we consider *Prop* a lower universe than T_1.

For easy reference the typing rules for *Prop* are summarized in Figure 5.2.

[3] An ordinary universe T_n is not even closed under quantification over T_n. For instance, $\forall x : \mathsf{T}_n. x$ is not a member of T_n. That is, the judgement $\emptyset \Rightarrow \forall x : \mathsf{T}_n. x : \mathsf{T}_n$ is not derivable.

$$\textbf{Uni}_{Prop}\ \frac{}{\emptyset \Rightarrow Prop : \mathsf{T}_1} \qquad \textbf{Sub}_{Prop}\ \frac{\Gamma \Rightarrow s : \forall x_1 : t_1 \cdots \forall x_k : t_k . Prop}{\Gamma \Rightarrow s : \forall x_1 : t_1 \cdots \forall x_k : t_k . \mathsf{T}_0}\ k \geq 0$$

$$\textbf{Fun}_{Prop}\ \frac{\Gamma, x : s \Rightarrow t : Prop}{\Gamma \Rightarrow \forall x : s . t : Prop}$$

Figure 5.2: Typing rules for *Prop*

$$\begin{array}{rcl}
TRUE & := & \forall X : Prop.\ X \to X \\
FALSE & := & \forall X : Prop.\ X \\
NOT & := & \lambda X : Prop.\ X \to FALSE \\
AND & := & \lambda XY : Prop.\ \forall Z : Prop.\ (X \to Y \to Z) \to Z \\
OR & := & \lambda XY : Prop.\ \forall Z : Prop.\ (X \to Z) \to (Y \to Z) \to Z \\
EX_n & := & \lambda X : \mathsf{T}_n . \lambda p : X \to Prop.\ \forall Z : Prop.\ (\forall x : X . p\, x \to Z) \to Z \\
EQ_n & := & \lambda X : \mathsf{T}_n . \lambda xy : X.\ \forall p : X \to Prop.\ p\, x \to p\, y
\end{array}$$

Figure 5.3: Impredicative definitions of logical operations

```
Definition TRUE : Prop := forall X : Prop, X -> X.

Definition FALSE : Prop := forall X : Prop, X.

Definition NOT (X : Prop) : Prop := X -> FALSE.

Definition AND (X Y : Prop) : Prop :=
forall Z : Prop, (X -> Y -> Z) -> Z.

Definition OR (X Y : Prop) : Prop :=
forall Z : Prop, (X -> Z) -> (Y -> Z) -> Z.

Definition EX (X : Type) (p : X -> Prop) : Prop :=
forall Z :Prop, (forall x:X, p x -> Z) -> Z.

Definition EQ (X : Type) (x y : X) : Prop :=
forall p : X -> Prop, p x -> p y.
```

Figure 5.4: Impredicative definitions of logical operations in Coq

A **proposition** is a term t such that $\emptyset \vdash t : \mathit{Prop}$. A **proof term** for a proposition t is a term s such that $\emptyset \vdash s : t$. We now have a type theory with propositions where implication and universal quantification are native logical operations. The remaining logical operations can be expressed using the impredicative characterizations we have seen in Exercises 3.8.2 and 3.10.3. We will use the notational definitions shown in Figure 5.3 and speak of the **impredicative definitions**. Note that we define equality and existential quantification for every universe T_n. Figure 5.4 realizes the impredicative definitions in Coq. Note that the universe levels of *EQ* and *EX* are handled automatically by Coq.

We have now arrived at a type theory in which we can prove propositions with proof terms and where proof checking is obtained as type checking. Coq implements an extension of this theory. We can use Coq to check whether a term is a proof term for a proposition.

```
Goal forall X Y : Prop, X -> Y -> AND X Y.

Proof. exact (fun X Y x y Z f => f x y).
Show Proof. Qed.
```

Note that we leave it to Coq to derive the argument types of the proof term. The full proof term will be shown by the command *Show Proof*.[4] Here are further examples.

```
Goal forall X Y : Prop, AND X Y -> Y.

Proof. exact (fun X Y A => A Y (fun _ y => y)).
Show Proof. Qed.

Goal forall X : Prop, (X -> NOT X) -> (NOT X -> X) -> FALSE.

Proof. exact (fun X A B => (fun x => A x x) (B (fun x => A x x))).
Show Proof. Qed.

Goal NOT (EQ TRUE FALSE).

Proof. intros A. apply (A (fun z => z)). exact (fun X x => x).
Show Proof. Qed.
```

Exercise 5.5.1 Check that the following judgements are derivable.

a) $\emptyset \Rightarrow \mathit{TRUE} : \mathit{Prop}$

b) $\emptyset \Rightarrow \mathit{FALSE} : \mathit{Prop}$

c) $\emptyset \Rightarrow \mathit{NOT} : \mathit{Prop} \to \mathit{Prop}$

d) $\emptyset \Rightarrow \mathit{AND} : \mathit{Prop} \to \mathit{Prop} \to \mathit{Prop}$

[4] The command *Show Proof* can be used in proof scripts before the full proof term is constructed. It then displays the partial proof term constructed so far and connects it to the open subgoals.

e) $\emptyset \Rightarrow OR : Prop \to Prop \to Prop$

f) $\emptyset \Rightarrow EQ_n : \forall X : \mathsf{T}_n . X \to X \to Prop$

g) $\emptyset \Rightarrow EX_n : \forall X : \mathsf{T}_n . (X \to Prop) \to Prop$

Exercise 5.5.2 Let X and Y be variables.

a) Verify that $EX_n\, X\, (\lambda x : X . Y)$ and $AND\, X\, Y$ are convertible.

b) Is there a term t such that $X : Prop \vdash EX_n\, X\, : t$?

c) Is there a term t such that $X : Prop, Y : Prop \vdash EX_n\, X\, (\lambda x : X . Y)\, : t$?

Exercise 5.5.3 Find proof terms for the following propositions. Check your findings with the tactic *exact*.

(a) TRUE

(b) NOT FALSE

(c) OR TRUE FALSE

(d) forall X Y : Prop, X -> OR X Y

(e) forall (X : Type) (x : X), EQ x x

(f) forall (X : Type) (x y : X), EQ x y -> EQ y x

(g) forall (X : Type) (p : X -> Prop) (x : X), p x -> EX p

Exercise 5.5.4 Rewrite the Church-Girard definitions in Section 4.5 such that *Type* is used instead of *Prop*. Coq will then reject the definition of *plus* with the error message "*universe inconsistency*" since there is no consistent level assignment. Find out why this is the case.

Exercise 5.5.5 Argue that the type theory is consistent if and only if the type *FALSE* is empty. Recall Exercise 5.3.2.

5.6 Remarks

We have now seen the full development of a basic dependent type theory with an impredicative universe for propositions. Coq implements an extension of this theory. The theory is such that we can represent propositions and proofs and that type checking acts as proof checking.

It is perfectly possible to make T_0 impredicative and omit the extra universe *Prop*. It is for historical reasons that Coq does not realize this simpler solution. It is however impossible to make any universe higher than T_0 impredicative since this results in an inconsistent theory. Inconsistency also results if one works with a single universe that is a member of itself.

The impredicative universe Prop has considerable computational power, as can be seen from the fact that it can accommodate Church-Girard programming (see Section 4.5). As it comes to proving, the Church-Girard coding of numbers is however inadequate since we cannot even prove basic facts such as $0 \neq 1$. Thus Coq represents the natural numbers with an inductive type.

Coq represents all proofs as proof terms. Coq's tactics provide a convenient means to construct proof terms interactively. The (partial) proof term constructed with a proof script can be displayed with the command *Show Proof*. The command *Qed* type checks the proof term that has beed constructed by the preceding script. There are some cases where the tactics fail to construct a fully well-typed proof term. In this case *Qed* will fail.

A command "*Lemma* $x : t$" announces the attempt to construct a term of type t. If such a term s is successfully constructed, the lemma command together with the proof script has the same effect as the command "*Definition* $x : t := s$", except that Coq will not unfold the name x. The exact behavior of the definition "*Definition* $x : t := s$" can be obtained by ending the proof script with the command *Defined* rather than *Qed*. The command "*Lemma* $x : t$" can be used for any type t, not just propositions.

The type theory presented in this chapter is known as calculus of constructions. The original version of the calculus of construction was given in 1985 by Coquand and Huet as the foundation of the initial Coq system. We recommend the Luo's [1994] book in case you want to know more about the calculus of constructions.

6 Adding Bool and Nat

We now extend the basic dependent type theory of the last chapter with inductive types *bool* and *nat* for booleans and natural numbers. For each of the two types we add constructors and a match providing for case analysis. For *nat* we also add a fix providing for recursion. The typing rules for *nat* are complemented by a guardedness condition ensuring termination. To provide for proof terms for case analysis and induction, the theory comes with dependent matches generalizing the simple matches we have seen so far.

6.1 Dependent Matches

Consider the following proof of the case analysis principle for booleans.

```
Goal forall p : bool -> Prop,
p true -> p false -> forall x : bool, p x.

Proof. intros p A B x. destruct x. exact A. exact B. Show Proof. Qed.
```

The proof term displayed involves a dependent match with return annotation.[1] We prove the principle once more by stating the proof term explicitly.

```
Goal forall p : bool -> Prop, p true -> p false -> forall x : bool, p x.

Proof. exact (fun p A B x => match x as z return p z with
                             | true => A
                             | false => B
                             end).
Qed.
```

The *as* and *return* annotations determine the **return function** $\lambda z : bool.\, pz$ of the dependent match. A boolean match for a term s with a return function φ is type-checked as follows:

1. The type of the match is φs.
2. The body of the rule for *true* must have the type $\varphi\, true$.
3. The body of the rule for *false* must have the type $\varphi\, false$.

[1] Coq will display boolean matches with the if-then-else notation.

The **typing rule for boolean matches** states this discipline in full detail.

$$\frac{\Gamma \Rightarrow s : \mathit{bool} \qquad \Gamma, z : \mathit{bool} \Rightarrow t : U \qquad \Gamma \Rightarrow u : t^z_{\mathit{true}} \qquad \Gamma \Rightarrow v : t^z_{\mathit{false}}}{\Gamma \Rightarrow \mathsf{match}\ s\ \mathsf{as}\ z\ \mathsf{return}\ t\ \mathsf{with}\ \mathit{true} \Rightarrow u \mid \mathit{false} \Rightarrow v\ \mathsf{end} : t^z_s}$$

The return function of the match in the typing rule is $\lambda z.t$. We say that the match is **dependent** if z is free in t, and that the match is **simple** otherwise. For simple matches the return annotation is usually omitted. Coq can also infer the return function for some dependent matches.

When we check the match in our proof of the boolean case analysis principle, we can assume the context

$$\Gamma \;=\; [p : \mathit{bool} \to \mathit{Prop},\ A : p\ \mathit{true},\ B : p\ \mathit{false},\ x : \mathit{bool}]$$

The typing rule will give the match the required type $p\,x$ if the following judgements are derivable:

- $\Gamma \Rightarrow x : \mathit{bool}$
- $\Gamma,\ z : \mathit{bool} \Rightarrow p\ z : \mathit{Prop}$
- $\Gamma \Rightarrow A : p\ \mathit{true}$
- $\Gamma \Rightarrow B : p\ \mathit{false}$.

6.2 Adding Bool as Inductive Type

We now extend the basic type theory of the last chapter with an inductive type *bool* for booleans.

Terms

In order to extend the type theory to include bool, we first extend the syntax for terms to include three constructors bool, true and false as well as the match construct. The grammar for terms is

$$\begin{aligned} s, t, u, v \;::=\;& x \mid \lambda x : s.t \mid s\ t \mid \forall x : s.t \mid U \\ \mid\;& \mathit{bool} \mid \mathit{true} \mid \mathit{false} \\ \mid\;& \mathsf{match}\ s\ \mathsf{as}\ z\ \mathsf{return}\ t\ \mathsf{with}\ \mathit{true} \Rightarrow u \mid \mathit{false} \Rightarrow v\ \mathsf{end} \end{aligned}$$

where x and z range over variables, and U ranges over the universes *Prop* and $\mathsf{T}_0, \mathsf{T}_1, \mathsf{T}_2, \ldots$. In a match of the form

$$\mathsf{match}\ s\ \mathsf{as}\ z\ \mathsf{return}\ t\ \mathsf{with}\ \mathit{true} \Rightarrow u \mid \mathit{false} \Rightarrow v\ \mathsf{end}$$

the variable z is bound in the term t. Substitution is defined to respect this binding structure.

Reduction

We next extend the reduction relation. In addition to the usual beta reduction, we include reductions for the match. There are two ways a match can reduce:

$$\mathsf{match}\ \mathit{true}\ \mathsf{as}\ z\ \mathsf{return}\ t\ \mathsf{with}\ \mathit{true} \Rightarrow u \mid \mathit{false} \Rightarrow v\ \mathsf{end} \triangleright_1 u$$
$$\mathsf{match}\ \mathit{false}\ \mathsf{as}\ z\ \mathsf{return}\ t\ \mathsf{with}\ \mathit{true} \Rightarrow u \mid \mathit{false} \Rightarrow v\ \mathsf{end} \triangleright_1 v$$

A simple example of a function defined with a match is negb from Chapter 2. As a Coq definition, negb was defined as

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

In our type theory the corresponding term is

$$\lambda x : \mathit{bool}.\ \mathsf{match}\ x\ \mathsf{as}\ z\ \mathsf{return}\ \mathit{bool}\ \mathsf{with}\ \mathit{true} \Rightarrow \mathit{false} \mid \mathit{false} \Rightarrow \mathit{true}\ \mathsf{end}$$

Since this is a simple match we may omit the annotations and simply write

$$\lambda x : \mathit{bool}.\ \mathsf{match}\ x\ \mathsf{with}\ \mathit{true} \Rightarrow \mathit{false} \mid \mathit{false} \Rightarrow \mathit{true}\ \mathsf{end}$$

Let *negb* denote this term. We have

$$\begin{array}{ll} & \mathit{negb}\ \mathit{true} \\ \triangleright_1 & \mathsf{match}\ \mathit{true}\ \mathsf{with}\ \mathit{true} \Rightarrow \mathit{false} \mid \mathit{false} \Rightarrow \mathit{true}\ \mathsf{end} \qquad \text{by beta reduction} \\ \triangleright_1 & \mathit{false} \qquad \text{by match reduction} \end{array}$$

Exercise 6.2.1 Explain why $\mathit{negb}\ \mathit{false} \triangleright \mathit{true}$ holds. Does $\mathit{negb}\ \mathit{false} \triangleright_1 \mathit{true}$ hold?

Exercise 6.2.2 What is the return function in the match defining negb?

Typing

We finish the definition of the type theory with *bool* by stating the typing rules for the new constructs in Figure 6.1. The typing relation $\Gamma \vdash s : t$ for the type theory with *bool* holds if the judgement $\Gamma \Rightarrow s : t$ is derivable using the rules in Figures 5.1, 5.2 and 6.1.

The Canonical Form Theorem extends to the type theory with bool as follows.

Canonical Form for the Type Theory with Bool

If $\emptyset \vdash s : t$ and s and t are both normal, then one of the following holds:

$$\textbf{bool}\ \frac{}{\emptyset \Rightarrow \mathit{bool} : \mathsf{T}_0} \qquad \textbf{true}\ \frac{}{\emptyset \Rightarrow \mathit{true} : \mathit{bool}} \qquad \textbf{false}\ \frac{}{\emptyset \Rightarrow \mathit{false} : \mathit{bool}}$$

$$\textbf{Match}\ \frac{\Gamma \Rightarrow s : \mathit{bool} \qquad \Gamma, z : \mathit{bool} \Rightarrow t : U \qquad \Gamma \Rightarrow u : t^z_{\mathit{true}} \qquad \Gamma \Rightarrow v : t^z_{\mathit{false}}}{\Gamma \Rightarrow \mathsf{match}\ s\ \mathsf{as}\ z\ \mathsf{return}\ t\ \mathsf{with}\ \mathit{true} \Rightarrow u \mid \mathit{false} \Rightarrow v\ \mathsf{end} : t^z_s}$$

Figure 6.1: Typing rules for *bool*

1. s and t are both universes and s is lower than t.
2. s is a function type and t is a universe.
3. s is a lambda abstraction and t is a function type.
4. s is bool and t is T_n for some n.
5. s is true and t is bool.
6. s is false and t is bool.

6.3 Proving that *true* and *false* are Different

We can use a simple match to prove that *true* and *false* are different. In order to remain within the basic type theory we use the impredicative definitions from Figure 5.3. The proposition we wish to prove is

$$\mathit{NOT}\ (\mathit{EQ}\ \mathit{bool}\ \mathit{true}\ \mathit{false}).$$

A term which has this type is

$$\lambda A : \mathit{EQ}\ \mathit{bool}\ \mathit{true}\ \mathit{false}.$$
$$A\ (\lambda x : \mathit{bool}.\ \mathsf{match}\ x\ \mathsf{with}\ \mathit{true} \Rightarrow \mathit{EQ}\ \mathit{bool}\ \mathit{true}\ \mathit{false} \mid \mathit{false} \Rightarrow \mathit{FALSE}\ \mathsf{end})\ A$$

Since the match is simple, we omit the return type annotation. We briefly explain why this term has the intended type. Note that

$$\mathit{EQ}\ \mathit{bool}\ \mathit{true}\ \mathit{false} \triangleright\ \forall p : \mathit{bool} \to \mathit{Prop}.\ p\ \mathit{true} \to p\ \mathit{false}$$

Let $\Gamma = [A : \mathit{EQ}\ \mathit{bool}\ \mathit{true}\ \mathit{false}]$ and let s be the term

$$\lambda x : \mathit{bool}.\ \mathsf{match}\ x\ \mathsf{with}\ \mathit{true} \Rightarrow \mathit{EQ}\ \mathit{bool}\ \mathit{true}\ \mathit{false} \mid \mathit{false} \Rightarrow \mathit{FALSE}\ \mathsf{end}.$$

It is easy to check that $\Gamma \vdash s : bool \to Prop$ and so $\Gamma \vdash A\ s : s\ true \to s\ false$. Note

$$s\ true \rhd EQ\ bool\ true\ false$$

and

$$s\ false \rhd FALSE.$$

By the conversion rule we have

$$\Gamma \vdash A\ s : EQ\ bool\ true\ false \to FALSE$$

Consequently, $\Gamma \vdash A\ s\ A : FALSE$ as desired.

We now construct this proof term with a Coq proof script using the conversion tactic *change*. We use the impredicative definitions of the logical operations from Figure 5.4.

```
Goal NOT (EQ true false).

Proof. intros A.
change (match false with true => EQ true false | false => FALSE end).
apply (A (fun x => match x with true => EQ true false | false => FALSE end)).
exact A. Show Proof. Qed.
```

The conversion tactic change will change the goal to any proposition which is convertible to the current claim. At the point where change is used in the proof above, the goal is to prove FALSE. Since

$$\mathsf{match}\ false\ \mathsf{with}\ true \Rightarrow EQ\ true\ false \mid false \Rightarrow FALSE\ \mathsf{end} \rhd FALSE$$

we can use change so that the goal becomes

$$\mathsf{match}\ false\ \mathsf{with}\ true \Rightarrow EQ\ true\ false \mid false \Rightarrow FALSE\ \mathsf{end}.$$

Note that using change corresponds to using the conversion rule Conv in Figure 5.1. By the conversion rule, a proof of

$$\mathsf{match}\ false\ \mathsf{with}\ true \Rightarrow EQ\ true\ false \mid false \Rightarrow FALSE\ \mathsf{end}$$

will also be a proof of *FALSE*. After using change we can use the assumed equation with the appropriate term of type $bool \to Prop$ to rewrite *false* to *true* so that the goal becomes

$$\mathsf{match}\ true\ \mathsf{with}\ true \Rightarrow EQ\ true\ false \mid false \Rightarrow FALSE\ \mathsf{end}$$

which reduces in one step to *EQ true false*. Note that the assumption *A* is a proof of *EQ true false*.

Consider the following variant of case analysis using negb:

$$\forall p : bool \to Prop. \forall x : bool. p\ x\ \to\ p\ (negb\ x)\ \to\ p\ true.$$

A possible proof term is

$$\begin{array}{l}\lambda\ p : bool \to Prop.\ \lambda\ x : bool.\\ \mathsf{match}\ x\ \mathsf{as}\ z\ \mathsf{return}\ p\ z\ \to\ p\ (negb\ z)\ \to\ p\ true\ \mathsf{with}\\ \quad true \Rightarrow \lambda A : p\ true.\ \lambda B : p(negb\ true).\ A\\ \mid\ false \Rightarrow \lambda A : p\ false.\ \lambda B : p(negb\ false).\ B\\ \mathsf{end}\end{array}$$

Exercise 6.3.1 What is the return type function of the match used to prove $true \neq false$?

Exercise 6.3.2 Give terms that have the following types in the empty context.

a) $\forall p : bool \to Prop. \forall x : bool.\ p\ x\ \to\ p\ (negb\ x)\ \to\ p\ false$

b) $\forall p : bool \to Prop. \forall x\, y : bool.\ p\ x\ \to\ p\ (negb\ x)\ \to\ p\ y$

Exercise 6.3.3 Let $n \geq 0$ be given. Give a type of the term

$$\forall p : bool \to \mathsf{T}_n. p\ true\ \to\ p\ false\ \to\ \forall x : bool.\ p\ x$$

in the empty context? Give a term which has this type in the empty context.

Exercise 6.3.4 Give a term of type

$$\forall p : bool \to Prop. \forall x : bool. p\ x\ \to\ p\ (negb\ (negb\ x))$$

in the empty context.

Exercise 6.3.5 Prove the following results in Coq using only the tactic *exact*. You may use the first two lemmas in the third proof term.

```
Lemma true_NOTEQ_false : NOT (EQ true false).
Lemma false_NOTEQ_true : NOT (EQ false true).
Goal forall x:bool, NOT (EQ (negb x) x).
```

6.4 Adding Nat as an Inductive Type

We next extend the type theory to include an inductive type nat of natural numbers. We could easily include both bool and nat, but we omit bool for simplicity. To include nat, we need not only a match construct, but also a fix construct for recursion.

Terms

We first extend the language of terms to include three constructors (nat, O and S) as well as a match construct for case analysis and a fix construct for recursion. As before, the match will be annotated to give a return function. The grammar for terms is

$$\begin{array}{rcl} s,t,u,v & ::= & x \mid \lambda x{:}s.t \mid s\,t \mid \forall x{:}s.t \mid U \\ & \mid & nat \mid O \mid S \\ & \mid & \mathsf{match}\ s\ \mathsf{as}\ z\ \mathsf{return}\ t\ \mathsf{with}\ O \Rightarrow u \mid S\,x \Rightarrow v\ \mathsf{end} \\ & \mid & \mathsf{fix}\ f\ (x{:}nat){:}s := t \end{array}$$

where f, x and z range over variables, and U ranges over the universes *Prop* and $\mathsf{T}_0, \mathsf{T}_1, \mathsf{T}_2, \ldots$. The binding structure of the match for *nat* is analogous to the one for *bool*. There is one new bound variable: the x in the S case. In a match of the form

$$\mathsf{match}\ s\ \mathsf{as}\ z\ \mathsf{return}\ t\ \mathsf{with}\ O \Rightarrow u \mid S\,x \Rightarrow v\ \mathsf{end}$$

the variable z is bound in the term t and the variable x is bound in the term v. In a fix of the form

$$\mathsf{fix}\ f\ (x : nat)\ :\ s\ :=\ t$$

the x is bound in s and both the f and x are bound in t. Substitution is defined to respect this binding structure.

Reduction

We next extend the reduction relation. In addition to the usual beta reduction, we include reductions for the match and fix. As with boolean matches, there are two ways a match for *nat* can reduce:

$$\mathsf{match}\ O\ \mathsf{as}\ z\ \mathsf{return}\ t\ \mathsf{with}\ O \Rightarrow u \mid S\,x \Rightarrow v\ \mathsf{end} \quad \triangleright_1 \quad u$$

$$\mathsf{match}\ S\,s\ \mathsf{as}\ z\ \mathsf{return}\ t\ \mathsf{with}\ O \Rightarrow u \mid S\,x \Rightarrow v\ \mathsf{end} \quad \triangleright_1 \quad v^x_s$$

The reader should carefully study the role of x in the second match reduction.

There are also two reduction rules for a fix:

$$(\mathsf{fix}\ f\ (x{:}nat){:}s := t)O \quad \triangleright_1 \quad (t\,{}^{f}_{\mathsf{fix}\ f\ (x:nat):s:=t})^x_O$$

$$(\mathsf{fix}\ f\ (x{:}nat){:}s := t)(S\,u) \quad \triangleright_1 \quad (t\,{}^{f}_{\mathsf{fix}\ f\ (x:nat):s:=t})^x_{S\,u}$$

That is, when a fix is applied to a term of the form O or $S\,u$, then the fix is reduced by substituting x with the argument and f with the fix itself.

$$\textbf{nat}\ \frac{}{\emptyset \Rightarrow nat : \mathsf{T}_0} \qquad \textbf{O}\ \frac{}{\emptyset \Rightarrow O : nat} \qquad \textbf{S}\ \frac{}{\emptyset \Rightarrow S : nat \to nat}$$

$$\textbf{Match}\ \frac{\Gamma \Rightarrow s : nat \qquad \Gamma, z : nat \Rightarrow t : U \qquad \Gamma \Rightarrow u : t^z_O \qquad \Gamma, x : nat \Rightarrow v : t^z_{Sx}}{\Gamma \Rightarrow \mathsf{match}\ s\ \mathsf{as}\ z\ \mathsf{return}\ t\ \mathsf{with}\ O \Rightarrow u \mid S\,x \Rightarrow v\ \mathsf{end} : t^z_s}$$

$$\textbf{Fix}\ \frac{\Gamma, f : \forall x : nat.s,\ x : nat\ \Rightarrow\ t : s}{\Gamma \Rightarrow \mathsf{fix}\ f\,(x : nat) : s\ :=\ t\ :\ \forall x : nat.s}$$

Figure 6.2: Typing rules for *nat*

Typing

We give the additional typing rules in Figure 6.2. The typing rules in Figures 5.1, 5.2 and 6.2 define a notion of being well-typed. However, this does not yet ensure termination. Consider the term

$$\mathsf{fix}\, f\,(x : nat) : nat\ :=\ f\,x$$

It is easy to use the typing rules to derive the judgement

$$\emptyset \Rightarrow (\mathsf{fix}\, f\,(x : nat) : nat\ :=\ f\,x)O\ :\ nat$$

However, it is also easy to see that the term $(\mathsf{fix}\, f\,(x : nat) : nat\ :=\ f\,x)O$ does not terminate. In order to ensure termination, we need an extra condition called guardedness.

Guardedness

We say a fix term $\mathsf{fix}\ f\,(x : nat) : s\ :=\ t$ is **guarded** if every occurrence of f in t is applied to an argument resulting from stripping off at least one constructor from x with a match. Note that

$$\mathsf{fix}\, f\,(x : nat) : nat\ :=\ f\,x$$

is not guarded. On the other hand,

$$\mathsf{fix}\, f\,(x : nat) : nat\ :=\ \mathsf{match}\, x\, \mathsf{with}\, O\ \Rightarrow\ O\ \mid\ S\,y\ \Rightarrow\ S\,(S\,(f\,y))\ \mathsf{end}$$

is guarded.

We define the typing relation $\Gamma \vdash s : t$ to hold if every fix term in Γ, s and t is guarded and the judgement $\Gamma \Rightarrow s : t$ is derivable using rules in Figures 5.1, 5.2 and 6.2.

The Canonical Form Theorem extends to the type theory with nat as follows.

Canonical Form for the Type Theory with Natural Numbers

If $\emptyset \vdash s : t$ and s and t are both normal, then one of the following holds:

1. s and t are both universes and s is lower than t.
2. s is a function type and t is a universe.
3. s is a lambda abstraction and t is a function type.
4. s is nat and t is T_n for some n.
5. s is O and t is nat.
6. s is $S\,u$ (for some term u) and t is nat.

Exercise 6.4.1 Let $\tilde{\triangleright}$ be an extension of $\triangleright_1$ which includes the more general fix reduction

$$(\mathsf{fix}\ f\,(x : nat) : s := t)u \ \triangleright_1\ (t\ {}^{f}_{\mathsf{fix}\ f\,(x:nat):s:=t})^{x}_{u}$$

Give an example of a term s such that $\emptyset \vdash s : t$ and yet there is an infinite $\tilde{\triangleright}$ reduction sequence starting from s.

Exercise 6.4.2 Without the guardedness condition there is a non-terminating term that is a proof term for *FALSE*. Hence the such generalized type theory is inconsistent and thus logically useless (recall Exercise 5.5.5).

a) Find a term φ such that
   i) $\emptyset \Rightarrow \varphi : nat \to Prop$ is derivable
   ii) φO reduces to $NOT\ (\varphi O)$.

b) Check that the following judgements are derivable.
   i) $\emptyset \Rightarrow \lambda x : \varphi O.\, xx \ :\ \varphi O \to False$
   ii) $\emptyset \Rightarrow \lambda x : \varphi O.\, xx \ :\ False$
   iii) $\emptyset \Rightarrow (\lambda x : \varphi O.\, xx)(\lambda x : \varphi O.\, xx) \ :\ FALSE$

c) There is a much shorter term φ such that the judgement $\emptyset \Rightarrow \varphi : FALSE$ is derivable. Try to find it. Hint: Exploit that a nonterminating fix may have any return type.

d) Find a term φ such that $\varphi O \triangleright S(\varphi O)$ and the judgement $\emptyset \Rightarrow \varphi O : nat$ is derivable.

6.5 Constructor Disjointness and Injectivity

We can prove $O \neq S\,O$ using the same techniques we used to prove $true \neq false$. We write 1 for $S\,O$.

```
Goal NOT (EQ O 1).

Proof.
exact (fun A => A (fun x => match x with O => EQ O 1 | S y => FALSE end) A).
Qed.
```

Let s be the term

$$\lambda x : nat.\ \text{match}\ x\ \text{with}\ O \Rightarrow EQ\ nat\ O\ 1 \mid S\,y \Rightarrow FALSE\ \text{end}.$$

The term $\lambda A.A\,s\,A$ has the intended type since

$$A : EQ\ nat\ O\ 1 \vdash A\,s : EQ\ nat\ O\ 1 \to FALSE$$

Next we prove that the constructor S is injective. The proof uses Coq's predefined predecessor function.

```
Goal forall x y : nat,
EQ (S x) (S y) -> EQ x y.

Proof. intros x y A p. exact (A (fun z => p (pred z))). Qed.
```

Exercise 6.5.1 Do the proof that the constructor S is injective with paper and pencil in the basic type theory with *nat*.

Exercise 6.5.2 Prove the following goal using only *exact*.

```
Goal forall x:nat, NOT (EQ O (S x)).
```

6.6 Inductive Proofs are Recursive Proofs

Here is a proof of the so-called **induction principle** for *nat*.

```
Goal forall p : nat -> Prop,
p O -> (forall x : nat, p x -> p (S x)) -> forall x : nat, p x.

Proof. exact (fun p A B => fix f x :=
                match x as z return p z with
                | O => A
                | S x' => B x' (f x')
                end).
Qed.
```

Note that the proof term employs a recursion on x using a fix. Whenever we use the tactic *induction* for a variable of type *nat*, Coq will synthesize the proof term with a lemma *nat_ind* taking the proposition of the goal as type. Here is an example.

```
Goal forall x, EQ (x + 0) x.

Proof. intros x. induction x. exact (fun p A => A).
intros p. exact (IHx (fun z => p (S z))).
Show Proof. Qed.
```

This script synthesizes the following proof term (argument types deleted):

```
fun x => nat_ind
           (fun z => EQ (z + 0) z)
           (fun p (A : p 0) => A)
           (fun x (IHx : EQ (x + 0) x) => fun p => IHx (fun z => p (S z)))
           x
```

It takes some effort to verify the proof term by hand. The important message to take home here is that inductive proofs in Coq are recursive proofs with fix and match.

Exercise 6.6.1 Prove the following goals in Coq using only *exact*.

```
Goal forall p : nat -> Prop,
p O -> (forall x : nat, p x -> p (S x)) -> forall x : nat, p x.

Goal forall (p : nat -> Prop) (x : nat),
p O -> (forall x : nat, p x -> p (S x)) -> p x.

Goal forall p : nat -> Prop,
p O -> p (S O) -> (forall x : nat, p x -> p (S (S x))) -> forall x : nat, p x.

Goal forall p : nat -> Prop,
p O -> (forall x : nat, p (S x)) -> forall x : nat, p x.
```

Exercise 6.6.2 Here is the so-called *inversion principle* for *nat*.

```
Goal forall x : nat,
x = 0 \/ exists y, x = S y.
```

Writing a proof script for this goal is straightforward. Things become more challenging if we reformulate the goal with the impredicative definitions of disjunction, equality, and existential quantification.

```
Goal forall x : nat, OR (EQ x O) (EX (fun y => EQ x (S y))).
```

Coming up with a proof term for this goal with paper and pencil is tedious. Things are more enjoyable if you synthesize the proof term interactively with a Coq proof script.

6.7 Regular Inductive Types

What we have learned about *bool* and *nat* generalizes to a class of inductive types we call *regular inductive types*. In Coq, the definition of a regular inductive type takes the form

Inductive $c\ (x_1 : s_1) \cdots (x_m : s_m)\ :\ U\ :=\ c_1 : t_1 \mid \cdots \mid c_n : t_n\,.$

where $m, n \geq 0$ and U ranges over the universes. The definition introduces a **type constructor** c and **member constructors** $c_1, \ldots, c_n$. The argument variables $x_1, \ldots, x_m$ given for the type constructor c are called **parameters**. The essential constraint for regular inductive types says that the constructor c can occur only in the leftmost position of an application $cx_1 \ldots x_m$ in the terms $t_1, \ldots, t_n$ specifying the types of the member constructors. The type of a member constructor c_i is in fact $\forall x_1 : s_1 \cdots \forall x_m : s_m.\ t_i$.

Things will become clearer if you review the inductive type we have seen so far. The types *bool*, *nat*, *False*, and *True* are regular inductive types without parameters. The type constructors *prod*, *list*, *option*, *and*, *or*, and *ex* are regular inductive type constructors with one or two parameters. The inductive type constructor *eq* for equations is not regular. We will study non-regular inductive types later. The matches for non-regular inductive types take a special form where the return function takes additional arguments.

6.8 The Elim Restriction

We call inductive type constructors that target the universe *Prop* **inductive predicates**, and their member constructors **proof constructors**. There is a restriction on the form of matches for inductive predicates known as **elim restriction**. The elim restriction is needed for consistency and can be seen as the price to pay for the impredicativity of *Prop*. The elim restriction says that a match on proof terms must not leak information to non-propositional types. A match whose return type is a proposition always satisfies the elim restriction. On the other hand, the match

```
Check fun X : True \/ True =>
match X with
| or_introl _ => true
| or_intror _ => false
end.
% Error : Incorrect elimination of "X" in the inductive type "or"
```

violates the elim restriction since it discriminates on proofs and returns booleans.

6.9 Remarks

We have now arrived at an explicit definition of a dependent type theory with inductive types for *bool* and *nat*. The theory can define many functions on natural numbers and at the same time prove many results about natural numbers. All functions definable in the theory are computable and terminating. The restriction to terminating recursion ensures consistency of the theory and decidability of type checking (which subsumes proof checking).

The formulation of computational type theories is a major achievement of the second half of the 20th century. The enterprise started with Gödel's System T in the late 1950's. Gödel's system can express many computable functions on natural numbers but cannot express propositions and proofs. Further steps were the discovery of the propositions-as-types principle and the development of dependent type theories. The accommodation of the natural numbers as inductive type started with the work of Martin-Löf in the 1970's and took its current form with match and fix and the guardedness condition around 1990.

7 More Fun with Bool and Nat

In this chapter we prove various theorems about booleans and numbers making full use of Coq's tactics and notational devices. This should give you an impression of how mathematical results can be established in Coq. The topics of this chapter include complete induction, primitive recursion, and Cantor's theorem.

7.1 Disjointness and Injectivity of Constructors

Recall the proofs of the disjointness and injectivity of the member constructors of *nat* you have seen in the last chapter.

```
Goal forall x, S x <> 0.   (* Disjointness *)
```

```
Proof. intros x A. change (match S x with 0 => True | S _ => False end).
rewrite A. auto. Qed.
```

```
Goal forall x y, S x = S y -> x = y.   (* Injectivity *)
```

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

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

```
Goal forall x, S x <> 0.
```

```
Proof. intros x A. discriminate A. Qed.
```

```
Goal forall x y, S x = S y -> x = y.
```

```
Proof. intros x y A. injection A. auto. Qed.
```

Exercise 7.1.1 Prove the following goal twice: Once with *discriminate* and once with *change* and without *discriminate*.

```
Goal forall (X : Type) (x : X) (xs : list X),
cons x xs <> nil.
```

Exercise 7.1.2 Prove the following goals twice: once with *injection* and once with *change* and without *injection*.

(a)
```
Goal forall (X : Type) (x y x' y' : X),
  pair x y = pair x' y' -> y = y'.
```

(b) **Goal** forall (X : Type) (x x' : X) (xs xs' : list X),
cons x xs = cons x' xs' −> xs = xs'.

Exercise 7.1.3 Prove the following goals.

(a) **Goal** forall x, negb x <> x.

(b) **Goal** forall x, S x <> x.

(c) **Goal** forall x y z, x + y = x + z −> y = z.

(d) **Goal** forall x y : nat, x = y \/ x <> y.

7.2 Booleans as Propositions

The types *bool* and *Prop* are very different: While *bool* is an inductive type with exactly two elements, *Prop* is the universe of propositions. However, there is a canonical embedding of *bool* into *Prop*.

```
Coercion bool2Prop (x : bool) := if x then True else False.
```

The command *Coercion* defines *bool2Prop* as a function that is automatically inserted when a term of type *bool* is used in a context where a proposition is required.

```
Compute ~false.
% False → False
Set Printing All.
Check ~false.
% not (bool2Prop false) : Prop
Unset Printing All.
```

From now on we will always assume that the coercion *bool2Prop* is active.

```
Goal forall X : Prop,
(exists b : bool, b <−> X) <−> X \/ ~X.
Proof. unfold bool2Prop. split.
intros [b A]. destruct b ; tauto.
intros [A|A]. exists true. tauto. exists false. tauto. Qed.
```

The proof unfolds the coercion *bool2Prop* explicitly since the tactic *tauto* will not do it.

Exercise 7.2.1 Prove the following goals.

(a) **Goal** forall x : bool, ~x <−> negb x.

(b) **Goal** forall x y : bool, x /\ y <−> andb x y.

(c) **Goal** forall x y : bool, x \/ y <−> orb x y.

(d) **Goal** forall (b : bool) (X : Prop), (b <−> X) −> (~b <−> ~X).

7.3 Boolean Equality Tests

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

```
Fixpoint eq_nat (x y : nat) : bool :=
match x, y with
| 0, 0 => true
| S x', S y' => eq_nat x' y'
| _, _ => false
end.
```

We prove that boolean equality on *nat* agrees with Coq's equality.

```
Goal forall x y : nat,
eq_nat x y <-> x = y.
```

```
Proof. induction x ; destruct y ; split ; simpl ; intros A ; try tauto ; try discriminate A.
f_equal. apply IHx. exact A.
apply IHx. injection A. auto. Qed.
```

The proof script uses several advanced features of Coq's tactic language and deserves careful studying. The initial induction, case analysis, and split leave us with 8 subgoals. Four of them are solved by *tauto*, and 2 of them are solved by *discriminate*. Note the use of the **tactical** *try*. It is needed since *tauto* and *discriminate* will fail on some of the goals. A command *try t* behaves like the tactic *t* if *t* succeeds but leaves the goal unchanged and succeeds if *t* fails. For the remaining two subgoals, the inductive hypothesis is applied. Note that *IHx* is a proof of a quantified equivalence and that the tactic *apply* is smart enough to choose the correct implication from the equivalence.

Exercise 7.3.1 Write boolean equality tests for the following types and prove that they agree with Coq's equality.

a) *bool*

b) *list nat*

7.4 Boolean Order on Nat

We define the canonical order on *nat* as a boolean test.

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

We use Coq's notation command to define the usual notations for comparisons.[1]

```
Notation "s >= t" := (leb t s).
Notation "s <= t" := (leb s t).
Notation "s > t" := (leb (S t) s).
Notation "s < t" := (leb (S s) t).
```

We can now write the following.

```
Compute 7 > 5.
% true : bool

Compute 4 < 3 -> False.
% False → False : Prop

Set Printing All.

Check 1 < 0 -> False.
% bool2Prop (leb (S(SO)) O) → False : Prop
```

We prove some simple facts about comparisons.

```
Lemma le_refl x :
x <= x.

Proof. induction x ; simpl ; auto. Qed.

Lemma le_trans x y z :
x <= y -> y <= z -> x <= z.

Proof. revert y z. induction x ; simpl. now auto.
destruct y ; simpl. now auto.
destruct z ; simpl. now auto.
exact (IHx _ _). Qed.

Goal forall x y z,
x < y -> y < z -> S x < z.

Proof. intros x y z. exact (le_trans (S (S x)) (S y) _). Qed.

Goal forall x y,
~ (x < y /\ y <= x).

Proof. induction x ; destruct y ; try (simpl ; tauto). exact (IHx _). Qed.
```

In the above proofs conversion plays a major role. Make sure you understand every detail.

Exercise 7.4.1 Prove the following goals.

(a) `Goal forall x, ~ x < x.`

(b) `Goal forall x y, x <= y -> x < y \/ x = y.`

[1] We overwrite Coq's definitions of these notations. Coq defines the order on *nat* in a different way we will explain later.

(c) **Goal** forall x y, x<y \/ y<=x.

(d) **Goal** forall x y, negb (x <= y) = (x > y).

(e) **Goal** forall x y, x = y <-> x <= y /\ y <= x.

(f) **Goal** forall x y, x < y \/ x = y \/ x > y.

(g) **Goal** forall x y z, x < y <-> z + x < z + y.

(h) **Goal** forall x y, x <= y <-> exists z, x + z = y.

7.5 Complete Induction and Size Induction

Recall the basic induction principle for *nat*.

```
Lemma nat_ind (p : nat -> Prop) :
p 0 -> (forall n, p n -> p (S n)) -> forall n, p n.

Proof. intros A B.
exact (fix f x := match x as z return p z with 0 => A | S x' => B x' (f x') end). Qed.
```

This principle is applied whenever the tactic *induction* is used. It can be used to establish the seemingly stronger principle of *complete induction*, which says that in the induction step it suffices to prove pn under the assumption that pm is provable for all $m < n$. The reformulated induction step subsumes the base case.

```
Lemma complete_induction (p: nat->Prop) :
(forall n, (forall m, m < n -> p m) -> p n) -> forall n, p n.

Proof. intros A n. apply A. induction n ; simpl. tauto.
intros m B. apply A. intros k C. apply IHn.
exact (le_trans _ _ _ C B). Qed.
```

The proof is remarkable in that it first reduces the claim to a proposition that can be shown by ordinary induction.

A further generalization of complete induction is **size induction**. Size induction works for any type X and any predicate $p : X \to Prop$ provided there is a size function $f : X \to nat$. It suffices to prove px under the assumption that there is a proof of py for all y smaller than x.

```
Lemma size_induction (X : Type) (f : X -> nat) (p: X ->Prop) :
(forall x, (forall y, f y < f x -> p y) -> p x)
-> forall x, p x.

Proof. intros A x. apply A. induction (f x) ; simpl. tauto.
intros y B. apply A. intros z C. apply IHn.
exact (le_trans _ _ _ C B). Qed.
```

The proof script is similar to the proof script for complete induction, with the remarkable difference that the induction tactic is applied to the term *fx*, which is not a variable. This means that a local lemma is created that is applied to *fx* after it has been shown by induction. Here is a proof script that explicitly states the auxiliary lemma with the tactic *assert*.

```
Goal forall (X : Type) (f : X -> nat) (p: X ->Prop),
(forall x, (forall y, f y  < f x -> p y)  -> p x)
-> forall x, p x.

Proof. intros X f p A x. apply A.
assert (L :  forall  n y,  f  y < n -> p y).
clear x. induction n ; simpl. tauto.
intros y B. apply A. intros z C. apply IHn.
exact (le_trans _ _ _ C B).
exact (L (f x)). Qed.
```

Note the use of the tactic *clear*. It clears away an assumption that is not needed for the proof of *L*.

Exercise 7.5.1 Prove the following proposition in two ways.

```
Goal forall p : nat -> Prop,
p 0 -> p 1 -> (forall n, p n -> p (S (S n))) -> forall n, p n.
```

a) With a proof term using fix and match.

b) With complete induction. After a few steps you will be left with the claim $n \leq Sn$. Prove this claim inline with the induction tactic. Use the tactic *clear* to clear away unnecessary assumptions before you apply the induction tactic.

Exercise 7.5.2 Prove complete induction using size induction.

Exercise 7.5.3 Prove the basic induction principle for *nat* using complete induction.

Exercise 7.5.4 Prove the following goal formulating a principle we may call zigzag induction. Use the induction tactic.

```
forall  p : nat -> Prop,
p O ->
(forall  x, p x -> p (S (S x))) ->
(forall  x, p (S x) -> p x) ->
forall  x, p x.
```

7.6 Case Analysis with case_eq

Kaminski's equation 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 $f\ \mathit{false}$. If we do the proof with *destruct*, we face the problem that *destruct* $(f\ \mathit{true})$ does not provide the equations $f\ \mathit{true} = \mathit{true}$ and $f\ \mathit{true} = \mathit{false}$ coming with the case analysis. Fortunately, there is a variant of *destruct* called *case_eq* providing the equations.

```
Goal forall (f : bool -> bool) (x : bool),
f (f (f x)) = f x.

Proof. destruct x ; case_eq (f true) ; case_eq (f false) ; congruence. Qed.
```

Note the use of *congruence*, an automation tactic that does simple equational proofs. Replace the semicolon before *congruence* with a period and solve the 8 subgoals by hand to understand.

For boolean case analyses, the tactic *case_eq* can be simulated with the following lemma.

```
Lemma case_eq_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 forall (f : bool -> bool) (x : bool),
f (f (f x)) = f x.

Proof. destruct x ;
pattern (f true) ; apply case_eq_bool ;
pattern (f false) ; apply case_eq_bool ;
congruence. Qed.
```

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

Exercise 7.6.1 Prove the following variant of Kaminski's equation.

```
Goal forall (f g : bool -> bool) (x : bool),
f (f (f (g x))) = f (g (g (g x))).
```

7.7 Specification of Functions

In Chapter 2 we specified functions on *bool* and *nat* by systems of characteristic equations. In Coq we can express such specifications as predicates. Here is a specification of addition.

```
Definition addition (f : nat -> nat -> nat) : Prop :=
forall x y,
f 0 y = y  /\
f (S x) y = S (f x y).
```

Given the specification of addition, there are two questions:

1. Does Coq's predefined addition function *plus* satisfy the specification?
2. Are any two functions satisfying the specification equivalent in the sense that they yield the same results for the same arguments?

We answer both questions positively.

```
Goal addition plus.
```

```
Proof. unfold addition. simpl. auto. Qed.
```

```
Goal forall f g x y,
addition f -> addition g -> f x y = g x y.
```

```
Proof. intros f g x y A B. induction x.
destruct (A 0 y) as [C _]. destruct (B 0 y) as [D _]. congruence.
destruct (A x y) as [_ C]. destruct (B x y) as [_ D]. congruence. Qed.
```

Note that we need fix and match to show that *addition* has a solution. We also need fix and match to show that *addition* has at most one solution up to equivalence. This is the case since we need induction to establish this fact, and fix and match are needed to establish induction.

Next we specify a function known as Ackermann's function.[2]

```
Definition ackermann (f : nat -> nat -> nat) : Prop :=
forall 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).
```

Ackermann argued the existence and uniqueness of his function as follows. Since for any two arguments exactly one of the 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.

Ackermann's termination argument is outside the scope of Coq's termination checker. Coq will insist that for every fix there is a single argument that is structurally decreased by every recursive application. The problem can be resolved by formulating Ackermann's function with two nested recursions.

[2] 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 7.8.2 you will show that Ackermann's function can be computed with higher-order primitive recursion.

```
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 returns a recursive function. Each of the two recursions is structural on its single 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 forall 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 7.7.1 Specify multiplication and subtraction and prove that Coq's predefined functions satisfy the specifications. Also prove that two functions agree on all arguments if they satisfy one of the specifications.

7.8 Primitive Recursion

The following function known as **primitive recursion** captures most of the power of guarded recursion for *nat*.[3]

```
Definition primrec (t : nat -> Type) (A : t 0) (B : forall n, t n -> t (S n))
: forall n : nat, t n
:= fix f n := match n as z return t z with
              | 0 => A
              | S n' => B n' (f n')
              end.
```

[3] What we define here is higher-order primitive recursion. Higher-order primitive recursion is much more powerful than first-order primitive recursion as used in recursion theory.

If we are allowed only a single use of *fix* for *nat*, we would define *primrec* and then express all further recursions with *primrec*. To start with, primitive recursion gives us the basic induction principle for *nat*.

```
Check fun p : nat -> Prop => primrec p.
% forall p : nat → Prop, p 0 → (forall n : nat, p n → p (S n)) → forall n : nat, p n
```

Note that the application of *primrec* to *p* type checks due to the subsumption rule (see Section 5.4).

The definition of addition with primitive recursion is straightforward.

```
Definition add := primrec
                  (fun _ => nat -> nat)
                  (fun y => y)
                  (fun _ r y => S (r y)).

Compute add 3 7.
% 10
```

We prove that *add* satisfies the specification *addition*.

```
Goal addition add.

Proof. intros x y ; simpl. auto. Qed.
```

Exercise 7.8.1 Write a multiplication function using primitive recursion. Prove that your function agrees with Coq's multiplication function *mult*.

Exercise 7.8.2 Write an Ackermann function using *primrec* twice. Prove that your function satisfies the specification *ackermann*.

Exercise 7.8.3 We specify primitive recursion as follows.

```
Definition primitive_recursion
(f : forall t : nat -> Type, t 0 -> (forall n, t n -> t (S n)) -> forall n, t n) : Prop :=
forall t A B n,
f t A B 0 = A /\
f t A B (S n) = B n (f t A B n).
```

Show that *primrec* satisfies this specification and that any two functions satisfying the specification are equivalent.

7.9 Bool and Nat Are Not Equal

We now prove that the types *bool* and *nat* are not equal. For this we need a predicate on types that distinguishes *bool* and *nat*. For this we use the property that a type has at most two elements.

```
Goal bool <> nat.

Proof. intros A.
pose (p X := forall x y z : X, x=y \/ x=z \/ y=z).
assert (B : ~ p nat).
intros B. destruct (B 0 1 2) as [C|[C|C]] ; discriminate.
apply B. rewrite <- A. intros [] [] [] ; auto. Qed.
```

The tactic *pose* provides for local definitions.[4] We use it to define the discriminating predicate p. The tactic *assert* is used to establish a proof B of $\neg\, p\, nat$. Once we have B, we apply it to the claim, which leaves us with the claim $p\ nat$. Rewriting with the assumed proof A of $bool = nat$ yields the claim $p\ bool$. Proving this claim is routine.

Here is a proof of a disequation between propositions making clever use of the tactics *assert* and *tauto*.

```
Goal forall X : Prop, (~X) <> X.

Proof. intros X A.
assert (B : ~(~X <-> X)) by tauto.
apply B. rewrite A. tauto. Qed.
```

Exercise 7.9.1 Prove the following goals.

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

7.10 Cantor's Theorem

Cantor's theorem says that the power set of a set is always strictly larger than the set. For the proof of his theorem Cantor used a technique known as diagonalisation. It turns out that Cantor's result carries over to type theory.

Given two types X and Y, we call X *smaller than* Y if there exists no surjection from X to Y.[5]

```
Definition smaller (X Y : Type) : Prop :=
~ exists f : X -> Y, forall y, exists x, f x = y.
```

We show that every type X is smaller than $X \to bool$.

```
Goal forall X : Type, smaller X (X -> bool).
```

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

[5] A surjection from X to Y is a function from X to Y that reaches all members of Y.

```
Proof. intros X [f A].
pose (g x := negb (f x x)).
destruct (A g) as [a B].
absurd (f a a = g a).
unfold g. destruct (f a a) ; discriminate.
rewrite B. reflexivity. Qed.
```

The proof assumes a surjection f from X to $X \to bool$ and constructs a proof of *False*. To do so, the proof defines a *spoiler function* $g : X \to bool$ that cannot be reached by f. The trick is to define g in terms of f such that for the a such that $fa = g$ (exists by assumption) the equation $faa = ga$ is disprovable. This is the case if we define g as $gx := negb(fxx)$. We use the tactic *pose* to define g and the tactic *absurd* to disprove $faa = ga$. In general, a command *absurd s* replaces the current goal with goals for $\neg s$ and s.

Exercise 7.10.1 Prove that two types X and Y are different if X is smaller than Y.

```
Goal forall X Y : Type, smaller X Y -> X <> Y.
```

Exercise 7.10.2 Use the diagonalisation technique to prove the following goals. In each case define the spoiler function as in the example above where *negb* is replaced by some other function.

(a) `Goal smaller nat (nat -> nat).`

(b) `Goal smaller nat (nat -> Prop).`

Exercise 7.10.3 Prove the following generalized diagonalisation theorem. All diagonalisation results stated so far can be obtained as instances of the general result. The theorem shows that diagonalisation can be formulated as a purely logical result not depending on the inductive types *bool* and *nat*.

```
Definition strong (X : Type) : Prop :=
exists f : X -> X, forall x, f x <> x.

Theorem Cantor (X Y : Type) :
strong Y -> smaller X (X -> Y).
```

Exercise 7.10.4 The predicate *smaller* is not a well-behaved order predicate if the right type is *False*. We fix the problem with a reflexive and transitive greater-equal predicate for types.

```
Definition ge (X Y : Type) : Prop :=
Y -> exists f : X -> Y, forall y, exists x, f x = y.
```

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

b) Prove the following goal.

```
Goal forall (X : Type) (Y : Prop) ,
X -> Y <-> (exists x : X, True) -> Y.
```

c) Prove that *ge* is reflexive and transitive.

d) Prove $ge\ X\ False$ for all types X.

e) Prove $\neg ge\ X\ Y \to X \neq Y$ for all types X and Y.

f) Prove $Y \to smaller\ X\ Y \to ge\ X\ Y \to False$ for all types X and Y.

Exercise 7.10.5 Using diagonalisation one can show that there is no injective function from the power set of a set back to the set. To come up with the right spoiler function for this proof is much harder than for the proof of the surjective Cantor Theorem. Complete the following proof of a similar result in type theory.

```
Definition injective (X Y : Type) (f : X->Y) : Prop :=
forall x x' : X, f x = f x' -> x = x'.

Goal forall (X : Type) (f : (X -> Prop) -> X), ~injective f.

Proof. intros X f A.
pose (g x := exists h, f h = x /\ ~ h x).
absurd (~ g (f g)).    ...
```

7.11 Projections

We call a function **arithmetic** if it has a type $nat \to \cdots \to nat$ with $n \geq 1$ arrows. An arithmetic function is **constant** if it always returns the same value. An arithmetic function is a **projection** if it always returns one of its arguments.

We can write functions that yield constant functions and projections for a given number of arguments. We start with a function *AF* that given a number n yields the type $nat \to \cdots \to nat$ with n arrows.

```
Fixpoint AF (n : nat) : Type :=
match n with
| O => nat
| S n' => nat -> AF n'
end.

Compute AF 3
% nat → nat → nat → nat
```

We now write a function K that given n and c returns a constant function taking n arguments and returning c.

```
Fixpoint K (n c : nat) : AF n :=
match n as z return AF z with
| O => c
| S n' => fun _ => K n' c
end.

Compute K 3 7.
% fun _ _ _ : nat ⇒ 7 : AF 3
```

Note that K employs a dependent match and that the dependency cannot be avoided since the return type depends on the first argument of K.

Exercise 7.11.1 Write a function $P : \forall n{:}nat.\ nat \to AF\ n$ that given two arguments n and k returns a projection returning its $k{+}1$-th argument. For instance, $P\ 4\ 2$ should reduce to $fun\ \_\ \_\ x\ \_ : nat \Rightarrow x$.

Exercise 7.11.2 Write a function K' that is equivalent to K using primitive recursion. Prove the equivalence.

7.12 Tactics Summary

change t	Converts the claim to term t.
pattern t	Converts the claim to a beta redex $s\ t$ by abstracting out all subterms t.
pose (x := t)	Local definition.
assert (x : t)	Local lemma.
absurd t	Replaces current goal with goals for $\neg t$ and t.
discrimate t	Solves goal if t is a proof of a disequation contradicting constructor disjointness.
injection t	Weakens claim by equations following by constructor injectivity from the equation proved by t.
case_eq t	Like destruct t but adds equation $t = \ldots$ to claims.
congruence	Tries to solve current goal by equational reasoning.
clear x	Deletes assumption x.
try t	Applies tactic t. If t fails, *try* t succeeds and leaves the goal unchanged.

8 Inductive Predicates with Proper Arguments

In this chapter we will further study inductively defined predicates (i.e., inductive types which live in the universe *Prop*). The logical operations such as conjunction and existential quantification were defined as inductive predicates. The inductive predicates we consider in this chapter will have so-called proper arguments. Equality is an example of an inductive predicate with a proper argument. The inclusion of proper arguments means we must include a further annotation on dependent matches in order to specify the return type.

8.1 Inductive Predicates

In Coq, the definition of an inductive predicate takes the form

$$\begin{aligned}&\mathsf{Inductive}\ \ c\ (x_1 : s_1) \cdots (x_m : s_m)\ :\ \forall x_{m+1} : s_{m+1}. \ldots \forall x_{m+k} : s_{m+k}.\mathit{Prop}\ :=\\ &\qquad\qquad \mid c_1 : t_1\\ &\qquad\qquad \ldots\\ &\qquad\qquad \mid c_n : t_n\,.\end{aligned}$$

where $m, k, n \geq 0$. As with regular inductive types, the definition introduces a **type constructor** c and **member constructors** $c_1, \ldots, c_n$. As explained in Section 6.8 (see also Section 3.13) we also call type constructors such as c **inductive predicates** and call the member constructors $c_1, \ldots, c_n$ **proof constructors**. An **inductive proposition** is a term of the form $c\,u_1 \cdots u_{m+k}$ of type *Prop* where c is an inductive predicate. The variables $x_1, \ldots, x_m$ in the definition are called the **parameters** of the inductive definition. Likewise, the first m arguments of c are called **parameteric arguments**. The last k arguments of c are called **proper arguments**. That is, given an inductive proposition $c\,u_1 \cdots u_m u_{m+1} \cdots u_{m+k}$, $u_1, \ldots, u_m$ are parametric arguments and $u_{m+1}, \ldots, u_{m+k}$ are proper arguments. In this case, we say the inductive predicate has m parameters and k proper arguments. There are restrictions on how c can occur in each type t_i (e.g., "strict positivity"), but we will not discuss these restrictions here.

In general, we will speak of **introduction principles** and **elimination principles** for inductive predicates. This terminology corresponds to the introduction

and elimination rules for logical operations as described in Section 3.11. An **introduction principle** describes how we can *prove* an inductive proposition. An **elimination principle** describes how we can *use* an assumed inductive proposition. We will sometimes also refer to an elimination principle as an **induction principle** when there is recursion involved. Each inductive predicate will have an introduction principle given by each proof constructor. Also, each inductive predicate will have an elimination principle which we will identify and prove as a lemma using a match and possibly a fix.

Recall the inductive definition of equality.

```
Inductive eq (X : Type) (x : X) : X−> Prop :=
| eq_refl : eq x x.
```

Note that the inductive predicate *eq* has two parameters (the first of which is implicit) and one proper argument.

A simpler inductive predicate that is very similar to *eq* is a test for zero.

```
Inductive zerop : nat −> Prop :=
| zerop_I : zerop O.
```

Note that *zerop* has one proper argument and no parameters.

Inductive predicates can also be recursive. For example, the even numbers can be defined as an inductive predicate as follows.

```
Inductive even : nat −> Prop :=
| evenO : even 0
| evenS : forall n, even n −> even (S (S n)).
```

The type constructor *even* has one proper argument an no parameters. We may express this definition informally with two inference rules.

$$\frac{}{even\ 0} \qquad\qquad \frac{even\ n}{even\ (S(S\ n))}$$

In this chapter we will consider these examples of inductive predicates. We will also consider an inductive predicate giving the $\leq$ relation on *nat*. Along the way, we will learn some new Coq tactics.

8.2 Singleton Zero Predicate

Consider the inductive predicate zerop with one proper argument.

```
Inductive zerop : nat −> Prop :=
| zeropI : zerop O.
```

The proof constructor *zeropI* justifies that 0 satisfies the predicate.

```
Goal zerop 0.
Proof. exact zeropI. Qed.
```

In fact, zeropI is the (only) introduction principle for zerop.

How do we know that 1 does not satisfy the zerop predicate? In order to prove 1 does not satisfy zerop, we need an elimination principle for zerop. Such an elimination principle comes from the match for zerop.

Since zerop has a proper argument, the return type of the match for zerop will have a new kind of dependency. Here is a match for *zerop*.

```
Check
fun (p:nat -> Type) A n (B:zerop n) =>
match B in zerop z return p z with zeropI => A end.
% forall p : nat -> Type,
% p O ->
% forall n : nat, zerop n -> p n
```

Note that z is a local variable connecting the proper argument of *zerop* with the return type. The reduction rule for the match for *zerop* is simple.

$$\text{match } \mathit{zeropI} \text{ in } \mathit{zerop}\ z \text{ return } t \text{ with } \mathit{zeropI} \Rightarrow u \text{ end } \triangleright_1 u$$

The typing rule for the match for *zerop* is also straightforward.

$$\frac{\Gamma \Rightarrow s : \mathit{zerop}\ s_1 \qquad \Gamma, z : \mathit{nat} \Rightarrow t : U \qquad \Gamma \Rightarrow u : t_0^z}{\Gamma \Rightarrow \text{match } s \text{ in } \mathit{zerop}\ z \text{ return } t \text{ with } \mathit{zeropI} \Rightarrow u \text{ end } : t_{s_1}^z}$$

There is one subtlety: The return type t is allowed to live in any universe U. In general, the elim restriction (see Section 6.8) would mean the universe in this typing rule must be *Prop*. In some special cases, the elim restriction does not apply; zerop is one of these cases.

We now prove an elimination principle for zerop.

```
Lemma zeropE :
forall p:nat -> Prop, p 0 -> forall x:nat, zerop x -> p x.
Proof.
exact (fun p A x B => match B in zerop z return p z with zeropI => A end).
Qed.
```

We can also prove this using tactics. In particular, destruct will build the match for us.

```
Lemma zeropE :
forall p:nat -> Prop, p 0 -> forall x:nat, zerop x -> p x.
Proof. intros p A x B. destruct B. exact A. Qed.
```

Now that we have an elimination principle, we return to the question of how we can prove that 1 does not satisfy the predicate. It is enough to give a property p which 0 satisfies but 1 does not. We can easily do this with a match.

```
Goal ~zerop 1.
Proof. intros A.
exact (zeropE (fun n => match n with O => True | S _ => False end) I A).
Qed.
```

Remember

One might expect that it is possible to prove that 1 does not satisfy zerop using destruct. For example, we could start the proof as follows.

```
Goal ~zerop 1.
Proof. intros A. destruct A.
```

Unfortunately, this leads to a dead end. The reason is that the assumption A has type zerop 1 and 1 is not a variable. Whenever there is a non-variable in the proper argument position of zerop, destruct will forget information. One can explicitly remember that 1 was in this position using the remember tactic. The remember tactic replaces every occurrence of a given term by a fresh variable and assumes an equation between the variable and the term. Consider the following proof script.

```
Goal ~zerop 1.
Proof. remember 1. intros A. destruct A. discriminate Heqn. Qed.
```

After remember 1 the assumptions are n:nat and Heqn:n = 1 and the claim is zerop n. After we introduce the assumption A:zerop n we can get more information from destruct. Since there is a variable (namely, n) in the proper argument position, destruct can substitute the appropriate term for this n for each constructor of the inductive predicate. In the case of zerop, the n will be replaced by 0. Hence the assumption Heqn will be 0 = 1 and we can finish the proof using discriminate.

We often need to manipulate assumptions of inductive predicates with proper arguments so that the proper arguments are distinct variables. We call this **linearizing** the arguments. The tactic remember is one way of linearizing the arguments. Another way is to use an assert to form an appropriate lemma.

```
Goal ~zerop 1.
Proof. intros A.
assert (B:forall n:nat, zerop n -> n <> 1).
intros n C. destruct C. discriminate.
apply (B 1). exact A. reflexivity.
Qed.
```

One can also prove the lemma given by the assert using the elimination principle. This is not surprising since destruct and the elimination principle zeropE both contain the essential information given by a match for zerop.

```
Goal ~zerop 1.
Proof. intros A.
assert (B:forall  n:nat, zerop n -> n <> 1).
apply zeropE. discriminate.
apply (B 1). exact A. reflexivity.
Qed.
```

Inversion

The easiest way to prove the goal is to use the tactic inversion. The inversion tactic is like destruct but keeps more information. In particular, inversion derives equations and then processes these equations using discriminate and injection.

```
Goal ~zerop 1.
Proof. intros A. inversion A. Qed.
```

Decidability

A proposition s is **decidable** if the proposition $s \vee \neg s$ is provable. Show that the following propositions are decidable. We can generalize this to predicates. For example, we say $s : u \to Prop$ is **decidable** if $\forall x : u \to Prop.sx \vee \neg sx$ is provable. This generalizes in an evident way to predicates taking n arguments.

We now prove the predicate zerop is decidable. An easy way to accomplish this is to define an equivalent boolean test.

```
Definition zerob (n:nat) : bool :=
match n with
| O => true
| S _ => false
end.
```

The equivalence of zerob and zerop follows by an easy case analysis on the natural number n and inversion.

```
Lemma zerob_zerop n : zerob n <-> zerop n.
Proof. destruct n; simpl.
split. intros _. exact zeropI. tauto.
split. tauto. intros A. inversion A. Qed.
```

Decidability of zerop follows easily.

```
Lemma zerop_dec n : zerop n \/ ~zerop n.
Proof.
assert (A:zerob n <-> zerop n). now apply zerob_zerop.
assert (B:zerob n \/ ~zerob n). destruct (zerob n); simpl; tauto.
tauto. Qed.
```

Exercise 8.2.1 Prove the following in four different ways: using exact, using assert, using remember and using inversion.

```
Goal ~zerop 2.
```

Exercise 8.2.2 Suppose $X : nat \to Type$, $n : nat$ and $A : zerop\ n$. One can use A to cast between the types $X\ n$ and $X\ 0$. Show this by defining two functions and proving the lemmas below. (We assume the argument n to zerop_cast1 and zerop_cast2 are implicit.)

```
Definition zerop_cast1 (X:nat -> Type) (n:nat) : zerop n -> X O -> X n :=

Definition zerop_cast2 (X:nat -> Type) (n:nat) : zerop n -> X n -> X O :=

Lemma zerop_cast1_eq X (x:X O) :
zerop_cast1 X zeropl x = x.

Lemma zerop_cast2_eq X (x:X O) :
zerop_cast2 X zeropl x = x.

Lemma zerop_cast12_eq X n (A:zerop n) (x:X O) :
zerop_cast2 X A (zerop_cast1 X A x) = x.

Lemma zerop_cast21_eq X n (A:zerop n) (x:X n) :
zerop_cast1 X A (zerop_cast2 X A x) = x.
```

Exercise 8.2.3 Consider the following inductive predicate.

```
Inductive bitp : nat -> Prop :=
| bitp0 : bitp 0
| bitp1 : bitp 1.
```

Give the two reduction rules and typing rule for the match.

8.3 Equality

Equality was inductively defined as follows.

```
Inductive eq (X : Type) (x : X) : X-> Prop :=
| eq_refl : eq x x.
```

Following the rule for *zerop*, we obtain the following typing rule for matches at *eq*. (We make the first argument of eq explicit while describing the theory.)

$$\frac{s : eq\ s_1\ s_2\ s_3 \qquad z : s_1 \Rightarrow t : U \qquad u : t^z_{s_2}}{\text{match } s \text{ in } eq\ \_\ \_\ z \text{ return } t \text{ with } eq\_refl \Rightarrow u \text{ end } : t^z_{s_3}}$$

Like zerop the elim restriction does not apply to eq. This is why the typing rule allows $t : U$ instead of requiring $t : Prop$.

We can prove an elimination principle for equality with a straightforward match.

```
Lemma eq_E (X : Type) (x y : X) (p : X -> Prop) :
eq x y -> p y -> p x.
Proof. exact (fun e =>
match e in eq _ z return p z -> p x with
| eq_refl => fun A : p x => A
end). Qed.
```

This elimination principle can also be proven with a very short proof script.

```
Lemma eq_E' (X : Type) (x y : X) (p : X -> Prop) :
eq x y -> p y -> p x.

Proof. intros [] A. exact A. Qed.
```

Check that the proof script constructs exactly the proof term we give above for *eq_E*. Also make sure that you understand why *eq_E* justifies the *rewrite* tactic.

Although we were able to prove decidability of *zerop n* for all n : *nat*, we cannot prove decidability of $eq\,X\,x\,y$ for all X : *Type*, $x : X$ and $y : X$. However, it is possible to prove decidability of *eq nat* x y for x : *nat* and y : *nat*.

Exercise 8.3.1 Use the boolean equality test eq_nat on nat to prove *eq nat* is decidable.

Exercise 8.3.2 Prove each of the following lemmas not using other lemmas. Give scripts and proof terms.

```
Lemma ex_eq_ref (X : Type) (x : X) :  eq x x.
Lemma ex_eq_sym (X : Type) (x y : X) :  eq x y -> eq y x.
Lemma ex_eq_trans (X : Type) (x y z : X) :  eq x y -> eq y z -> eq x z.
Lemma ex_eq_f_equal (X Y : Type) (f : X -> Y) (x y : X) : eq x y -> eq (f x) (f y).
Lemma ex_eq_rewrite_R (X : Type) (x y : X) (p : X -> Prop) : eq x y -> p x -> p y.
```

Exercise 8.3.3 Consider the following inductive definition of equality where the inductive predicate *eq2* takes two proper arguments.

```
Inductive eq2 (X : Type) : X -> X-> Prop :=
| eq2_I : forall  x :  X, eq2 x x.
```

a) Given the typing rule for matches at *eq2*.

b) Prove the following elimination principle for *eq2*. Give a script and a proof term.

```
Lemma eq2_E (X : Type) (x y : X) (p : X -> Prop) :  eq2 x y -> p y -> p x.
```

c) Explain why the following term is ill-typed.

```
fun (X : Type) (x y : X) (p : X -> Type) (A:eq2 x y) =>
 match A in eq2 w z return p z -> p w with eq2_I z => fun B => B end
```

8.4 Even Numbers

There are many possibilities for specifying the even numbers. For instance, one may define a boolean predicate $evenb : nat \to bool$ as in Exercise 2.3.1. Another possibility characterizes the even numbers inductively:

1. 0 is an even number.
2. If n is an even number, then $S(S\,n)$ is an even number.
3. There are no other even numbers.

The inductive characterization can be expressed as an inductive definition in Coq.

```
Inductive even : nat -> Prop :=
| evenO : even O
| evenS : forall n, even n -> even (S (S n)).
```

The first argument of *evenS* is implicit in Coq. When describing the theory we will explicitly write the first argument of *evenS*. When considering examples, we will omit the first argument and display things as in Coq.

Since *even* is recursive, there will be both a match and a fix construct.

Here is an example of a match for *even*.

```
Check
fun (p : nat -> Prop) u v n (s : even n) =>
match s in even z return p z with
| evenO => u
| evenS x y => v x y
end.
% forall p : nat -> Prop,
% p O ->
% (forall x : nat, even x -> p (S (S x))) ->
% forall n : nat, even n -> p n
```

As with *zerop*, *z* is a local variable connecting the proper argument of *even* with the return type.

In the previous inductive predicates the match above proves an elimination principle. Since even is recursive, we obtain a stronger elimination principle using a combination of fix and match. In this case, it makes sense to call the elimination principle an induction principle for even.

```
Lemma even_ind' :
forall p:nat -> Prop,
p 0 ->
(forall n, even n -> p n -> p (S (S n))) ->
forall n, even n -> p n.
Proof.
exact (fun (p : nat -> Prop) u v =>
fix f n (s : even n) :=
match s in even z return p z with
| evenO => u
| evenS x y => v x y (f x y)
end).
Qed.
```

In fact, this induction principle is automatically proven under the name even_ind by Coq when the inductive predicate even is defined. The induction tactic makes use of even_ind. Note that the type of the recursive function f has type

```
forall n : nat, even n -> p n
```

and that the fix recurses on its second argument. This is the first time we use a recursive abstraction with more than one argument that cannot be reduced to a recursive abstraction with a single argument (since the type of the second argument depends on the first argument). We will study inductive proofs for even in the next section.

Here are the reduction rules for match for *even*.

$$\mathsf{match}\ \mathit{evenO}\ \mathsf{in}\ \mathit{even}\ z\ \mathsf{return}\ t\ \mathsf{with}\ \mathit{evenO} \Rightarrow u \mid \mathit{evenS}\ x\ y \Rightarrow v\ \mathsf{end} \rhd_1 u$$

$$\mathsf{match}\ \mathit{evenS}\ s_1\ s_2 \mathsf{in}\ \mathit{even}\ z\ \mathsf{return}\ t\ \mathsf{with}\ \mathit{evenO} \Rightarrow u \mid \mathit{evenS}\ x\ y \Rightarrow v\ \mathsf{end} \rhd_1 (v^{x}_{s_1})^{y}_{s_2}$$

Here is the typing rule for matches for *even*.

$$\frac{\Gamma \Rightarrow s : \mathit{even}\ s_1 \qquad \Gamma, z : \mathit{nat} \Rightarrow t : \mathit{Prop} \qquad \Gamma \Rightarrow u : t^{z}_{0} \qquad \Gamma, x : \mathit{nat},\ y : \mathit{even}\ x \Rightarrow v : t^{z}_{S(Sx)}}{\Gamma \Rightarrow \mathsf{match}\ s\ \mathsf{in}\ \mathit{even}\ z\ \mathsf{return}\ t\ \mathsf{with}\ \mathit{evenO} \Rightarrow u \mid \mathit{evenS}\ x\ y \Rightarrow v\ \mathsf{end} : t^{z}_{s_1}}$$

Unlike the cases of *zerop* and *eq*, the elim restriction does apply to *even* and so the type t must be in *Prop*. We omit the reduction and typing rules for fix.

The most important constraint on the design of the typing rules for matches is the requirement that the reduction rules for matches must be type preserving. For the above rule the argument goes as follows.

1. Suppose $s = \mathit{evenO}$. Then $s : \mathit{even}\ 0$. Since we also have $s : \mathit{even}\ s_1$, we have $0 \approx s_1$. Thus $t^{z}_{0} \approx t^{z}_{s_1}$.
2. Suppose $s = \mathit{evenS}\ w_1\ w_2$. Then $s : \mathit{even}\ (S\ (S\ w_1))$. Since we also have $s : \mathit{even}\ s_1$, we have $S\ (S\ w_1) \approx s_1$. Thus $t^{z}_{S\ (S\ w_1)} \approx t^{z}_{s_1}$.

With a dependent match for *even* we can prove the following lemma.

```
Lemma even_pred n :
even n -> even (pred (pred n)).

Proof. intros [|n' A]. exact evenO. exact A. Qed.

Print even_pred.
% fun (n : nat) (H : even n) =>
% match H in (even z) return (even (pred (pred z))) with
% | evenO => evenO
% | evenS _ A => A
% end
```

The member constructors of *even* provide for proofs of the inductive propositions obtained with *even*. Here is an example.

```
Lemma even4 :
even 4.
Proof. apply evenS. apply evenS. apply evenO. Qed.
```

The print command reveals the proof term we have constructed.

```
Print even4.
% evenS (evenS evenO)
```

When proving facts about inductively defined predicates it is sometimes tedious to remember the names of all the constructors. The tactic *constructor* is a convenience that for an inductive claim tries to apply one of the associated member constructors. We can use *constructor* to construct the same proof that 4 is even without referencing the member constructor names *evenO* or *evenS*.

```
Lemma even4 :
even 4.
Proof. constructor. constructor. constructor. Qed.
```

As we have seen in the case of zerop, the *inversion* tactic can be used to simplify proofs that would otherwise require linearization via remember or assert.

```
Lemma even_inversion_1 :
~ even 1.
Proof. intros A. inversion A. Qed.

Lemma even_inversion_SS n :
even (S (S n)) -> even n.
Proof. intros A. inversion A as [|n' A']. exact A'. Qed.
```

Exercise 8.4.1 Prove the following without using inversion or any lemmas. (Hint: Use remember.)

```
Goal ~ even 1.
Goal forall n, even (S (S n)) -> even n.
```

Exercise 8.4.2 Prove the following goals using inversion.

```
Goal ~ even 3.
Goal forall n, even (4+n) -> even n.
```

8.5 Induction on Even

As noted earlier, Coq automatically proves an induction principle

```
Lemma even_ind :
forall  p:nat -> Prop,
p 0 ->
(forall  n, even n -> p n -> p (S (S n))) ->
forall  n, even n -> p n.
```

when even is defined. This induction principle is used when the induction tactic is applied to an assumption of the form even s. As was the case with destruct, it is usually a good idea to linearize the arguments of even (e.g., using remember) before applying induction. Since the elements of inductive predicates such as even n are proofs, we will sometimes refer to such inductions as inductions on proof terms.

Here is a simple example of a proof using the induction principle for even by applying even_ind.

```
Lemma even_notS n :
even n -> ~even (S n).

Proof. revert n. apply even_ind.
intros A. inversion A.
intros n A B C. inversion C. tauto. Qed.
```

Here is the same example using the induction tactic.

```
Lemma even_notS' n :
even n -> ~even (S n).

Proof. intros A. induction A. intros B. inversion B.
intros B. inversion B. tauto. Qed.
```

Performing an induction with the tactic *induction* is a complex affair comprising three main steps:

1. Move assumptions of the goal to the claim so that the induction principle of the relevant type constructor becomes applicable (can be done with *revert*).
2. Apply the induction principle.
3. For each subgoal obtained, do introduction steps so that the claim of the subgoal corresponds to the initial claim.

Performing a case analysis with *destruct* is like performing an induction with *induction* except that the destructuring principle of the type constructor is applied in place of the induction principle. The destructuring principle can be obtained from the induction principle by omitting the inductive hypotheses, and vice versa the induction principle can be obtained from the destructuring principle by adding the inductive hypotheses.

Decidability

We next prove even is decidable. Unlike the case of zerop, we will not prove this by giving an equivalent boolean test. (This we leave as an exercise.) Instead we prove the result by induction on the proper argument n in the predicate even n. First we prove the following lemma by induction on n.

```
Lemma even_nSn n :
(~even n -> even (S n)) /\ (~even (S n) -> even n).

Proof. induction n.
split. intros A. contradiction (A evenO). intros. constructor.
split. intros A. apply evenS. apply IHn. exact A.
intros A. apply IHn. intros B. apply A. constructor. exact B.
Qed.
```

Using even_notS and even_nSn we can prove even is decidable.

```
Lemma even_dec n : even n \/ ~even n.

Proof. induction n. left. constructor.
destruct IHn as [A|A].
right. apply even_notS. exact A.
left. apply even_nSn. intros B. inversion B. contradiction (A H0).
Qed.
```

Exercise 8.5.1 Let evenb be the following boolean test.

```
Fixpoint evenb (n : nat) : bool :=
match n with
| 0 => true
| S n => negb (evenb n)
end.
```

Prove the following equivalence.

```
Lemma evenib n : even n <-> evenb n.
```

Exercise 8.5.2 Prove the following lemmas by induction on proof terms.

```
Lemma even_sum m n : even m -> even n -> even (m+n).
Lemma even_sum' m n : even (m+n) -> even m -> even n.
```

Exercise 8.5.3 Here is an impredicative definition of evenness.

```
Definition evenp (n : nat) : Prop :=
forall  p :  nat −> Prop,
p 0 −> (forall n, p n −> p (S (S n))) −> p n.
```

Prove that the impredicative definition of evenness agrees with the inductive definition.

```
Lemma evenip n :  even n <−> evenp n.
```

8.6 Natural Order

With the command

```
Locate "<=".
```

we find out that Coq realizes the order "≤" on *nat* with the predicate *le*. With the command

```
Print le.
```

we find out that Coq defines *le* as an inductive predicate as follows.

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

With can depict this definition with 2 inference rules.

$$\frac{}{n \leq n} \qquad\qquad \frac{n \leq m}{n \leq Sm}$$

Note that *le* has one parameter and one proper argument. The induction principle for *le* (which can be proven using fix and match as usual) is as follows.

```
Check le_ind.
%  le_ind
%   : forall (n : nat) (P : nat -> Prop),
%   P n ->
%   (forall m : nat, n <= m -> P m -> P (S m)) ->
%   forall m : nat, n <= m -> P m
```

We prove two properties of the predicate *le* using induction on *le*.

```
Lemma le_trans (x y z : nat) : le x y −> le y z −> le x z.

Proof. intros A B. induction B. exact A. constructor. assumption. Qed.

Lemma le_S (x y : nat) : le x y −> le (S x) (S y).
```

```
Proof. intros A. induction A; constructor. assumption. Qed.
```

It is also sometimes useful to do induction on the variable in the proper argument position.

```
Lemma le_O (y : nat) : le O y.

Proof. induction y ; constructor. exact IHy. Qed.
```

Here is an example of a proof using inversion on le.

```
Lemma le_SO (x : nat) : ~ le (S x) 0.

Proof. intros A. inversion A. Qed.
```

One can prove le is decidable by showing it is equivalent to a boolean test. We leave this as an exercise.

Exercise 8.6.1 Let leb be the following boolean test.

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

Prove the following facts in order to prove leb is equivalent to le and that le is decidable.

```
Lemma leb_S (x y : nat) :
leb x y -> leb x (S y).

Lemma leb_refl ( x : nat) :
leb x x.

Lemma leb_le (x y : nat) :
le x y -> leb x y.

Lemma le_leb (x y : nat) :
leb x y -> le x y.

Lemma le_leb_e (x y : nat) :
le x y <-> leb x y.

Lemma le_dec (x y : nat) :
le x y \/ ~le x y.
```

Exercise 8.6.2 Prove the following claims.

```
Lemma le_Sleft1 (x y : nat) :
le (S x) y -> le x y.
```

```
Lemma le_Sleft (x y : nat) :
le (S x) (S y) -> le x y.

Lemma le_antisym (x y : nat) :
le x y -> le y x -> x = y.

Goal forall x, le x 0 -> x = 0.

Goal forall x, ~ le (S x) x.
```

8.7 Remarks

In this chapter we have learned about inductive predicates with proper arguments. Inductive predicates with proper arguments can be used to define equality, evenness and $\leq$. In future chapters we will make extensive use of inductive predicates with proper arguments. The dependence on proper arguments means that return types for matches have a new dependency given using the in keyword.

We also learned about some new Coq tactics: remember, inversion and constructor.

- remember can be used to linearize the proper arguments of an inductive predicate. Linearizing the proper arguments is often necessary before applying destruct or induction.
- inversion is similar to destruct except that it uses injection and discriminate in order to derive useful equations.
- constructor can be used when one of the constructors of an inductive predicate can be applied to the claim of the current goal. In principle, one can always use apply c instead of constructor where c is the name of the particular constructor which applies. An advantage of using constructor instead of apply c is that one does not need to refer to the particular name c.

We also learned more about the induction tactic by doing induction on the proofs of inductive predicates (induction on proof terms).

9 Proof Systems for Propositional Logic

In this chapter we consider proof systems for a small, decidable fragment of the language of Coq. The fragment corresponds (more or less) to the propositions that the Coq tactic tauto can prove.

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.

```
Definition Not (s : form) : form :=
Imp s Fal.
```

We will first consider a natural deduction style proof system which corresponds closely to the proof system in Coq. We will then consider a Hilbert style proof system and prove the equivalence of the two systems.

We next turn our attention to a classical propositional logic by giving a Hilbert 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.

9.1 Natural Deduction System

Deduction rules for logical operators were given in Figure 3.1. 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

$$\mathsf{A}_{\mathcal{N}}\ \frac{}{\Gamma, s \Rightarrow s} \qquad \mathsf{W}_{\mathcal{N}}\ \frac{\Gamma \Rightarrow t}{\Gamma, s \Rightarrow t} \qquad \mathsf{I}^{\rightarrow}_{\mathcal{N}}\ \frac{\Gamma, s \Rightarrow t}{\Gamma \Rightarrow s \to t} \qquad \mathsf{E}^{\rightarrow}_{\mathcal{N}}\ \frac{\Gamma \Rightarrow s \to t \qquad \Gamma \Rightarrow s}{\Gamma \Rightarrow t}$$

$$\mathsf{E}^{\bot}_{\mathcal{N}}\ \frac{\Gamma \Rightarrow \bot}{\Gamma \Rightarrow s}$$

Figure 9.1: Natural Deduction Rules

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 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 first on the list. In order to handle assumptions that are in the tail of the list, we also include a **weakening rule** in which the context of the premise is the tail of the context of the conclusion of the rule. These rules are given in Figure 9.1.

We can represent this natural deduction system in Coq as an inductive predicate nd with two proper arguments. An inductive proposition nd G s corresponds to the sequent $\Gamma \Rightarrow s$, and the inductive proposition nd G s is inhabited if and only if the sequent $\Gamma \Rightarrow s$ is derivable with the rules in Figure 9.1. When $\Gamma \Rightarrow s$ is derivable, we write $\vdash_{\mathcal{N}} \Gamma \Rightarrow s$ or (for brevity) $\Gamma \vdash_{\mathcal{N}} s$. The reason the formula s must be a proper argument to nd is that many of the rules change the formula in the sequent $\Gamma \Rightarrow s$ when passing from a premise to the conclusion. The reason the context must be a proper argument is that the context changes in the weakening $\mathsf{W}_{\mathcal{N}}$ and implication introduction $\mathsf{I}^{\rightarrow}_{\mathcal{N}}$ rules.

```
Definition context := list form.

Inductive nd : context -> form -> Prop :=
| ndA G s :
  nd (s::G) s
| ndW G s t :
  nd G s -> nd (t::G) s
| ndII G s t :
  nd (s::G) t -> nd G (Imp s t)
| ndIE G s t :
  nd G (Imp s t) -> nd G s -> nd G t
| ndE G s :
  nd G Fal -> nd G s.
```

In order to investigate the strength of this proof system, we consider a few special classes of formulas.

- A *K*-**formula** is a formula of the form $s \to t \to s$.
- An *S*-**formula** is a formula of the form $(s \to t \to u) \to (s \to t) \to s \to u$.
- An **explosion formula** is a formula of the form $\bot \to s$
- A **double negation formula** is a formula of the form $\neg\neg s \to s$.

In Coq we can define functions producing such formulas.

```
Definition FK (s t : form) : form :=
Imp s (Imp t s).

Definition FS (s t u : form) : form :=
(Imp (Imp s (Imp t u))
(Imp (Imp s t)
(Imp s u))).

Definition FE (s : form) : form :=
Imp Fal s.

Definition FDN (s : form) : form :=
Imp (Not (Not s)) s.
```

Which of these kinds of formulas can be proven in a context Γ? It is easy to prove $\Gamma \vdash_{\mathcal{N}} s$ whenever s is a K-formula, S-formula or explosion formula. In Coq we can prove $\Gamma \vdash_{\mathcal{N}} s \to t \to s$ as follows.

```
Lemma ndK G s t :
nd G (FK s t).
```

```
Proof. apply ndII, ndII, ndW, ndA. Qed.
```

We can also display the proof as the following derivation.

$$\dfrac{\dfrac{\dfrac{\mathsf{A}_{\mathcal{N}}\ \dfrac{}{\Gamma, s \vdash_{\mathcal{N}} s}}{\Gamma, s, t \vdash_{\mathcal{N}} s}\,\mathsf{W}_{\mathcal{N}}}{\Gamma, s \vdash_{\mathcal{N}} t \to s}\,\mathsf{I}^{\to}_{\mathcal{N}}}{\Gamma \vdash_{\mathcal{N}} s \to t \to s}\,\mathsf{I}^{\to}_{\mathcal{N}}$$

We can similarly display the proof that $\Gamma \vdash_{\mathcal{N}} \bot \to s$ as follows.

$$\dfrac{\dfrac{\mathsf{A}_{\mathcal{N}}\ \dfrac{}{\Gamma, \bot \vdash_{\mathcal{N}} \bot}}{\Gamma, \bot \vdash_{\mathcal{N}} s}\,\mathsf{E}^{\bot}_{\mathcal{N}}}{\Gamma \vdash_{\mathcal{N}} \bot \to s}\,\mathsf{I}^{\to}_{\mathcal{N}}$$

Here is the corresponding Coq proof.

```
Lemma ndE' G s :
nd G (FE s).

Proof. apply ndII, ndE, ndA. Qed.
```

Hence we know $\Gamma \vdash_{\mathcal{N}} s$ whenever s is a K-formula or an explosion formula.

We next turn to S-formulas. Let Γ' be $\Gamma, (s \to t \to u), s \to t, s$. We know $\Gamma \vdash_{\mathcal{N}} (s \to t \to u) \to (s \to t) \to s \to u$ follows from $\Gamma' \vdash_{\mathcal{N}} u$ using $\mathsf{I}^{\to}_{\mathcal{N}}$ three times. Here is a derivation of $\Gamma' \vdash_{\mathcal{N}} t \to u$.

$$\mathsf{E}^{\to}_{\mathcal{N}}\ \dfrac{\mathsf{W}_{\mathcal{N}}\ \dfrac{\mathsf{A}_{\mathcal{N}}\ \dfrac{}{\Gamma, s \to t \to u \vdash_{\mathcal{N}} s \to t \to u}}{\Gamma, s \to t \to u, s \to t \vdash_{\mathcal{N}} s \to t \to u}}{\Gamma' \vdash_{\mathcal{N}} s \to t \to u} \qquad \mathsf{A}_{\mathcal{N}}\ \dfrac{}{\Gamma' \vdash_{\mathcal{N}} s}}{\Gamma' \vdash_{\mathcal{N}} t \to u}$$

Here is a derivation of $\Gamma' \vdash_{\mathcal{N}} t$.

$$\mathsf{E}^{\to}_{\mathcal{N}}\ \dfrac{\mathsf{W}_{\mathcal{N}}\ \dfrac{\mathsf{A}_{\mathcal{N}}\ \dfrac{}{\Gamma, s \to t \to u, s \to t \vdash_{\mathcal{N}} s \to t}}{\Gamma' \vdash_{\mathcal{N}} s \to t} \qquad \mathsf{A}_{\mathcal{N}}\ \dfrac{}{\Gamma' \vdash_{\mathcal{N}} s}}{\Gamma' \vdash_{\mathcal{N}} t}$$

Combining these two facts with $\mathsf{E}^{\to}_{\mathcal{N}}$ we have $\Gamma' \vdash_{\mathcal{N}} u$ as desired. This is the first derivation using the $\mathsf{E}^{\to}_{\mathcal{N}}$ rule. When we apply the $\mathsf{E}^{\to}_{\mathcal{N}}$ rule (using the constructor ndIE) in Coq we must give the left hand side s of the implication $s \to t$. We do this using a with. Here is the proof in Coq.

```
Lemma ndS G s t u :
nd G (FS s t u).

Proof. apply ndII, ndII, ndII.
apply ndIE with (s:=t). apply ndIE with (s:=s).
now apply ndW, ndW, ndA. now apply ndA.
apply ndIE with (s:=s). now apply ndW, ndA. apply ndA. Qed.
```

On the other hand, we do not generally have $\Gamma \vdash_{\mathcal{N}} \neg\neg s \to s$. If we did have this, the logic would be classical. We will consider a classical propositional logic later in this chapter.

Given the management of assumptions in the natural deduction system it is clear that we have $\Gamma \vdash_{\mathcal{N}} s$ whenever s is in the list Γ. We write $s \in \Gamma$ to mean s is in the list Γ and define this in Coq as an inductive predicate mem.

```
Inductive mem (X : Type) (x : X) : list X -> Prop :=
| memH xs : mem x (x::xs)
| memT y xs : mem x xs -> mem x (y::xs).
```

We can now prove the following result.

Lemma 9.1.1 If $s \in \Gamma$, then $\Gamma \vdash_{\mathcal{N}} s$.

Proof We argue by induction on the proof of $s \in \Gamma$. There are two cases. In the first case we must prove $\Gamma, s \vdash_{\mathcal{N}} s$. This follows precisely from $\mathsf{A}_{\mathcal{N}}$. In the inductive case we have $\Gamma \vdash_{\mathcal{N}} s$ by the inductive hypothesis and we must prove $\Gamma, t \vdash_{\mathcal{N}} s$. This follows from $\mathsf{W}_{\mathcal{N}}$. ∎

In Coq the result is proven as follows.

```
Lemma ndMem G s :
mem s G -> nd G s.

Proof. intros A ; induction A. apply ndA. apply ndW, IHA. Qed.
```

Exercise 9.1.2 Prove the following in Coq.

```
Goal forall s t, nd nil (Imp (Not s) (Imp s t)).
```

Exercise 9.1.3 Prove the following in Coq.

```
Lemma ndApply G s t :
nd (Imp s t :: G) s -> nd (Imp s t :: G) t.

Goal forall G s t,
nd (s :: G) t <-> nd G (Imp s t).

Goal forall G s t,
(forall G, nd G s -> nd G t) -> nd G (Imp s t).
```

9.2 Hilbert System

Our definition of nd as an inductive predicate required two proper arguments because the context was dynamic. It is natural to ask whether there is an alternative deduction system for which the context does not change. Obviously we can formulate an alternative assumption rule which allows us to derive $\Gamma \Rightarrow s$ whenever s is in the list Γ. This avoids the need for a weakening rule. The other natural deduction rule changing the context is implication introduction $\mathsf{I}^{\rightarrow}_{\mathcal{N}}$. A system in which the context Γ does not change simply cannot have a rule like $\mathsf{I}^{\rightarrow}_{\mathcal{N}}$.

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 K-formula is provable and another initial rule states that every S-formula is provable. Doing this would yield a system in which only two rules

$$\mathsf{A}_{\mathcal{H}}\ \frac{}{\Gamma \Rightarrow s}\ s \in \Gamma \qquad \mathsf{K}_{\mathcal{H}}\ \frac{}{\Gamma \Rightarrow s \to t \to s}$$

$$\mathsf{S}_{\mathcal{H}}\ \frac{}{\Gamma \Rightarrow (s \to t \to u) \to (s \to t) \to s \to u} \qquad \mathsf{E}_{\mathcal{H}}\ \frac{}{\Gamma \Rightarrow \bot \to u}$$

$$\mathsf{MP}_{\mathcal{H}}\ \frac{\Gamma \Rightarrow s \to t \qquad \Gamma \Rightarrow s}{\Gamma \Rightarrow t}$$

Figure 9.2: Hilbert Rules for Intuitionistic Propositional Logic

have premises: a rule like $\mathsf{E}^{\to}_{\mathcal{N}}$ and a rule like $\mathsf{E}^{\bot}_{\mathcal{N}}$. 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 $\mathsf{E}^{\to}_{\mathcal{N}}$ since we can replace the $\mathsf{E}^{\bot}_{\mathcal{N}}$ rule with an initial rule stating that every explosion formula is provable. Such systems are called **Hilbert systems**. The rules in Figure 9.2 define our **Hilbert system for intuitionistic propositional logic**.

We can define this in Coq as an inductive predicate hil with one parameter (the context Γ) and one proper argument (the formula s). An inductive proposition hil G s corresponds to the sequent $\Gamma \Rightarrow s$, and the inductive proposition hil G s is inhabited if and only if the sequent $\Gamma \Rightarrow s$ is derivable with the rules in Figure 9.2. When $\Gamma \Rightarrow s$ is derivable, we write $\vdash_{\mathcal{H}} \Gamma \Rightarrow s$ or (for brevity) $\Gamma \vdash_{\mathcal{H}} s$.

```
Inductive hil (G : context) : form -> Prop :=
| hilA s :
  mem s G -> hil G s
| hilK s t :
  hil G (FK s t)
| hilS s t u :
  hil G (FS s t u)
| hilE s :
  hil G (FE s)
| hilMP s t :
  hil G (Imp s t) -> hil G s -> hil G t.
```

Using the lemmas from the previous section, we can easily prove by induction (on proof terms) that if $\Gamma \vdash_{\mathcal{H}} s$, then $\Gamma \vdash_{\mathcal{N}} s$.

Lemma 9.2.1 If $\Gamma \vdash_{\mathcal{H}} s$, then $\Gamma \vdash_{\mathcal{N}} s$.

Proof We argue by induction on the proof of $\Gamma \vdash_{\mathcal{H}} s$. We must argue a case for each rule in Figure 9.2. If s is a K-formula, an S-formula or an explosion formula,

then we have already proven $\Gamma \vdash_{\mathcal{N}} s$ in the previous section. If $s \in \Gamma$, then we know $\Gamma \vdash_{\mathcal{N}} s$ by Lemma 9.1.1. Finally, we consider the modus ponens case. Assume $\Gamma \vdash_{\mathcal{H}} s \to t$ and $\Gamma \vdash_{\mathcal{H}} s$. The inductive hypotheses yield $\Gamma \vdash_{\mathcal{N}} s \to t$ and $\Gamma \vdash_{\mathcal{N}} s$. We conclude $\Gamma \vdash_{\mathcal{N}} t$ using $\mathsf{E}^{\to}_{\mathcal{N}}$. ∎

We can also prove this result in Coq.

```
Lemma agree_nd G s :
hil  G s -> nd G s.

Proof. intros A. induction A.
(* hilA *) apply ndMem ; assumption.
(* hilK *) now apply ndK.
(* hilS *) now apply ndS.
(* hilDN *) now apply ndE'.
(* hilMP *) apply ndIE with (s:=s) ; assumption. Qed.
```

We should also be able to prove that if $\Gamma \vdash_{\mathcal{N}} s$, then $\Gamma \vdash_{\mathcal{H}} s$. If one tries to argue by induction, one will get stuck at the weakening and implication introduction rules. This can be remedied with two lemmas. First, we prove that provability in the Hilbert system respects weakening the context with one new assumption. That is, if $\Gamma \vdash_{\mathcal{H}} s$, then $\Gamma, t \vdash_{\mathcal{H}} s$.

Lemma 9.2.2 If $\Gamma \vdash_{\mathcal{H}} s$, then $\Gamma, t \vdash_{\mathcal{H}} s$.

Proof We argue by induction on the proof of $\Gamma \vdash_{\mathcal{H}} s$. If s is a K-formula, an S-formula, an explosion formula or in Γ, then we know $\Gamma, t \vdash_{\mathcal{N}} s$. Assume $\Gamma \vdash_{\mathcal{H}} s \to u$ and $\Gamma \vdash_{\mathcal{H}} s$. By the inductive hypotheses we know $\Gamma, t \vdash_{\mathcal{H}} s \to u$ and $\Gamma, t \vdash_{\mathcal{H}} s$. We conclude $\Gamma, t \vdash_{\mathcal{H}} u$ using $\mathsf{MP}_{\mathcal{H}}$. ∎

Here is the easy induction proof in Coq.

```
Lemma hilW G s t :
hil  G s -> hil (t::G) s.

Proof. intros A ; induction A.
now apply hilA, memT, H.
now apply hilK.
now apply hilS.
now apply hilE.
exact (hilMP IHA1 IHA2). Qed.
```

In order to handle the implication introduction rule, we need to prove that if $\Gamma, s \vdash_{\mathcal{H}} t$, then $\Gamma \vdash_{\mathcal{H}} s \to t$. This result is known as the **deduction theorem**.

Theorem 9.2.3 (Deduction Theorem) If $\Gamma, s \vdash_{\mathcal{H}} t$, then $\Gamma \vdash_{\mathcal{H}} s \to t$.

Proof We prove this by induction on the proof of $\Gamma, s \vdash_{\mathcal{H}} t$. There are three possible cases to consider.

- Suppose $t \in \Gamma$, t is a K-formula, t is an S-formula or t is an explosion formula. In any of these cases $\Gamma \vdash_{\mathcal{H}} t$ and $\Gamma \vdash_{\mathcal{H}} t \to s \to t$. Hence $\Gamma \vdash_{\mathcal{H}} s \to t$.
- Suppose t is s. We need to prove $\Gamma \vdash_{\mathcal{H}} s \to s$. This follows from the fact that $(s \to (s \to s) \to s) \to (s \to s \to s) \to s \to s$ is an S-formula while $s \to (s \to s) \to s$ and $s \to s \to s$ are K-formulas.
- Suppose $\Gamma, s \vdash_{\mathcal{H}} u \to t$ and $\Gamma, s \vdash_{\mathcal{H}} u$. By the inductive hypothesis $\Gamma \vdash_{\mathcal{H}} s \to u \to t$ and $\Gamma \vdash_{\mathcal{H}} s \to u$. In order to see that $\Gamma \vdash_{\mathcal{H}} s \to t$ it suffices to note that $(s \to u \to t) \to (s \to u) \to s \to t$ is an S-formula. ∎

We state the deduction theorem in Coq and leave its Coq proof as an exercise.

```
Lemma ded s G t :
hil (s::G) t -> hil G (Imp s t).
```

Now we can prove by an easy induction that if $\Gamma \vdash_{\mathcal{N}} t$, then $\Gamma \vdash_{\mathcal{H}} t$.

Lemma 9.2.4 If $\Gamma \vdash_{\mathcal{N}} t$, then $\Gamma \vdash_{\mathcal{H}} t$.

Proof We must argue a case for each rule in Figure 9.1. For the $\mathsf{A}_{\mathcal{N}}$ case we must prove $\Gamma, s \vdash_{\mathcal{H}} s$. We know this by $\mathsf{A}_{\mathcal{H}}$ since $s \in \Gamma, s$. For the $\mathsf{E}^{\to}_{\mathcal{N}}$ case the inductive hypotheses give $\Gamma \vdash_{\mathcal{H}} s \to t$ and $\Gamma \vdash_{\mathcal{H}} s$. We conclude $\Gamma \vdash_{\mathcal{H}} t$ using $\mathsf{MP}_{\mathcal{H}}$. For the $\mathsf{E}^{\bot}_{\mathcal{N}}$ case the inductive hypothesis gives $\Gamma \vdash_{\mathcal{H}} \bot$. We conclude $\Gamma \vdash_{\mathcal{H}} s$ using the following derivation.

$$\mathsf{MP}_{\mathcal{H}}\ \frac{\mathsf{E}_{\mathcal{H}}\ \dfrac{}{\Gamma \vdash_{\mathcal{H}} \bot \to s} \qquad \Gamma \vdash_{\mathcal{H}} \bot}{\Gamma \vdash_{\mathcal{H}} s}$$

For the $\mathsf{W}_{\mathcal{N}}$ case we use Lemma 9.2.2. For the $\mathsf{I}^{\to}_{\mathcal{N}}$ case we use the deduction theorem (Theorem 9.2.3). ∎

We can easily replay this proof in Coq.

```
Lemma agree_hil G s :
nd G s -> hil G s.

Proof. intros A ; induction A.
(* ndA *) now apply hilA, memH.
(* ndW *) apply hilW ; assumption.
(* ndII *) exact (ded IHA).
(* ndIE *) exact (hilMP IHA1 IHA2).
(* ndE *) exact (hilMP (hilE G s) IHA). Qed.
```

Combining Lemmas 9.2.1 and 9.2.4 we obtain the following theorem.

Theorem 9.2.5 $\Gamma \vdash_{\mathcal{H}} s$ if and only if $\Gamma \vdash_{\mathcal{N}} s$.

Exercise 9.2.6 Prove the following lemmas in Coq and use them to prove the deduction theorem in Coq.

```
Lemma hilAK G s t :
hil  G s -> hil G (Imp t s).

Lemma hilAS G s t u :
hil  G (Imp s (Imp t u)) -> hil G (Imp s t) -> hil G (Imp s u).

Lemma hill G s :
hil  G (Imp s s).

Lemma ded s G t :
hil  (s::G) t  -> hil G (Imp s t).
```

9.3 Admissible Rules

In addition to the rules defining $\Gamma \vdash_{\mathcal{N}} s$ and $\Gamma \vdash_{\mathcal{H}} s$, it is often useful to have other rules which do not change the relation. We call such rules **admissible**. In general a rule of the form

$$\frac{\Gamma_1 \Rightarrow s_1 \qquad \cdots \qquad \Gamma_n \Rightarrow s_n}{\Gamma \Rightarrow s}$$

is **admissible** for the natural deduction system if we know $\Gamma \vdash_{\mathcal{N}} s$ whenever we know $\Gamma_1 \vdash_{\mathcal{N}} s_1$, ..., and $\Gamma_n \vdash_{\mathcal{N}} s_n$. Since we know $\Gamma \vdash_{\mathcal{H}} s$ if and only if $\Gamma \vdash_{\mathcal{N}} s$, we know a rule is admissible for the Hilbert system if and only if it is admissible for the natural deduction system. For the rest of this section, we will simply say a rule is admissible if it is admissible for either system (i.e., both systems).

As an example, note that we have already proven that if $s \in \Gamma$, then $\Gamma \vdash_{\mathcal{N}} s$. We proved this as Lemma 9.1.1 and in Coq as the lemma named ndMem. This fact justifies admissibility of the rule

$$\mathsf{A}^{\in}\ \frac{}{\Gamma \Rightarrow s}\ s \in \Gamma$$

We are free to make use of this rule in order to demonstrate $\Gamma \vdash_{\mathcal{N}} s$.

General Weakening

We prove admissibility of the following general weakening rule for both the Hilbert system and the natural deduction system.

$$\mathsf{W}^{\subseteq}\ \frac{\Gamma \Rightarrow s}{\Gamma' \Rightarrow s}\ \Gamma \subseteq \Gamma'$$

Admissibility of $\mathsf{W}^{\subseteq}$ is a consequence of the following result.

Lemma 9.3.1 Suppose $\Gamma \subseteq \Gamma'$. If $\Gamma \vdash_{\mathcal{H}} s$, then $\Gamma' \vdash_{\mathcal{H}} s$.

Proof We prove this by an easy induction on the proof of $\Gamma \vdash_{\mathcal{H}} s$. If $s \in \Gamma$, then $s \in \Gamma'$ and we have $\Gamma' \vdash_{\mathcal{H}} s$. If s is a K-formula, an S-formula or an explosion formula, then we have $\Gamma' \vdash_{\mathcal{H}} s$. We finally consider the modus ponens case. Suppose $\Gamma \vdash_{\mathcal{H}} s \to t$ and $\Gamma \vdash_{\mathcal{H}} s$. By the inductive hypothesis we have $\Gamma' \vdash_{\mathcal{H}} s \to t$ and $\Gamma' \vdash_{\mathcal{H}} s$. By modus ponens rule we conclude $\Gamma' \vdash_{\mathcal{H}} t$. ∎

Lemma 9.3.1 can be proven in Coq as follows and then used as an admissible rule for the Hilbert system.

```
Lemma hilGW G G' s : (forall u, mem u G -> mem u G') -> hil G s -> hil G' s.
Proof. intros A B. induction B.
apply hilA. now apply A.
now apply hilK.
now apply hilS.
now apply hilE.
now apply hilMP with (s := s).
Qed.
```

In order to obtain the corresponding admissible rule for the natural deduction system, we prove the following lemma in Coq (relying on the result for the Hilbert system).

```
Lemma ndGW G G' s : (forall u, mem u G -> mem u G') -> nd G s -> nd G' s.
Proof. intros A B. apply agree_nd.
 apply hilGW with (G := G). assumption.
 now apply agree_hil. Qed.
```

Using the generalized weakening result we can easily prove the following.

```
Lemma ndW2 G s t u :
nd (t :: G) u -> nd (t :: s :: G) u.
Proof. apply ndGW. intros v A. inversion A. now apply memH.
now apply memT, memT. Qed.
```

Application

In Coq, it is common to use the apply tactic with an assumed implication. We can simulate this with the following admissible rule.

$$\mathsf{Ap}\ \frac{\Gamma \Rightarrow s}{\Gamma \Rightarrow t}\ s \to t \in \Gamma$$

The proof that this rule is admissible is easy: Assume $s \to t \in \Gamma$ and $\Gamma \vdash_{\mathcal{N}} s$. By $\mathsf{A}^{\in}$ we know $\Gamma \vdash_{\mathcal{N}} s \to t$. By $\mathsf{E}^{\rightarrow}_{\mathcal{N}}$ we conclude $\Gamma \vdash_{\mathcal{N}} t$ as desired. In Coq the lemma ndAp corresponding to the rule Ap is stated and proven as follows.

```
Lemma ndAp G s t : mem (Imp s t) G -> nd G s -> nd G t.
Proof. intros A B. apply ndIE with (s := s). apply ndMem; assumption. assumption. Qed.
```

We can apply the admissible rule Ap and use weakening to obtain a special case N.

$$\mathsf{N}\ \frac{\Gamma \Rightarrow s}{\Gamma, \neg s \Rightarrow \bot}$$

The justification of admissibility of N is expressed by the following.

$$\mathsf{Ap}\ \frac{\mathsf{W}_{\mathcal{N}}\ \dfrac{\Gamma \vdash_{\mathcal{N}} s}{\Gamma, \neg s \vdash_{\mathcal{N}} s} \qquad \Gamma, \neg s \vdash_{\mathcal{N}} \bot}{\Gamma, \neg s \vdash_{\mathcal{N}} \bot}$$

In Coq we have the following lemma.

```
Lemma ndN G s :
nd G s -> nd (Not s::G) Fal.
Proof. intros A. apply ndAp with (s := s). now apply memH. now apply ndW. Qed.
```

Refutation Cases

As a final example of an admissible rule, we note that if we want to prove $\Gamma \vdash_{\mathcal{N}} \bot$, then it is enough to prove $\bot$ in two cases: once assuming s and once assuming $\neg s$.

$$\mathsf{RC}\ \frac{\Gamma, s \Rightarrow \bot \qquad \Gamma, \neg s \Rightarrow \bot}{\Gamma \Rightarrow \bot}$$

We argue admissibility of this rule as follows. Assume $\Gamma, s \vdash_{\mathcal{N}} \bot$ and $\Gamma, \neg s \vdash_{\mathcal{N}} \bot$. Using $\mathsf{I}^{\rightarrow}_{\mathcal{N}}$ with each of these assumptions we know $\Gamma \vdash_{\mathcal{N}} \neg s$ and $\Gamma \vdash_{\mathcal{N}} \neg s \to \bot$. By $\mathsf{E}^{\rightarrow}_{\mathcal{N}}$ we conclude $\Gamma \vdash_{\mathcal{N}} \bot$. In Coq we state this rule is stated and proven as follows.

$$\mathsf{A}_{\mathcal{NC}}\ \frac{}{\Gamma, s \Rightarrow s} \qquad \mathsf{W}_{\mathcal{NC}}\ \frac{\Gamma \Rightarrow t}{\Gamma, s \Rightarrow t} \qquad \mathsf{I}\vec{}_{\mathcal{NC}}\ \frac{\Gamma, s \Rightarrow t}{\Gamma \Rightarrow s \to t} \qquad \mathsf{E}\vec{}_{\mathcal{NC}}\ \frac{\Gamma \Rightarrow s \to t \qquad \Gamma \Rightarrow s}{\Gamma \Rightarrow t}$$

$$\mathsf{C}_{\mathcal{NC}}\ \frac{\Gamma, \neg s \Rightarrow \bot}{\Gamma \Rightarrow s}$$

Figure 9.3: Classical Natural Deduction Rules

```
Lemma ndRC G s :
nd (s :: G) Fal -> nd (Not s :: G) Fal -> nd G Fal.
Proof. intros A B. apply ndIE with (s := Not s); now apply ndII. Qed.
```

9.4 Classical Propositional Logic

We now consider classical propositional logic. We can modify both the natural deduction system and the Hilbert system to be classical by replacing explosion with a different rule. In the natural deduction system we replace explosion with a rule for proof by contradiction. In the Hilbert system, we use an initial rule for double negation.

The classical natural deduction system is defined by the rules in Figure 9.3. We write $\Gamma \vdash_{\mathcal{NC}} s$ if $\Gamma \Rightarrow s$ is derivable with the rules in Figure 9.3. In Coq, the classical natural deduction system can be defined as the following inductive predicate ndc.

```
Inductive ndc : context -> form -> Prop :=
| ndcA G s :
  ndc (s::G) s
| ndcW G s t :
  ndc G s -> ndc (t::G) s
| ndcII G s t :
  ndc (s::G) t -> ndc G (Imp s t)
| ndcIE G s t :
  ndc G (Imp s t) -> ndc G s -> ndc G t
| ndcC G s :
  ndc (Not s::G) Fal -> ndc G s.
```

An easy induction proves that if $\Gamma \vdash_{\mathcal{N}} s$, then $\Gamma \vdash_{\mathcal{NC}} s$. We leave this as an exercise.

The classical Hilbert system is defined by the rules in Figure 9.4. We write $\Gamma \vdash_{\mathcal{HC}} s$ if $\Gamma \Rightarrow s$ is derivable with the rules in Figure 9.4.

$$\mathsf{A}_{\mathcal{HC}}\ \frac{}{\Gamma \Rightarrow s}\ s \in \Gamma \qquad \mathsf{K}_{\mathcal{HC}}\ \frac{}{\Gamma \Rightarrow s \to t \to s}$$

$$\mathsf{S}_{\mathcal{HC}}\ \frac{}{\Gamma \Rightarrow (s \to t \to u) \to (s \to t) \to s \to u} \qquad \mathsf{DN}_{\mathcal{HC}}\ \frac{}{\Gamma \Rightarrow \neg\neg s \to s}$$

$$\mathsf{MP}_{\mathcal{HC}}\ \frac{\Gamma \Rightarrow s \to t \qquad \Gamma \Rightarrow s}{\Gamma \Rightarrow t}$$

Figure 9.4: Hilbert Rules for Classical Propositional Logic

In Coq, the classical Hilbert system can be defined as the following inductive predicate hilc.

```
Inductive hilc (G : context) : form -> Prop :=
| hilcA s :
  mem s G -> hilc G s
| hilcK s t :
  hilc G (FK s t)
| hilcS s t u :
  hilc G (FS s t u)
| hilcDN s :
  hilc G (FDN s)
| hilcMP s t :
  hilc G (Imp s t) -> hilc G s -> hilc G t.
```

One can prove that $\Gamma \vdash_{\mathcal{HC}} s$ if and only if $\Gamma \vdash \mathcal{NC}s$ using the same methods as we used to prove Theorem 9.2.5. We leave this as an exercise.

Exercise 9.4.1 Prove that if $\Gamma \vdash_{\mathcal{N}} s$, then $\Gamma \vdash_{\mathcal{NC}} s$.

```
Lemma nd_ndc G s : nd G s -> ndc G s.
```

Exercise 9.4.2 Prove that $\Gamma \vdash_{\mathcal{NC}} s$ if and only if $\Gamma, \neg s \vdash_{\mathcal{NC}} \bot$.

```
Goal forall G s, ndc G s <-> ndc (Not s::G) Fal.
```

Exercise 9.4.3 Prove the following results and conclude $\Gamma \vdash_{\mathcal{NC}} s$ if and only if $\Gamma \vdash_{\mathcal{HC}} s$.

```
Lemma ndcMem G s :
mem s G -> ndc G s.
```

```
Lemma agree_ndc G s :
hilc G s -> ndc G s.

Lemma hilcW G s t :
hilc G s -> hilc (t :: G) s.

Lemma hilcAK G s t :
hilc G s -> hilc G (Imp t s).

Lemma hilcAS G s t u :
hilc G (Imp s (Imp t u)) -> hilc G (Imp s t) -> hilc G (Imp s u).

Lemma hilcI G s :
hilc G (Imp s s).

Lemma dedc s G t :
hilc (s :: G) t -> hilc G (Imp s t).

Lemma agree_hilc G s :
ndc G s -> hilc G s.
```

Exercise 9.4.4 Use the previous exercises to prove the following.

```
Lemma hil_hilc G s : hil G s -> hilc G s.
```

9.5 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 $\Gamma \vdash_{\mathcal{HC}} s$, then $\Gamma \vdash_{\mathcal{N}} \neg\neg s$. (We leave the other half as an exercise.) There are different ways to prove the Glivenko theorem. We prove it via the following lemma from which the Glivenko result easily follows (via $\mathsf{I}^{\rightarrow}_{\mathcal{N}}$).

Lemma 9.5.1 If $\Gamma \vdash_{\mathcal{HC}} s$, then $\Gamma, \neg s \vdash_{\mathcal{N}} \bot$.

Proof We argue by induction on the proof of $\Gamma \vdash_{\mathcal{HC}} s$. Suppose s is a K-formula, an S-formula or a member of Γ. In any of these cases $\Gamma \vdash_{\mathcal{N}} s$ and so $\Gamma, \neg s \vdash_{\mathcal{N}} \bot$ by the admissible rule N.

Next suppose s is of the form $\neg\neg t \to t$. The following derivation demon-

strates that $\Gamma, \neg(\neg\neg t \to t) \vdash_{\mathcal{N}} \bot$. Let Γ' be $\Gamma, \neg(\neg\neg t \to t)$.

$$\mathsf{RC}\ \frac{\mathsf{Ap}\ \dfrac{\mathsf{I}^{\to}_{\mathcal{N}}\ \dfrac{\mathsf{A}^{\in}\ \dfrac{}{\Gamma', t, \neg\neg t \vdash_{\mathcal{N}} t}}{\Gamma', t \vdash_{\mathcal{N}} \neg\neg t \to t}}{\Gamma', t \vdash_{\mathcal{N}} \bot} \qquad \mathsf{Ap}\ \dfrac{\mathsf{I}^{\to}_{\mathcal{N}}\ \dfrac{\mathsf{E}^{\bot}_{\mathcal{N}}\ \dfrac{\mathsf{N}\ \dfrac{\mathsf{A}_{\mathcal{N}}\ \dfrac{}{\Gamma', \neg t \vdash_{\mathcal{N}} \neg t}}{\Gamma', \neg t, \neg\neg t \vdash_{\mathcal{N}} \bot}}{\Gamma', \neg t, \neg\neg t \vdash_{\mathcal{N}} t}}{\Gamma', \neg t \vdash_{\mathcal{N}} \neg\neg t \to t}}{\Gamma', \neg t \vdash_{\mathcal{N}} \bot}}{\Gamma' \vdash_{\mathcal{N}} \bot}$$

Finally suppose $\Gamma \vdash_{\mathcal{HC}} t \to s$ and $\Gamma \vdash_{\mathcal{HC}} t$. The inductive hypotheses imply $\Gamma, \neg(t \to s) \vdash_{\mathcal{N}} \bot$ and $\Gamma, \neg t \vdash_{\mathcal{N}} \bot$. Let Γ'' be $\Gamma, \neg s, t \to s, t$. The following derivation demonstrates $\Gamma'' \vdash_{\mathcal{N}} \bot$.

$$\mathsf{Ap}\ \frac{\mathsf{E}^{\to}_{\mathcal{N}}\ \dfrac{\mathsf{A}^{\in}\ \dfrac{}{\Gamma'' \vdash_{\mathcal{N}} t \to s} \qquad \mathsf{A}_{\mathcal{N}}\ \dfrac{}{\Gamma'' \vdash_{\mathcal{N}} t}}{\Gamma'' \vdash_{\mathcal{N}} s}}{\Gamma'' \vdash_{\mathcal{N}} \bot}$$

Let Γ' be $\Gamma, \neg s$. Using the inductive hypotheses and $\Gamma'' \vdash_{\mathcal{N}} \bot$, the following derivation demonstrates $\Gamma' \vdash_{\mathcal{N}} \bot$.

$$\mathsf{RC}\ \frac{\mathsf{RC}\ \dfrac{\Gamma', t \to s, t \vdash_{\mathcal{N}} \bot \qquad \mathsf{W}^{\subseteq}\ \dfrac{\Gamma, \neg t \vdash_{\mathcal{N}} \bot}{\Gamma', t \to s, \neg t \vdash_{\mathcal{N}} \bot}}{\Gamma', t \to s \vdash_{\mathcal{N}} \bot} \qquad \mathsf{W}^{\subseteq}\ \dfrac{\Gamma, \neg(t \to s) \vdash_{\mathcal{N}} \bot}{\Gamma', \neg(t \to s) \vdash_{\mathcal{N}} \bot}}{\Gamma' \vdash_{\mathcal{N}} \bot}$$

The Glivenko result easily follows.

Theorem 9.5.2 (Glivenko) If $\Gamma \vdash_{\mathcal{HC}} s$, then $\Gamma \vdash_{\mathcal{N}} \neg\neg s$.

Proof Suppose $\Gamma \vdash_{\mathcal{HC}} s$. By Lemma 9.5.1 we know $\Gamma, \neg s \vdash_{\mathcal{N}} \bot$. By $\mathsf{I}^{\to}_{\mathcal{N}}$ we have $\Gamma \vdash_{\mathcal{N}} \neg\neg s$. ∎

We also obtain the following corollary.

Corollary 9.5.3 If $\Gamma \vdash_{\mathcal{HC}} \bot$, then $\Gamma \vdash_{\mathcal{N}} \bot$.

Proof Suppose $\Gamma \vdash_{\mathcal{HC}} \bot$. By Theorem 9.5.2 we know $\Gamma \vdash_{\mathcal{N}} \neg\neg\bot$. It is easy to prove $\Gamma \vdash_{\mathcal{N}} \neg\bot$ using $\mathsf{I}^{\to}_{\mathcal{N}}$ and $\mathsf{A}_{\mathcal{N}}$. Using $\mathsf{E}^{\to}_{\mathcal{N}}$ we conclude $\Gamma \vdash_{\mathcal{N}} \bot$. ∎

In Coq we can prove these results as follows.

```
Lemma Glivenko1 G s :
hilc G s -> nd (Not s :: G) Fal.

Proof. intros A ; induction A.
now apply ndN, ndMem.
now apply ndN, ndK.
now apply ndN, ndS.
apply ndRC with (s := s); apply ndAp with (s := FDN s); try now apply memT, memH.
now apply ndII, ndW, ndA.
now apply ndII, ndE, ndN, ndA.
apply ndRC with (s := s). apply ndRC with (s := Imp s t).
apply ndAp with (s := t). now apply memT, memT, memH.
apply ndAp with (s := s). now apply memH. now apply ndW,ndA.
now apply ndW2,ndW2.
now apply ndW2. Qed.

Lemma Glivenko G s :
hilc G s -> nd G (Not (Not s)).
Proof. intros A. apply ndII. apply Glivenko1; assumption. Qed.

Lemma Glivenko_cor G :
hilc G Fal -> nd G Fal.
Proof. intros A. apply ndIE with (s := Not Fal). apply Glivenko; assumption.
now apply ndII, ndA. Qed.
```

Another way to prove the Glivenko theorem is to first prove the following lemmas about natural deduction. Once one has these lemmas, a direct induction on the proof of $\Gamma \vdash_{\mathcal{HC}} s$ suffices.

1. $\Gamma \vdash_{\mathcal{N}} \neg\neg(\neg\neg s \to s)$.
2. If $\Gamma \vdash_{\mathcal{N}} s$, then $\Gamma \vdash_{\mathcal{N}} \neg\neg s$.
3. If $\Gamma \vdash_{\mathcal{N}} \neg\neg(s \to t)$ and $\Gamma \vdash_{\mathcal{N}} \neg\neg s$, then $\Gamma \vdash_{\mathcal{N}} \neg\neg t$.

We leave the reader to fill in the details of this proof as an exercise.

Exercise 9.5.4 Prove the following results in Coq and use them to prove Glivenko's theorem.

```
Lemma ndDNDN G s :
nd G (Not (Not (FDN s))).

Lemma ndDN' G s :
nd G s -> nd G (Not (Not s)).

Lemma ndDNMP G s t :
nd G (Not (Not (Imp s t))) -> nd G (Not (Not s)) -> nd G (Not (Not t)).
```

```
Lemma Glivenko G s :
hilc  G s -> nd G (Not (Not s)).
```

Exercise 9.5.5 Prove that if $\Gamma \vdash_{\mathcal{N}} \neg\neg s$, then $\Gamma \vdash_{\mathcal{HC}} s$.

```
Lemma Glivenko_conv G s :
nd G (Not (Not s)) -> hilc G s.
```

9.6 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 Boolean Satisfiability

We now study the two-valued interpretation of propositional formulas. We show that satisfiable formulas are not refutable. With this result it becomes easy to show that formulas are underivable in the classical systems.

10.1 Boolean Reflection

Coq's library *Bool* defines boolean versions of the logical connectives $\neg$, $\wedge$, $\vee$, and $\rightarrow$.

```
    neg x  :=  if  x then false else true
  andb x y  :=  if  x then y else false
   orb x y  :=   if  x then true else y
 implb x y  :=  if  x then y else true
```

From now on we will load the library *Bool* with the command

```
Require Import Bool.
```

This provides us with the infix operators && and || for *andb* and *orb*.

Recall that we use the coercion

```
Coercion bool2Prop (x : bool) := if x then True else False.
```

which embeds booleans into propositions. It turns out that on boolean arguments the logical connectives behave exactly as the boolean connectives.

```
Lemma negb_ref (x : bool) :  negb x <-> ~x.
```

```
Lemma andb_ref x y :  x && y <-> x /\ y.
```

```
Lemma orb_ref x y :  x || y <-> x \/ y.
```

```
Lemma implb_ref x y :  implb x y <-> x -> y.
```

Proving these **reflection lemmas** is straightforward, the following script does the job in each case:

```
Proof. destruct x ; simpl ; tauto. Qed.
```

Exercise 10.1.1 Boolean claims can be shown by contradiction. Prove the following lemma.

```
Lemma bcontra (x : bool) :  ~ (negb x) -> x.
```

Exercise 10.1.2 Boolean implication is best understood as boolean disjunction. Prove the following lemma.

```
Lemma implb_orb x y :
implb x y = negb x || y.
```

10.2 Rewriting with Logical Equivalences

Recall that logical equivalence (*iff* : *Prop* → *Prop* → *Prop*) is reflexive, symmetric, and transitive.

$$\frac{}{A \leftrightarrow A} \qquad \frac{A \leftrightarrow B}{B \leftrightarrow A} \qquad \frac{A \leftrightarrow B \qquad B \leftrightarrow C}{A \leftrightarrow C}$$

The following rules state that conjunction, disjunction, and implication are **compatible** with equivalence.

$$\frac{A \leftrightarrow A' \qquad B \leftrightarrow B'}{A \wedge B \leftrightarrow A' \wedge B'} \qquad \frac{A \leftrightarrow A' \qquad B \leftrightarrow B'}{A \vee B \leftrightarrow A' \vee B'} \qquad \frac{A \leftrightarrow A' \qquad B \leftrightarrow B'}{A \to B \leftrightarrow A' \to B'}$$

Taken together, these facts justify rewriting with equivalences below equivalences, conjunctions, disjunctions, and implications. Coq can rewrite with equivalences once we load the library *Setoid*.

```
Require Import Setoid.

Goal forall A B,
implb (A||B) (A&&B) <-> A \/ B -> A /\ B.

Proof. intros A B. rewrite implb_ref. rewrite orb_ref. rewrite andb_ref. reflexivity. Qed.
```

Step carefully through the proof and observe what happens. Say for each rewriting step which of the above rules are needed to justify it. The proof script may be written more concisely as follows:

```
Proof. intros. rewrite implb_ref, orb_ref, andb_ref. reflexivity. Qed.
```

Exercise 10.2.1 Prove the following goal with the tactics *intros*, *rewrite*, and *reflexivity*.

```
Goal forall A A' B B',   (A <-> A') -> (B <-> B') -> (A -> B <-> A' -> B').
```

10.3 Boolean Sums and Omega

Coq defines so-called **boolean sums** as follows.

```
Inductive sumbool (X Y : Prop) : Type :=
| left : X -> sumbool X Y
| right : Y -> sumbool X Y.
```

Since boolean sums play an important role in the library, Coq also defines a fancy notation for boolean sums:

```
{s} + {t} := sumbool s t
```

A boolean sum is like a disjunction but since it is not a proposition the elim restriction does not apply. Thus we can do unrestricted case analysis on boolean sums. An example will follow shortly.

From now on we will require Coq's library *Omega*, which provides functions, lemmas, and the automation tactic *omega* for natural numbers. A useful function *Omega* provides is

```
eq_nat_dec : forall x y : nat, {x = y} + {x <> y}
```

From the type alone it is clear that *eq_nat_dec* decides equality on *nat*. Given two numbers x and y, *eq_nat_dec* must return either a labeled proof of the equation $x=y$ or a labeled proof of the disequation $x \neq y$. Thus the type of *eq_nat_dec* fully specifies the behavior of *eq_nat_dec*, at least if we ignore the form of the proofs included in the results.

Since *eq_nat_dec* yields a boolean sum rather than a disjunction, we can freely compute with it.

```
Compute match eq_nat_dec 2 3 with left _ => true | right _ => false end.
% false
```

Coq's if-then-else notation applies to all types with two constructors, where we end up in the "then" branch if the first argument is obtained with the first constructor and in the "else" branch otherwise.

```
Compute if eq_nat_dec 3 3 then true else false.
% false
```

We may see the values of boolean sums as informative booleans carrying proofs with them. Using the if-then-else notation we can forget the proofs and just exploit the boolean information.

The automation tactic *omega* can construct many proofs that depend on addition and the linear order on numbers. Here are a few examples.[1]

```
Goal forall x, S x <> x.
```

[1] From now on we will use Coq's predefined order for *nat*.

```
Goal forall x y z, x + y + z = z + 2*x + y - x.
Goal forall x x' y , x < x' -> x + y < y + x'.
```

Each of the goals can be solved with the script

```
Proof. intros. omega. Qed.
```

10.4 List Membership

From now on we require Coq's library *List*. *List* provides the **cons notation**

```
s :: t := cons s t
```

and defines list membership as follows.

```
Fixpoint In (X : Type) (x : X) (A : list X) : Prop :=
match A with
| nil => False
| y :: A' => y = x \/ In x A'
end.
```

In addition to the cons notation we define the **bracket notation**.

```
Notation "[ a , .. , b ]" := (a :: .. (b :: nil) ..).
```

We can now write $[1,3]$ for the list $1 :: 3 :: nil$.

List provides a function

```
in_dec: forall X : Type,
        (forall x y : X, {x = y} + {x <> y}) ->
        forall (x : X) (A : list X), {In x A} + {~ In x A}
```

that given a type X and a decider for equality on X returns a decider for membership for lists over X. We define a boolean membership test for lists over *nat*.

```
Definition inb (x : nat) (xs : list nat) : bool :=
if in_dec eq_nat_dec x xs then true else false.
```

The boolean membership test reflects the propositional membership predicate.

```
Lemma inb_ref x A :
inb x A <-> In x A.
```

Often it is useful to require that every element of a list satisfies a given boolean test. *List* provides a boolean realization of this idea.

```
Fixpoint forallb (X : Type) (f : X -> bool) (A : list X) : bool :=
match A with
| nil => true
| x :: A' => f x && forallb f A'
end.
```

We can prove the following reflection lemma.

```
Lemma forallb_ref X f A :
forallb f A <-> forall x : X, In x A -> f x.

Proof. induction A ; simpl. now firstorder.
rewrite andb_ref, IHA ; clear IHA. split.
intros [B C] x [D|D]. congruence. now auto.
firstorder. Qed.
```

The tactic *firstorder* is an automation tactic that knows about logical operations. We will use *firstorder* only if it can solve a goal. Note that the proof rewrites with the equivalences *andb_ref* and *IHA*.

Exercise 10.4.1 Prove the lemma *inb_ref*.

Exercise 10.4.2 *List* defines an existential version *existsb* of *forallb*. Prove the following reflection lemma.

```
Lemma existsb_ref X f A :
existsb f A <-> exists x : X, In x A /\ f x.
```

Exercise 10.4.3 Prove the following de Morgan Lemma for *forallb* and *existsb*. Use the lemma *negb_andb* from *Bool*.

```
Definition negbfun (X : Type) (f : X -> bool) (x : X) : bool := negb (f x).

Lemma negb_forallb X (f : X -> bool) A :
negb (forallb f A) = existsb (negbfun f) A.
```

10.5 Rewriting with List Equivalences

Let A and B be lists over some type X. We write $A \subseteq B$ and say that A is **included** in B if every element of A is an element of B. We write $A \approx B$ and say that A and B are **equivalent** if and only if A and B have the same elements. *List* defines list inclusion as follows.

```
Definition incl (X : Type) (A B : list X) : Prop :=
forall x, In x A -> In x B.
```

We add a definition of list equivalence.

```
Definition equi X (A B : list X) :=
incl A B /\ incl B A.
```

The functions *cons* and *app* (list concatenation) are **compatible** with list inclusion.

$$\frac{A \subseteq A'}{x :: A \subseteq x :: A'} \qquad \frac{A \subseteq A' \qquad B \subseteq B'}{A +\!+ B \subseteq A' +\!+ B'}$$

Hence they are also compatible with list equivalence. Since list equivalence is an equivalence relation (i.e., is reflexive, symmetric, and transitive), we would hope that Coq can rewrite with list equivalences. It in fact can if we provide it with the necessary proofs.

```
Lemma equi_refl X (A : list X) :  equi A A.
Proof. firstorder. Qed.

Lemma equi_sym X (A B : list X) : equi A B -> equi B A.
Proof. firstorder. Qed.

Lemma equi_tran X (A B C : list X) : equi A B -> equi B C -> equi A C.
Proof. firstorder. Qed.

Add Parametric Relation X : (list X) (@equi X)
reflexivity  proved by (@equi_refl X)
symmetry proved by (@equi_sym X)
transitivity  proved by (@equi_tran X)
as equi_rel.
```

The command starting with *Add* tells the *Setoid* library that we want to rewrite with list equivalences. To justify this, we provide proofs certifying that list equivalence is an equivalence relation. Don't worry about the syntactic details of the command. Next we tell the **Setoid** library that cons is compatible with list equivalence.

```
Add Parametric Morphism X : (@cons X) with
signature eq ==> (@equi X) ==> (@equi X) as equi_cons_comp.
```

From this information *Setoid* generates the goal

```
forall  (x :  X) (A A' :  list  X),  equi A A' -> equi (x :: A) (x ::  A')
```

which we prove as follows.

```
Proof. firstorder. Qed.
```

Next we tell *Setoid* that list concatenation is compatible with list equivalence.

```
Add Parametric Morphism X : (@app X) with
signature (@equi X) ==> (@equi X) ==> (@equi X) as equi_app_comp.

Proof.  intros A B [C D] E F [G H] ;  split
;  auto using incl_app, incl_appl, incl_appr. Qed.
```

The *using* clause tells *auto* that it can use the lemmas *incl_app*, *incl_appl*, and *incl_appl* from *List*.

We can now rewrite with list equivalences.

```
Goal forall (X : Type) (x : X) (A A' B B' : list X),
equi A A' -> equi B B' -> equi (A ++ x :: B) (A' ++ x :: B').
```

```
Proof. intros X x A A' B B' C D. rewrite C, D. reflexivity. Qed.
```

It is not difficult to prove that the predicate *nd* for natural deduction from the last chapter is compatible with list inclusion (see Exercise 10.5.4).

```
Lemma nd_incl G G' s :
incl G G' -> nd G s -> nd G' s.
```

From this it follows that *nd* is compatible with list equivalence under logical equivalence.

$$\frac{G \approx G'}{nd\ G\ s \leftrightarrow nd\ G'\ s}$$

We register this fact with *Setoid* so that we can rewrite the first argument of *nd* with list equivalences.

```
Add Morphism nd with
signature (@equi form) ==> eq ==> iff as nd_mor.
```

```
Proof. firstorder using nd_incl. Qed.
```

We can now rewrite the first argument of *nd* with list equivalences.

```
Goal forall A G G' s,
equi G G' -> (nd (A ++ G) s <-> nd (A ++ G') s).
```

```
Proof. intros A G G' s B. rewrite B. reflexivity. Qed.
```

Exercise 10.5.1 Prove that *cons* and *app* are compatible with list inclusion.

```
Lemma cons_incl X (x : X) A B :
incl A B -> incl (x::A) (x::B).
```

```
Lemma app_incl X (A B C D : list X) :
incl A B -> incl C D -> incl (A++C) (B++D).
```

Exercise 10.5.2 Prove the following fact about list inclusion.

```
Lemma cons_incl_elim X (x : X) A B :
incl (x :: A) B <-> In x B /\ incl A B.
```

Exercise 10.5.3 Prove the following list equivalences. We will rewrite with all of them in the following.

```
Lemma swap_cons X (x y : X) A :
equi (x :: y :: A) (y :: x :: A).

Lemma shift_cons X (x : X) A B :
equi (x :: A ++ B) (A ++ x :: B).

Lemma rotate X (x : X) A :
equi (x :: A) (A ++ [x]).

Lemma push_member X (x : X) A :
In x A -> equi A (x :: A).
```

Exercise 10.5.4 Prove that the predicate *nd* for natural deduction from the last chapter is compatible with list inclusion.

```
Lemma ndM s G :
In s G -> nd G s.

Lemma nd_incl G G' s :
incl G G'-> nd G s -> nd G' s.
```

10.6 Boolean Evaluation and Satisfiability

Given boolean values for the variables, a pure propositional formula can be evaluated to a boolean value. To make this idea precise, we need **assignments**, which assign boolean values to variables. We represent an assignment as a list of variables, where the assignment assigns *true* to a variable if and only if the variable appears in the list.

```
Definition var := nat.

Inductive form : Type :=
| Var : var -> form
| Imp : form -> form -> form
| Fal : form.

Definition assignment := list var.

Fixpoint eval (a : assignment) (s : form) : bool :=
match s with
| Var x => inb x a
| Imp s t => implb (eval a s) (eval a t)
| Fal => false
end.

Definition eva (a : assignment) (G : list form) : bool :=
forallb (eval a) G.
```

An assignment **satisfies** a formula if the formula evaluates under the assignment to *true*. An assignment **satisfies** a list of formulas if it satisfies each formula in the list. A formula or a list of formulas is **satisfiable** if there is an assignment that satisfies the formula or the list.

```
Definition sat (G : list form) : Prop :=
exists a, eva a G.
```

We call a formula **valid** if it is satisfied by every assignment.

```
Definition valid (s : form) : Prop :=
forall a, eval a s.
```

A formula is valid if and only if its negation is unsatisfiable.

```
Lemma valid_unsat s :
valid s <-> ~sat [Not s].
```

```
Proof. unfold sat, valid ; simpl ; split.
intros A [a B]. specialize (A a). now destruct (eval a s) ; auto.
intros A a. apply bcontra ; intros B.
apply A. exists a. destruct (eval a s) ; auto. Qed.
```

Exercise 10.6.1 Prove that the functions *eva* and *sat* are compatible with list inclusion.

```
Lemma eva_incl G G' a :
incl G G' -> eva a G' -> eva a G.
```

```
Lemma sat_incl G G':
incl G G' -> sat G' -> sat G.
```

Exercise 10.6.2 Register the compatibility of *eva* and *sat* with list equivalence with *Setoid*.

Exercise 10.6.3 Prove the following facts about implications.

```
Lemma eva_imp_pos a s t G :
eva a (Imp s t :: G) = eva a (Not s :: G) || eva a (t :: G).
```

```
Lemma eva_imp_neg a s t G :
eva a (Not (Imp s t) :: G) = eva a (s :: Not t :: G).
```

```
Lemma sat_imp_pos s t G :
sat (Imp s t :: G) <-> sat (Not s :: G) \/ sat (t :: G).
```

```
Lemma sat_imp_neg s t G :
sat (Not (Imp s t) :: G) <-> sat (s :: Not t :: G).
```

Exercise 10.6.4 Prove the following fact.

```
Goal forall G s,
(forall a, eva a G -> eval a s) <-> ~sat (Not s :: G).
```

10.7 Soundness

There is a fundamental relationship between classical derivability and boolean evaluation known as **soundness**.

```
Lemma ndc_sound G s a :
ndc G s -> eva a G -> eval a s.
```

Here are some consequences of soundness.

1. If a formula is derivable in a context, then every assignment satisfying the context also satisfies the formula. This holds both for classical and intuitionistic derivability (since intuitionistic derivability implies classical derivability).
2. If a formula is unsatisfied by some assignment, then the formula is not derivable in the empty context. This holds for classical and intuitionistic derivability.
3. If a formula is derivable in the empty context, then it is valid. This holds for classical and intuitionistic derivability.

With (2) arguing that certain formulas are underivable becomes really easy. For instance, each of the formulas $\bot$, x, and $\neg x$ is underivable in the empty context since for each of the formulas there is a unsatisfying assignment. The soundness proof is routine.

```
Lemma ndc_sound G s a :
ndc G s -> eva a G -> eval a s.

Proof. intros A B ; induction A ; simpl in *|-*.
(* ndcA *) now destruct (eval a s) ; simpl ; auto.
(* ndcW *) now destruct (eval a t) ; simpl in B ; auto.
(* ndcII *) now destruct (eval a s) ; simpl in *|- ; auto.
(* ndcIE *) now destruct (eval a s) ; simpl in *|- ; auto.
(* ndcC *) now destruct (eval a s) ; simpl in *|- ; auto. Qed.
```

Note the use of the tactics "simpl in *|-*" and "simpl in *|-". With *|- *simpl* simplifies all assumptions of a goal, and with *|-* *simpl* simplifies all assumptions plus the claim of a goal. We now prove in Coq that $\bot$ is not derivable in the empty context.

```
Goal ~ndc nil Fal.

Proof. intros A. apply ndc_sound with (a:=nil) in A.
contradiction A. exact I. Qed.
```

The proof script applies the lemma *ndc_sound* with the empty assignment to the assumption *A*. This is the first time that we apply a lemma to an assumption. It is also possible to rewrite, unfold, and simplify assumptions.

From soundness it follows that ND-refutable contexts are unsatisfiable.

```
Lemma nd_sat G :
nd G Fal -> ~sat G.

Proof. intros A [a B]. apply nd_ndc in A. exact (ndc_sound A B). Qed.
```

Exercise 10.7.1 Prove the following goals. The second goal is a bit tricky since x is not a concrete variable.

```
Goal forall x, ~ndc nil (Var x).

Goal forall x, ~ndc nil (Not (Var x)).
```

10.8 Completeness and Decidability

We define **classical semantic consequence** as follows.

```
Definition csc (G : list form) (s : form) : Prop :=
forall a, eva a G -> eval a s.
```

Soundness says that classical derivability implies classical semantic consequence.

```
Goal forall G s,  ndc G s -> csc G s.

Proof. intros G s A a. exact (ndc_sound A). Qed.
```

We will also show that classical semantic consequence implies classical derivability. This direction is called **completeness**. Taken together, soundness and completeness say that classical derivability agrees with classical semantic consequence. One speaks of a **semantic characterization** of classical derivability.

In the next chapter we will construct a function

```
sat_nd : forall G, {sat G} + {nd G Fal}.
```

that for a context either constructs a satisfying assignment or an ND refutation. From the existence of such a function completeness follows. Given *sat_nd*, one can also construct a function that given a context and a formula decides whether the formula is derivable from the context in the classical system. This result is known as **decidability** of classical propositional logic.

We will now assume a function *sat_nd* and prove the above consequences. This can be done conveniently with Coq's section construct.

```
Section Main_Results.

Variable sat_nd : forall G, {sat G} + {nd G Fal}.
```

As long as we are in the section, we can use the function *sat_nd*. Once we close the section with the command *End Main_Results*, all definitions and lemmas established in the section will appear with an extra argument asking for a function of type $\forall G.\ \{sat\ G\} + \{nd\ G\ Fal\}$.

We first prove that classical natural deduction agrees with classical semantic consequence.

```
Goal forall G s, ndc G s <-> csc G s.

Proof. split.
intros A a. exact (ndc_sound A).
intros A. apply ndcC, nd_ndc.
destruct (sat_nd (Not s :: G)) as [[a B]|B].
exfalso. specialize (A a). simpl in B. destruct (eval a s) ; tauto.
exact B. Qed.
```

Next we establish a decision procedure for classical derivability.

```
Definition ndc_dec G s : {ndc G s} + {~ndc G s}.

destruct (sat_nd (Not s :: G)) as [A|A].
right. intros B. destruct A as [a A]. simpl in A.
apply andb_ref in A. destruct A as [A C]. apply (ndc_sound B) in C.
destruct (eval a s) ; tauto.
left . apply ndcC, nd_ndc, A. Defined.
```

With *sat_nd* we also obtain a straightforward proof that intuitionistic refutability agrees with classical refutability.

```
Goal forall G, nd G Fal <-> ndc G Fal.

Proof. split. now apply nd_ndc.
intros A ; destruct (sat_nd G) as [B|B].
destruct B as [a B]. contradiction (ndc_sound A B).
exact B. Qed.
```

We now close the section.

```
End Main_Results.
```

The function *ndc_dec* is still available but requires an additional argument that compensates for the assumption *sat_nd*.

```
Check ndc_dec.
% (forall G : list form, sat G + nd G Fal) ->
% forall (G : list form) (s : form), ndc G s + ~ndc G s
```

Exercise 10.8.1 Assume a decision function *sat_nd* and prove the following goals. The lemmas *nd_sat*, *ndc_sound*, and *nd_ndc* suffice.

```
Goal forall G, nd G Fal <-> ~sat G.
```

```
Goal forall s, ndc nil s <-> valid s.
```

Exercise 10.8.2 Assume a decision function *sat_nd* and construct functions as follows.

```
Definition sat_dec G : {sat G} + {~sat G}.
```

```
Definition nd_dec G : {nd G Fal} + {~nd G Fal}.
```

11 Classical Tableaux

In this chapter we study the tableau method for classical propositional logic. With the tableau method we can construct a decision function that for a list of formulas either yields a satisfying assignment or a refutation in the intuitionistic ND system. For the decision function we employ a Coq library providing for generalized recursion.

11.1 Negative Tableau System

In the context of the tableau method we call lists of formulas **clauses**. The tableau method will give us a function

```
sat_nd : forall C, {sat C} + {nd C Fal}.
```

that for a list of formulas either constructs a satisfying assignment or an ND refutation. Before we construct the function, we will consider two complementary tableau systems deriving clauses. We speak of the positive and the negative system. The positive system derives satisfiable clauses and the negative system derives ND-refutable clauses. Since ND-refutable clauses are unsatisfiable, no clause is derivable in both systems. We will eventually construct a function that for a clause yields a derivation in one of the two systems.

Figure 11.1 shows the rules of the negative tableau system. The rules derive clauses. The letter C ranges over clauses, the letters s and t range over formulas, and $\approx$ is list equivalence. The second rule is called **clash rule** and the last rule is called **shuffle rule**. If a clause is derivable with the negative tableau rules, we say that it is **tableau-refutable**.

The negative tableau rules have two **key properties** that are easy to verify.

1. If an assignment satisfies the conclusion of a rule, then the rule has a premise that is satisfied by the assignment.
2. If an assignment satisfies a premise of a rule, then it satisfies the conclusion of the rule.

From property (1) it follows that the tableau rules can only derive unsatisfiable clauses.

The formalization of the negative tableau system in Coq is straightforward.

$$\frac{}{C, \bot} \qquad \frac{}{C, s, \neg s} \qquad \frac{C, \neg s \qquad C, t}{C, s \to t} \qquad \frac{C, s, \neg t}{C, \neg(s \to t)} \qquad \frac{C}{C'}\ C \approx C'$$

Figure 11.1: Negative Tableau System

```
Inductive tab : list form -> Prop :=
| tabF C : tab (Fal :: C)
| tabC s C : tab (Not s :: s :: C)
| tabIP s t C : tab (Not s :: C) -> tab (t :: C) -> tab (Imp s t :: C)
| tabIN s t C : tab (s :: Not t :: C) -> tab (Not (Imp s t) :: C)
| tabS C C' : equi C C' -> tab C -> tab C'.
```

We prove that the clause $[\neg(\neg\neg x \to x)]$ is tableau-refutable.

```
Goal tab [Not (FDN (Var 0))].
```

```
Proof. apply tabIN. apply tabC. Qed.
```

The tableau derivation can be compactly represented with a diagram.

$$\begin{array}{ll} \neg(\neg\neg x \to x) & 1 \\ \quad \neg\neg x & \\ \quad \neg x & \\ \quad \otimes & \end{array}$$

Such diagrams depicting tableau derivations are called **tableaux**. Here is a more interesting tableau deriving the clause $[\neg(((x \to y) \to x) \to x)]$.

$$\begin{array}{lll} \neg(((x \to y) \to x) \to x) & 1 & \\ \quad (x \to y) \to x & 2 & \\ \quad\quad \neg x & & \\ \hline \quad \neg(x \to y) & 3 & x \\ \quad\quad x & & \otimes \\ \quad\quad \neg y & & \\ \quad\quad \otimes & & \end{array}$$

One speaks of the **branches** of a tableau and says that a branch is **closed** if it contains $\bot$ or a **clash** s and $\neg s$. Closed branches are marked with the symbol $\otimes$. The numbers in the tableau indicate the applications of the implication rules. Applications of the shuffle rule are not visible in the tableau. Here is the Coq proof corresponding to the tableau for $[\neg(((x \to y) \to x) \to x)]$.

```
Definition x := Var 0.
Definition y := Var 1.
Definition Peirce := Imp (Imp (Imp x y) x) x.
```

```
Goal tab [Not Peirce].

Proof. apply tabIN. apply tabIP.
apply tabIN.
apply tabS with (C := [Not x, x, Not y]). now firstorder.
now apply tabC.
apply tabS with (C := [Not x, x]). now firstorder.
now apply tabC. Qed.
```

Because of the shuffle rule *tab* is compatible with list equivalence. We register this fact with *Setoid*.

```
Add Morphism tab with
signature (@equi form) ==> iff as tab_equi_comp.
Proof. firstorder using tabS. Qed.
```

Now rewriting with list equivalences can replace the shuffle rule.

```
Goal tab [Not Peirce].

Proof. apply tabIN. apply tabIP.
apply tabIN. rewrite rotate ; simpl. rewrite rotate ; simpl. now apply tabC.
rewrite swap_cons. apply tabC. Qed.
```

Proving that tableau refutations translate into ND refutations is routine.

```
Lemma tab_nd C :
tab C -> nd C Fal.
```

From this result it follows that tableau refutable clauses are unsatisfiable.

It is interesting to compare the negative tableau system with the ND and Hilbert systems we have seen so far. In the ND systems we have action on both sides of the turnstile $\Gamma \vdash s$. In the Hilbert systems all action is on s, and in the negative tableau system all action is on Γ. In fact, the negative tableau system does not have a right side s. It will turn out that all of these systems can refute exactly the same clauses, and that a clause is refutable if and only if it is unsatisfiable.

Exercise 11.1.1 For each of the following formulas s give a tableau refutation of the clause $[\neg s]$ both with a tableau and with a Coq script.

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) $\neg\neg x \to \neg y \to \neg(x \to y)$

Exercise 11.1.2 Consider formulas that feature conjunctions and disjunctions. Extend the tableau system with rules for conjunctions and disjunctions such

$$\frac{}{C}\ C \text{ solved} \qquad \frac{C}{C, \neg\bot} \qquad \frac{C, \neg s}{C, s \to t} \qquad \frac{C, t}{C, s \to t}$$

$$\frac{C, s, \neg t}{C, \neg(s \to t)} \qquad \frac{C}{C'}\ C \approx C'$$

Figure 11.2: Positive Tableau System

that the two key properties for tableau rules are satisfied and conjunctions and disjunctions are decomposed (i.e., do not appear in the premises of the rules). You should have rules for positive and negative conjunctions and for positive and negative disjunctions.

Exercise 11.1.3 Prove that tableau-refutable clauses are ND-refutable.

Lemma tab_nd C : tab C −> nd C Fal.

Also prove that tableau-refutable clauses are unsatisfiable.

Exercise 11.1.4 Prove that weakening is admissible for the negative tableau system. Rewrite with the list equivalence *rotate* from Exercise 10.5.3.

Lemma tabW C s :
tab C −> tab (s :: C).

11.2 Positive Tableau System

A clause is **solved** if it only contains variables and negated variables, and does not contain a clash (i.e., both x and $\neg x$). Every solved clause is satisfiable. As satisfying assignment we can take the list of all variables that occur positively in the clause. For instance, the solved clause $[x, \neg y, \neg z, x']$ is satisfied by the assignment $[x, x']$.

Figure 11.2 shows the positive tableau system. Note that no rule of the positive system has more that one premise. It is easy to see that the positive system can only derive satisfiable clauses. In fact, if a rule has a premise, every assignment satisfying the premise satisfies the conclusion.

A clause is **failed** if it either contains $\bot$ or a clash (i.e., both s and $\neg s$). Obviously, every failed clause is unsatisfiable.

The positive and the negative tableau system are complementary. While the positive system starts from solved clauses and derives satisfiable clauses, the

negative system starts from failed clauses and derives unsatisfiable clauses. The two systems have the shuffle rule and the rule for negated implications in common. As it comes to unnegated implications, the positive system has three rules each with a single premise while the negative system has a single rule with two premisses.

As it comes to depicting the derivations of the positive system, we can still use the tableaux we used for the negative system. In fact, we can still develop the tableau by applying the rules of the negative system backwards. When we apply the rule for positive implications, we branch and follow one branch. A branch is **solved** if it represents a clause that is solved up to occurrences of $\neg\bot$. Here is an example.

$$\begin{array}{cc}
\neg(\neg\neg x \to \neg(x \to \neg y)) & 1 \\
\neg\neg x & 2 \\
\neg\neg(x \to \neg y) & 3 \\
x & \\
\neg\bot & \\
x \to \neg y & 4 \\
\neg\bot & \\
\hline
\neg x \;\big|\; \neg y & \\
\otimes \;\big|\; \text{solved} &
\end{array}$$

The left branch represents the failed clause $\{x, \neg x, \neg\bot\}$ and the right branch represents the solved clause is $\{x, \neg y\}$. We take the freedom to represent clauses as sets, which is justified by the shuffle rule. By the key properties of the negative tableau system we know that the assignments satisfying the initial formula $\neg(\neg\neg x \to \neg(x \to \neg y))$ are exactly the assignments satisfying the solved clause $\{x, \neg y\}$.

We now formalize the positive tableau system in Coq. We choose the following definition of solved clauses.

```
Definition nvar (x : var) : form := Not (Var x).

Definition solved (C : list form) : Prop :=
exists P : list var, exists N : list var,
C = map Var P ++ map nvar N /\ eva P (map nvar N).
```

Representing the rules of the positive tableau system with an inductive definition is straightforward.

```
Inductive cotab : list form -> Prop :=
| cotabV C : solved C -> cotab C
| cotabIPF C : cotab C -> cotab (Not Fal :: C)
| cotabIPL C s t : cotab (Not s :: C) -> cotab (Imp s t :: C)
| cotabIPR C s t : cotab (t :: C) -> cotab (Imp s t :: C)
```

```
| cotabIN C s t : cotab (s :: Not t :: C) -> cotab (Not (Imp s t) :: C)
| cotabS C C' : equi C C' -> cotab C -> cotab C'.
```

Because of the shuffle rule *cotab* is compatible with list equivalence. We register this fact with *Setoid*.

```
Add Morphism cotab with
signature (@equi form) ==> iff as cotab_equi_comp.
Proof. firstorder using cotabS. Qed.
```

Proving that the positive system can only derive satisfiable clauses is routine except for the satisfiability of solved clauses. Here we need a lemma.

```
Lemma eva_map_var A Q :
eva A (map Var Q) <-> incl Q A.

Lemma solved_sat C :
solved C -> sat C.

Lemma cotab_sat C :
cotab C -> sat C.
```

Exercise 11.2.1 Prove the lemmas *eva_map_var*, *solved_sat*, and *cotab_sat*.

Exercise 11.2.2 Formulate a weakening rule for the positive tableau system and prove it correct.

11.3 Signed Tableau System and Subformula Property

We now attack the problem of writing a function

```
tableau : forall C, {cotab C} + {tab C}
```

With *tableau* writing the function *sat_nd* we are aiming at will be straightforward. The idea for *tableau* is as follows: Starting from the initial clause we develop a tableau by applying the negative tableau rules backwards. If we find a solved branch, we have a derivation of the initial clause in the positive system *cotab*. If we end up with a **closed tableau** (a tableau all of whose branches are closed), we have a derivation of the initial clause in the negative system *tab*. We apply the rules in any order we like and do not backtrack.

There is the issue of termination, of course. As is, backwards application of the rules is not terminating. The shuffle rule can be used to duplicate formulas, and the implication $\neg\bot$ reproduces itself. Both issues can be resolved.

We first solve the problem abstractly. We represent clauses as sets of signed formulas. Working with sets instead of lists eliminates the need for the shuffle rule. Signed formulas make it possible to express negation without implication.

$$\frac{}{C, \bot} \qquad \frac{}{C, x, x^-} \qquad \frac{C, s^- \quad C, t}{C, s \to t} \qquad \frac{C, s, t^-}{C, s \to t^-}$$

Figure 11.3: Signed Tableau System

A **signed formula** is a formula together with one of the signs + and −. We write s^- for negatively signed formulas and just s for positively signed formulas. Signed formulas translate to ordinary formulas by omitting the positive sign and replacing the negative sign with negation (i.e. $s^- \rightsquigarrow \neg s$).

Figure 11.3 shows a tableau system deriving signed clauses (i.e., finite sets of signed formulas). The notation C, s^σ is to be read as $C \cup \{s^\sigma\}$. The signed system is an abstract version of the negative tableau system where clauses are represented as sets and negations are represented as negative signs. The two key properties concerning satisfying assignments are still satisfied. In addition, we observe that the premises of a rule of the signed system contain only subformulas of formulas appearing in the conclusion of the rule. This is the so-called **subformula property**. If we take the comma in the conclusions of the implication rules as disjoint union, the premises of the rules are smaller than the conclusions of the rules, where the size of a clauses is taken as the sum of the sizes of the formulas in the clause. Thus backward application of the rules of the signed tableau system will terminate once it has removed all implications. Hence the signed tableau system specifies a procedure that given an initial clause constructs a tableau that either contains a solved branch or is closed.

Observe that the signed clash rule is restricted to variables. The properties of the system are preserved if we generalize the clash rule to all formulas. We will show that the system with the restricted clash rule can still derive all unsatisfiable clauses.

Exercise 11.3.1 Give complete signed tableaux for the following clauses. A tableau is complete if every branch is either closed or solved.

a) $\{\neg\neg x \to \neg y \to \neg(x \to y)^-\}$

b) $\{\neg x \to \neg y \to \neg(y \to x)^-\}$

Exercise 11.3.2 Give signed tableau rules for conjunction and disjunction.

11.4 Decision Procedure

We now realize the signed tableau decision procedure in Coq. We represent a signed clause by four lists P, N, Q, and R as follows:

1. P contains formulas carrying a positive sign.
2. N contains formulas carrying a negative sign.
3. Q contains variables carrying a positive sign.
4. R contains variables carrying a negative sign.

```
Inductive clause : Type :=
| CL : list form -> list form -> list var -> list var -> clause.
```

We define a coercion that converts clauses into lists of formulas.

```
Coercion clause2list (C : clause) : list form :=
match C with CL P N Q R =>
  P ++ map Not N ++ map Var Q ++ map nvar R
end.
```

Note that the negative signs are translated into negations. A clause C is **satisfiable** if and only if the list *clause2list C* is satisfiable. Due to the coercion, we can just write *sat C* to say that the clause C is satisfiable.

We also define a function that converts lists of formulas to clauses.

```
Definition cl (C : list form) := CL C nil nil nil.
```

The conversions *cl* and *clause2list* commute up to list equivalence.

```
Lemma ecl (C : list form) :
equi C (cl C).
```

```
Proof. induction C ; simpl in *|-*. reflexivity.
rewrite <-IHC. reflexivity. Qed.
```

We define the **size** of formulas and clauses as follows.[1]

```
Fixpoint size_form (s : form) : nat :=
match s with
| Var _ => 1
| Imp s t => S (size_form s + size_form t)
| Fal => 1
end.
```

```
Definition size_list := fold_right (fun s a => size_form s + a) 0.
```

```
Definition size_clause (C : clause) : nat :=
match C with CL P N _ _ => size_list P + size_list N end.
```

[1] The function *fold_right* is from the library *List*.

```
Function dec (C : clause) {measure size_clause C} : bool :=

let (P,N,Q,R) := C in
match P with
| Fal :: _ => false
| Var x :: P' => dec (CL P' N (x :: Q) R)
| Imp s t :: P' => dec (CL P' (s :: N) Q R) || dec (CL (t :: P') N Q R)
| nil => match N with
           | Fal :: N' => dec (CL nil N' Q R)
           | Var x :: N' => dec (CL nil N' Q (x :: R))
           | Imp s t :: N' => dec (CL [s] (t :: N') Q R)
           | nil => eva Q (map nvar R)
         end
end.

intros ; simpl ; omega.   ...   intros ; simpl ; omega.

Defined.
```

Figure 11.4: Decision Procedure

Figure 11.4 defines the decision procedure $dec : clause \to bool$. It takes a clause C and returns *true* if and only if it can construct a signed tableau for C that contains a solved branch. The recursion is not structural but on the size of the clause. The library *Recdef* and the keyword *Function* provide for this form of recursion.[2] For each recursive call Coq generates a proof obligation ensuring that the argument of the recursive call is smaller than the primary argument. For our procedure *dec* these obligations are all straightforward and can be shown with *omega*. Step through the proof scripts to understand.

We can compute with the decision procedure.

```
Compute dec (cl [FDN (Var 0)]).
% true

Compute dec (cl [Not (FDN (Var 0))]).
% false
```

Exercise 11.4.1 Make sure you understand every detail of the decision procedure *dec*. You should be able to write the code of *dec* given your understanding of the signed tableau rules. Don't worry about the first line and the proof obligations.

[2] The library *Recdef* realizes size induction with structural recursion on proof terms. The underlying transformation is involved and will not be explained in this chapter.

11.5 Correctness of The Decision Procedure

We establish the correctness of the decision procedure with two lemmas.

```
Lemma dec_sat C :
dec C -> sat C.
```

```
Lemma dec_tab C :
negb (dec C) -> tab C.
```

Both lemmas can be proved by functional induction on the definition of *dec*. The proofs are shown in Figure 11.5. The most complicated case is the nonrecursive base case (i.e., P and Q are both empty), where for *dec_tab* the lemma *clash* is needed, which in turn requires the lemma *eva_map_nvar*.

It is now easy to write the functions *tableau* and *sat_nd*.

```
Definition IBC (x : bool) : {x} + {negb x}.
```

```
destruct x. left. exact I. right. exact I. Defined.
```

```
Definition tableau C : {cotab C} + {tab C}.
```

```
destruct (IBC (dec (cl C))) as [A|A].
left. rewrite ecl. exact (dec_cotab A).
right. rewrite ecl. exact (dec_tab A). Defined.
```

```
Definition sat_nd C : {sat C} + {nd C Fal}.
```

```
destruct (tableau C) as [A|A].
left. now apply cotab_sat, A.
right. now apply tab_nd, A. Defined.
```

Note that the function *tableau* combines the function *dec* and the accompanying correctness lemmas *dec_cotab* and *dec_tab* into a single object. Once we have the function *tableau*, there is no need to go back to the function *dec* and its correctness lemmas.

Exercise 11.5.1 Prove the following goals. The function *tableau* and the lemmas *cotab_sat*, *tab_nd*, and *nd_sat* suffice.

```
Goal forall C, cotab C <-> sat C.
```

```
Goal forall C, tab C <-> ~sat C.
```

Exercise 11.5.2 Define functions as follows.

```
Definition tab_dec C : {tab C} + {~tab C}.
```

```
Definition cotab_dec C : {cotab C} + {~cotab C}.
```

```
Lemma dec_cotab C :
dec C -> cotab C.

Proof. functional induction (dec C) ; simpl ; intros A.
(* Fal+ *)  contradiction A.
(* Var+ *)  rewrite shift_cons, shift_cons. exact (IHb A).
(* Imp+ *)  rewrite orb_ref in A ; destruct A as [A|A].
apply cotabIPL. rewrite shift_cons. exact (IHb A).
apply cotabIPR. exact (IHb0 A).
(* Fal- *)  apply cotabIPF. exact (IHb A).
(* Var- *)  rewrite shift_cons, shift_cons. exact (IHb A).
(* Imp- *)  apply cotabIN. exact (IHb A).
(* Done *)  apply cotabV. exists Q, R. auto. Qed.

Lemma eva_map_nvar Q R :
eva Q (map nvar R) = forallb (fun x => negb (inb x Q)) R.

Proof. induction R ; simpl. reflexivity.
rewrite IHR. destruct (inb a Q) ; reflexivity. Qed.

Lemma clash P N :
negb (eva P (map nvar N)) -> tab (map Var P ++ map nvar N).

Proof. rewrite eva_map_nvar, negb_forallb, existsb_ref. intros [x [A B]].
unfold negbfun in B. rewrite negb_involutive, inb_ref in B.
apply tabClash with (s:= Var x).
apply in_or_app ; right. exact (in_map _ _ _ A).
apply in_or_app ; left. exact (in_map _ _ _ B). Qed.

Lemma dec_tab C :
negb (dec C) -> tab C.

Proof. intros A ; functional induction (dec C) ; simpl in *|-*.
(* Fal+ *)  now apply tabF.
(* Var+ *)  rewrite shift_cons, shift_cons. exact (IHb A).
(* Imp+ *)  rewrite negb_orb, andb_ref in A. destruct A as [A1 A2].
apply tabIP. rewrite shift_cons. exact (IHb A1). exact (IHb0 A2).
(* Fal- *)  apply tabW. exact (IHb A).
(* Var- *)  rewrite shift_cons, shift_cons. exact (IHb A).
(* Imp- *)  apply tabIN. exact (IHb A).
(* Done *)  exact (clash A). Qed.
```

Figure 11.5: Correctness Proofs

12 Kripke Models and Independence Results

In the previous chapter we gave meaning to propositional formulas by evaluating them under a boolean assignment (assigning boolean values to variables). In this chapter we consider a different way of giving meaning to propositional formulas. Instead of boolean assignments, we will use Kripke models. The intuitionistic ND system is sound for Kripke models, but the classical ND system is not. For this reason, we can use Kripke models to prove that certain formulas are not intuitionistically provable. In particular, we will prove $\emptyset \nvdash_{\mathcal{N}} \neg\neg x \to x$.

12.1 Models for Intuitionistic Logic

In the classical case it was enough to consider assignments (sets or lists of variables). 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. Clearly if an assignment assigns x to *true*, then $\neg\neg x \to x$ will be *true*. Likewise, if an assignment is such that $\neg x$ evaluates to *true*, then $\neg\neg x \to x$ will be *true*. What we need is some way to interpret formulas so that a formula may be neither true nor false. One option is to use sets of assignments and to reconsider how we interpret implication. As a simple example, consider the two assignments $\emptyset$ and $\{x\}$. Since $\emptyset \subseteq \{x\}$, we consider $\{x\}$ as an extension of $\emptyset$. We can represent the two assignments as the following simple tree.

Let us (temporarily) refer to these two assignments "possible worlds" – $\emptyset$ is a possible world in which x is not yet true and $\{x\}$ is a later possible world in which x is true. Let us write $w \vDash s$ to mean s is forced to be true at the possible world w. We will also more briefly say w forces s.

We would like to define $w \vDash s$ so that $\emptyset \nvDash x$ and $\emptyset \nvDash \neg x$. In order to accomplish this, we will define $\vDash$ so that $w \vDash y$ holds if $y \in w$ and $w \vDash \neg y$ holds if

y is not in any of the extensions of w. In this particular case, $\emptyset \not\models x$ since $x \notin \emptyset$. $\emptyset \not\models \neg x$ since $x \in \{x\}$ and $\{x\}$ is an extension of $\emptyset$.

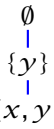
Recall that $\neg x$ is $x \to \bot$. We will define $\models$ so that $w \not\models \bot$. We want to know that $w \models \neg y$ holds if y is not in any extension of w. We can ensure this by defining $\models$ such that $w \models s \to t$ if every extension w' of w is such that if $w' \models s$, then $w' \models t$. (A more precise definition will be given in a more general setting in the next section.)

Since one of our main goals is to prove $\Gamma \not\vdash_{\mathcal{N}} s$ for example contexts Γ and formulas s, we will need interpretations with a possible world that forces all the formulas in Γ, but does not force s. We now consider a few such examples informally and construct an appropriate sets of assignments (as a set of "possible worlds").

Suppose we would like to force the formulas $x \to y$ and $\neg x \to y$ without forcing y. It is easy to see that having only the possible worlds $\emptyset$ and $\{x\}$ is not enough, since $\{x\} \models x$ but $\{x\} \not\models y$ and hence neither world forces $x \to y$. A simple fix is to replace the possible world $\{x\}$ with $\{x, y\}$.

Clearly $\emptyset \models x \to y$ and $\emptyset \not\models y$. Also, $w \not\models \neg x$ at both possible worlds and so $w \models \neg x \to y$.

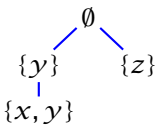
We can also have more than two assignments. Consider the following example with three assignments.



We compute $w \models s$ for the three worlds and various formulas s and display them in the following table. Make sure for each value in the table you can justify why $w \models s$ or $w \not\models s$. In the cases in which s is of the form $t \to u$, it suffices to look at the row for t and the row for u.

	$\emptyset$	$\{y\}$	$\{x, y\}$
x	$\not\models$	$\not\models$	$\models$
$\neg x$	$\not\models$	$\not\models$	$\not\models$
$\neg\neg x$	$\models$	$\models$	$\models$
y	$\not\models$	$\models$	$\models$
$\neg y$	$\not\models$	$\not\models$	$\not\models$
$\neg\neg y$	$\models$	$\models$	$\models$
$x \to y$	$\models$	$\models$	$\models$
$y \to x$	$\not\models$	$\not\models$	$\models$

We can also choose assignments so that inclusion is not a linear ordering. Considering the following example with four assignments.



We again compute $w \vDash s$ for the four worlds and various formulas s and display them as a table.

	$\emptyset$	$\{y\}$	$\{x,y\}$	$\{z\}$
x	$\nvDash$	$\nvDash$	$\vDash$	$\nvDash$
$\neg x$	$\nvDash$	$\nvDash$	$\nvDash$	$\vDash$
$\neg\neg x$	$\nvDash$	$\vDash$	$\vDash$	$\nvDash$
y	$\nvDash$	$\vDash$	$\vDash$	$\nvDash$
$\neg y$	$\nvDash$	$\nvDash$	$\nvDash$	$\vDash$
$\neg\neg y$	$\nvDash$	$\vDash$	$\vDash$	$\nvDash$
z	$\nvDash$	$\nvDash$	$\nvDash$	$\vDash$
$\neg z$	$\nvDash$	$\vDash$	$\vDash$	$\nvDash$
$\neg\neg z$	$\nvDash$	$\nvDash$	$\nvDash$	$\vDash$
$x \to y$	$\vDash$	$\vDash$	$\vDash$	$\vDash$
$y \to x$	$\nvDash$	$\nvDash$	$\vDash$	$\vDash$
$y \to z$	$\nvDash$	$\nvDash$	$\nvDash$	$\vDash$
$\neg x \to z$	$\vDash$	$\vDash$	$\vDash$	$\vDash$

In order to find enough counterexamples for intuitionistic propositional logic, it suffices to consider a finite number of assignments which form a tree under inclusion.

In the next section we generalize such models, define how one evaluates a formula given such a model, and prove the relationship to intuitionistic provability.

12.2 Kripke Models

Recall that an assignment is used to assign boolean values to variables. We represented an assignment by a list of variables in the previous chapter, but the idea is that the list represents the set of variables which are assigned to *true* and any variables not in the set are assigned to *false*. In this section we will work at the mathematical level and simply work with sets of variables.

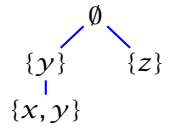
A **Kripke model** is a triple $(W, \leq, \alpha)$ where W is a nonempty set, $\leq$ is a reflexive, transitive relation on W, and α is a function from W to sets of variables such that for every $w \leq w'$ we have $\alpha(w) \subseteq \alpha(w')$. We will call the elements of W **worlds**. When $w \leq w'$ we will say that w' is **later** than w. We will also write $w' \geq w$ to mean the same thing as $w \leq w'$.

Let $(W, \leq, \alpha)$ be a fixed Kripke model. For each world $w \in W$ and formula s, we define a relation $w \vDash s$ by recursion on s. When $w \vDash s$ holds, we say w **forces** s. The definition is as follows.

- $w \vDash x$ if $x \in \alpha(w)$
- $w \nvDash \bot$
- $w \vDash s \to t$ if for all $w' \geq w$: $w' \vDash s$ implies $w' \vDash t$.

For a context Γ we write $w \vDash \Gamma$ if $w \vDash s$ for every s in Γ.

Each of the examples from the previous section gives a Kripke model. Reconsider the following four assignments:



Let W be $\{\emptyset, \{y\}, \{x, y\}, \{z\}\}$. Let $w \leq w'$ mean $w \subseteq w'$ and let $\alpha(w) = w$. Clearly $(W, \leq, \alpha)$ is a Kripke model. It is easy to verify $\emptyset$ forces $x \to y$ and $z \to \neg x$, but does not force the formulas y, $\neg x \to y$, $\neg(y \to x)$ and $\neg\neg z$.

From the previous example it is clearly possible for a world not to force a formula even though a later world does force the formula. It is natural to ask whether there is a similar example in which $w \leq w'$, $w \vDash s$ and $w' \nvDash s$. In fact, there is no such example. The definitions of Kripke models and forcing are such that we have the following **monotonicity** property: If $w \vDash s$ and $w \leq w'$, then $w' \vDash s$.

Theorem 12.2.1 (Monotonicity) Let $(W, \leq, \alpha)$ be a Kripke model. If $w \vDash s$ and $w \leq w'$, then $w' \vDash s$.

Proof We argue by cases on s. If $w \vDash x$ and $w \leq w'$, then $x \in \alpha(w) \subseteq \alpha(w')$ and so $w' \vDash x$. Since $w \nvDash \bot$, there is nothing to show if s is $\bot$. Assume $w \vDash s \to t$ and $w \leq w'$. We must prove $w' \vDash s \to t$. Let $w'' \geq w'$ such that $w'' \vDash s$ be given. By transitivity, $w \leq w''$. Since $w \vDash s \to t$, we know $w'' \vDash t$ as desired. ∎

The monotonicity property will be vital for proving the intuitionistic ND system is sound for Kripke models.

Exercise 12.2.2 Let $(W, \leq, \alpha)$ be a Kripke model. Argue the following facts.

a) $w \vDash \neg s$ if and only if $w' \nvDash s$ for all $w' \geq w$.

b) $w \nvDash \neg s$ if and only if $w' \vDash s$ for some $w' \geq w$.

c) $w \vDash \neg\neg s$ if and only if for every $w' \geq w$ there is some $w'' \geq w'$ such that $w'' \vDash s$.

Exercise 12.2.3 Prove Kripke models are a generalization of boolean assignments as follows. Suppose a is a boolean assignment. Find a Kripke model $(W, \leq, \alpha)$ such that for every formula s, s evaluates to *true* under assignment a if and only if $w \vDash s$ for every $w \in W$.

12.3 Soundness

We now prove the main soundness result of this chapter.

Theorem 12.3.1 (Soundness) Let $(W, \leq, \alpha)$ be a Kripke model. If $\Gamma \vdash_{\mathcal{N}} s$, then for every world $w \in W$ we have $w \vDash \Gamma$ implies $w \vDash s$.

Proof The proof is by induction on $\Gamma \vdash_{\mathcal{N}} s$.

$\mathsf{A}_{\mathcal{N}}$: Consider $\Gamma, s \vdash_{\mathcal{N}} s$. If $w \vDash \Gamma, s$, then $w \vDash s$ in particular.

$\mathsf{W}_{\mathcal{N}}$: Suppose $\Gamma, s \vdash_{\mathcal{N}} t$ follows from $\Gamma \vdash_{\mathcal{N}} t$. Assume $w \vDash \Gamma, s$. In particular, $w \vDash \Gamma$. By the inductive hypothesis $w \vDash t$ as desired.

$\mathsf{E}^{\perp}_{\mathcal{N}}$: Suppose $\Gamma \vdash_{\mathcal{N}} s$ follows from $\Gamma \vdash_{\mathcal{N}} \perp$. Assume $w \vDash \Gamma$. By the inductive hypothesis $w \vDash \perp$, a contradiction.

$\mathsf{E}^{\rightarrow}_{\mathcal{N}}$: Suppose $\Gamma \vdash_{\mathcal{N}} t$ follows from $\Gamma \vdash_{\mathcal{N}} s \rightarrow t$ and $\Gamma \vdash_{\mathcal{N}} s$. Assume $w \vDash \Gamma$. By the inductive hypotheses $w \vDash s \rightarrow t$ and $w \vDash s$. Since $w \leq w$, we know $w \vDash t$.

$\mathsf{I}^{\rightarrow}_{\mathcal{N}}$: Suppose $\Gamma \vdash_{\mathcal{N}} s \rightarrow t$ follows from $\Gamma, s \vdash_{\mathcal{N}} t$. Assume $w \vDash \Gamma$. We must prove $w \vDash s \rightarrow t$. Let $w' \geq w$ such that $w' \vDash s$ be given. We must prove $w' \vDash t$. By monotonicity we know $w' \vDash \Gamma$. Since $w' \vDash \Gamma, s$ we conclude $w' \vDash t$ by the inductive hypothesis. ∎

We can use the soundness result to conclude many consequences about Kripke models. Suppose $(W, \leq, \alpha)$ is a Kripke model and $w \in W$. Since $\emptyset \vdash_{\mathcal{N}} s \rightarrow s$ we know $w \vDash s \rightarrow s$. We can also conclude $w \nvDash \neg(s \rightarrow s)$.

We will usually use the soundness result to conclude that certain formulas are not provable in the intuitionistic ND system.

Exercise 12.3.2 There is no such soundness result for classical provability. Which rule of the classical ND calculus causes a problem?

Exercise 12.3.3 Let $(W, \leq, \alpha)$ be a Kripke model. Argue the following facts.

a) $w \vDash s \rightarrow \neg\neg s$ for all $w \in W$ and formulas s.

b) $w \vDash s \rightarrow t \rightarrow s$ for all $w \in W$ and formulas s and t.

c) $w \vDash (s \rightarrow t \rightarrow u) \rightarrow (s \rightarrow t) \rightarrow s \rightarrow u$ for all $w \in W$ and formulas s, t and u.

12.4 Independence Results

A formula s is **independent of the intuitionistic ND system** (or simply **independent**) if $\emptyset \nvdash_{\mathcal{N}} s$ and $\emptyset \nvdash_{\mathcal{N}} \neg s$. A very simple example of an independent formula is a variable x. We know neither x nor $\neg x$ is provable in the classical system (via soundness of classical ND with respect to boolean assignments). Hence neither is provable in the intuitionistic system.

A more interesting example is $\neg\neg x \to x$. Obviously $\emptyset \nvdash_{\mathcal{N}} \neg(\neg\neg x \to x)$ since $\emptyset \vdash_{\mathcal{NC}} \neg\neg x \to x$. We can also demonstrate $\emptyset \nvdash_{\mathcal{N}} \neg(\neg\neg x \to x)$ by choosing any Kripke model with a single world.

In order to prove $\emptyset \nvdash_{\mathcal{N}} \neg\neg x \to x$ we must construct a Kripke model with a world w such that $w \not\Vdash \neg\neg x \to x$. Let $(W, \leq, \alpha)$ be the Kripke model in which W is $\{\emptyset, \{x\}\}$, $w \leq w'$ means $w \subseteq w'$ and $\alpha(w) = w$. As a tree the Kripke model can be presented as follows:

$$\begin{array}{c} \emptyset \\ | \\ \{x\} \end{array}$$

(Note that this is exactly the same as the first example we considered in this chapter.) We will prove $\emptyset \not\models \neg\neg x \to x$. Since $\emptyset \not\models x$, it suffices to prove $\emptyset \models \neg\neg x$. In order to prove this we must verify that $\emptyset \not\models \neg x$ and $\{x\} \not\models \neg x$. Both of these follow from $\{x\} \models x$.

Exercise 12.4.1 Which of the following formulas are independent? Justify your answer either by giving appropriate proofs in the intuitionistic ND system or by giving appropriate Kripke models.

a) $\neg(\neg\neg x \to x)$

b) $(x \to y) \to (\neg x \to y) \to y$

c) $((x \to y) \to x) \to x$

Exercise 12.4.2 Suppose Γ and s are such that $\Gamma \vdash_{\mathcal{NC}} s$. Argue that $\Gamma \nvdash_{\mathcal{N}} \neg s$.

Exercise 12.4.3 Is there a Kripke model with a world w such that for every two distinct variables x and y we have $w \not\Vdash x \to y$?

12.5 Formalization in Coq

As always, we give a formal version of the results of this chapter in Coq. Instead of formalizing Kripke models in full generality, we will restrict ourselves to Kripke models given by a set of assignments. All the particular Kripke models we have considered so far were in this form. Assume the set of worlds W is finite and each world w is a finite set of variables. Assume $w \leq w'$ if and only

if $w \subseteq w'$. Further assume $\alpha(w)$ is simply w. To specify such a Kripke model it suffices to specify the particular finite set W of finite sets of variables. Since we do not have finite sets in Coq, we instead use lists. As before, a list of variables is an assignment. Hence we represent a Kripke model in Coq as simply a list of assignments.

The forces relation can be defined in Coq as recursive functions mapping to either Prop or bool. We consider the versions mapping to Prop first.

```
Definition assignment := list var.

Fixpoint forcesP (W : list assignment) (a : assignment) (s : form) : Prop :=
match s with
| Var x => In x a
| Imp s t => forall a', In a' W -> incl a a' -> forcesP W a' s -> forcesP W a' t
| Fal => False
end.

Definition forces'P (W : list assignment) (a : assignment) (G : list form) : Prop :=
forall s, In s G -> forcesP W a s.
```

The monotonicity result for formulas can be formalized as follows. The proof is by a case analysis on s and is left as an exercise.

```
Lemma monotone_forcesP (W : list assignment) (a a' : assignment) (s : form) :
In a' W -> incl a a' -> forcesP W a s -> forcesP W a' s.
Proof. intros A' B C. destruct s.
simpl. apply B. exact C.
intros a'' A'' B' C'. apply C; try assumption.
now apply incl_tran with (m := a').
contradiction C.
Qed.
```

Here is the Coq statement of the soundness result. The proof is by induction on the proof of nd G s and is left as an exercise.

```
Lemma soundnessKP (W : list assignment) G s : nd G s ->
forall a, In a W -> forces'P W a G -> forcesP W a s.
Proof. intros D. induction D; try now firstorder.
intros a A B a' A' B' C.
apply IHD. assumption.
intros u [E|E].
rewrite <- E. assumption.
apply monotone_forcesP with (a := a); try assumption.
apply B. assumption.
Qed.
```

The forces relation can also be defined in Coq as the recursive functions forces and forces' mapping to bool. We make use of the boolean version implb of

implication and a boolean version inclb of list inclusion. For readability, we use infix notation for these boolean functions.

```
Definition inclb (a a': list nat) : bool :=
forallb (fun x => inb x a') a.

Notation "w --> w'" := (implb w w').
Notation "w <= w'" := (inclb w w').

Fixpoint forces (W : list assignment) (w : assignment) (s : form) : bool :=
match s with
| Var x => inb x w
| Imp s t => forallb (fun w' => (w <= w') --> (forces W w' s) --> (forces W w' t)) W
| Fal => false
end.

Definition forces' (W : list assignment) (w : assignment) (G : list form) : bool :=
forallb (forces W w) G.
```

It is possible to prove reflection results.

```
Lemma forces_ref W w s : forces W w s <-> forcesP W w s.

Lemma forces'_ref W w G : forces' W w G <-> forces'P W w G.
```

The boolean versions of the monotonicity and soundness results follow.

```
Lemma monotone_forces (W : list assignment) (w w' : assignment) (s : form) :
In w' W -> w <= w' -> forces W w s -> forces W w' s.
Proof. rewrite inclb_ref. rewrite forces_ref. rewrite forces_ref. now apply monotone_forcesP. Qed.

Lemma soundnessK (W : list assignment) G s : nd G s ->
forall w, In w W -> forces' W w G -> forces W w s.
Proof. intros A w. rewrite forces'_ref. rewrite forces_ref. now apply soundnessKP. Qed.
```

We can now prove $\emptyset \nvdash_{\mathcal{N}} \neg\neg x \to x$ in Coq. The Kripke model we used was the one with worlds $\emptyset$ and $\{x\}$. In Coq we use the list [nil,[0]].

```
Lemma unprovable_DN : ~nd nil (Imp (Not (Not x)) x).
Proof. intros D.
pose (W := [nil,[0]]).
assert (Wnil:In nil W). left. reflexivity.
assert (A:forces' W nil nil). simpl. tauto.
exact (soundnessK D Wnil A).
Qed.
```

We can also prove the negation of the double negation formula is unprovable. In this case any one world model will do.

```
Lemma unprovable_nDN : ~nd nil (Not (Imp (Not (Not x)) x)).
Proof. intros D.
pose (W := [@nil nat]).
assert (Wnil:In nil W). left. reflexivity.
assert (A:forces' W nil nil). simpl. tauto.
exact (soundnessK D Wnil A).
Qed.
```

We define independence as a simple conjunction.

```
Definition indep s := ~nd nil s /\ ~nd nil (Not s).
```

Independence of the double negation principle follows.

```
Lemma indep_DN : indep (Imp (Not (Not x)) x).
Proof. split. apply unprovable_DN. apply unprovable_nDN. Qed.
```

Exercise 12.5.1 Do the proofs of monotone_forcesP and soundnessKP in Coq.

Exercise 12.5.2 Prove the following in Coq.

```
Lemma unprovable_Peirce : ~nd nil (Imp (Imp (Imp x y) x) x).

Lemma unprovable_nPeirce : ~nd nil (Not (Imp (Imp (Imp x y) x) x)).

Lemma indep_Peirce : indep (Imp (Imp (Imp x y) x) x).

Lemma unprovable_PM : ~nd nil (Imp (Imp x y) (Imp (Imp (Not x) y) y)).

Lemma unprovable_nPM : ~nd nil (Not (Imp (Imp x y) (Imp (Imp (Not x) y) y))).

Lemma indep_PM : indep (Imp (Imp x y) (Imp (Imp (Not x) y) y)).

Lemma unprovable_PWM : ~nd nil (Imp (Imp (Not x) y) (Imp (Imp (Not (Not x)) y) y)).

Lemma unprovable_nPWM : ~nd nil (Not (Imp (Imp (Not x) y) (Imp (Imp (Not (Not x)) y) y))).

Lemma indep_PWM : indep (Imp (Imp (Not x) y) (Imp (Imp (Not (Not x)) y) y)).
```

12.6 Signed Tableaux for Intuitionistic Logic

Up until now when we have desired a Kripke model to demonstrate unprovability of a formula in a context we have simply given a small model and checked that in some world the formulas in the context are forced but the conclusion formula is not forced. We now present a more systematic method for constructing

$$\frac{}{C, \bot} \qquad \frac{}{C, x, x^-} \qquad \frac{C, s^- \qquad C, t}{C}\ s \to t \in C \qquad \frac{C^+, s, t^-}{C}\ s \to t^- \in C$$

Figure 12.1: Signed Intuitionistic Tableau System

counterexamples. In particular, we give a signed tableau system for intuitionistic logic.

A **clause** is a finite set of signed formulas. We write C^+ for the clause consisting only of the positive formulas in C. That is, C^+ is $\{s | s \in C\}$. Equivalently, C^+ is $C \setminus \{s^- | s^- \in C\}$. Figure 12.1 shows a tableau system. If one can derive a signed clause C in this system, then we say there is an **intuitionistic tableau refutation of C**. We will usually use the tableau system to construct Kripke models satisfying C.

Let $(W, \leq, \alpha)$ be a Kripke model. We say w **satisfies** C if $w \vDash s$ for every positive $s \in C$ and if $w \nvDash s$ for every negative $s^- \in C$. Note that if w satisfies C and $s \to t \in C$, then w either satisfies C, s^- or satisfies C, t. Also, if w satisfies C and $s \to t^- \in C$, then there is some $w' \geq w$ such that w' satisfies C^+ (by monotonicity), $w' \vDash s$ and $w' \nvDash t$. From these facts we know that if there is a world satisfying the conclusion of a tableau rule, then one can choose a premise which is also satisfied by a (possibly different) world.

We say C is **satisfiable** if there is a Kripke model with a world satisfying C. As above, if a clause C is satisfiable and C is the conclusion of a tableau rule in Figure 12.1, then one of the premises of the rule is satisfiable.

Also, note that if we start form a clause C and apply the tableau rules backwards, we will only consider clauses containing signed subformulas of the formulas in C. There are only finitely many such clauses and so there are only finitely many tableaux to consider. If there is no refutation, then appropriate choices of premises will lead to a set of clauses satisfying certain closure conditions we now describe.

A clause C is **Hintikka** if the following three conditions hold:

1. $\bot \notin C$.
2. If $s \in C$, then $s^- \notin C$.
3. If $s \to t \in C$, then $s^- \in C$ or $t \in C$.

A set D of Hintikka clauses is a **demo** if for every $s \to t^- \in C \in D$ there is some $C' \in D$ such that $s \in C'$, $t^- \in C'$ and $C^+ \subseteq C'^+$.

A demo D induces a Kripke model as follows. We take D as the set of worlds and with each Hintikka clause $C \in D$ as a world. We define $C \leq C'$ to hold if

$C^+ \subseteq C'^+$. That is, $C \leq C'$ if every positive formula in C is also in C'. Equivalently, $C \leq C'$ iff $C^+ \subseteq C'$. Finally, let $\alpha(C)$ be $\{x | x \in C\}$. Clearly we have $\alpha(C) \subseteq \alpha(C')$ whenever $C \leq C'$ in D. Hence $(D, \leq, \alpha)$ is a Kripke model.

Theorem 12.6.1 (Demo) Let D be a demo. For every formula s and Hintikka clause $C \in D$, we have the following:

1. If $s \in C$, then $C \vDash s$.
2. If $s^- \in C$, then $C \nvDash s$.

Consequently, for each $C \in D$, $C \vDash D$.

Proof We prove this by induction on s. Let $C \in D$ be given. If $x \in C$, then $x \in \alpha(C)$ and so $C \vDash x$. If $x^- \in C$, then $x \notin C$ since C is Hintikka and so $x \notin \alpha(C)$ and $C \nvDash x$. We never have $\bot \in C$ since C is Hintikka. We always have $C \nvDash \bot$.

Suppose $s \to t \in C$. We must check $C \vDash s \to t$. Let $C' \in D$ such that $C \leq C'$ and $C' \vDash s$ be given. Since $s \to t \in C^+ \subseteq C'^+ \subseteq C'$, we know $s \to t \in C'$. Since C' is Hintikka, $s^- \in C'$ or $t \in C'$. Since $C' \vDash s$, we cannot have $s^- \in C'$ by the inductive hypothesis for s. Hence $t \in C'$. By the inductive hypothesis for t, we know $C' \vDash t$ as desired.

Suppose $s \to t^- \in C$. We must check $C \nvDash s \to t$. Since D is a demo, there is some $C' \in D$ such that $C^+ \subseteq C'$, $s \in C'$ and $t^- \in C'$. Since $C^+ \subseteq C'$ we have $C \leq C'$ and so C' witnesses $C \nvDash s \to t$. ∎

We now consider a few examples demonstrating how we can obtain demos using the tableau rules.

Consider the clause with two negative implications: $x \to y^-$ and $y \to x^-$. We first apply the negative implication rule to $x \to y^-$ and obtain a new clause with x and y^-.

$$\begin{array}{c} x \to y^-, y \to x^- \\ | \\ x, y^- \end{array}$$

We next apply the negative implication rule to $y \to x^-$ and obtain a new clause with y and x^-.

$$\begin{array}{ccc} x \to y^-, y \to x^- & & \\ \swarrow & & \searrow \\ x, y^- & & y, x^- \end{array}$$

There are no more rules to apply and in fact we have obtained a demo. The top clause is a world which forces neither $x \to y$ nor $y \to x$. We know this fact by the Demo Theorem.

Let us now reconsider the example $\neg\neg x \to x$. We begin with the clause $\{\neg\neg x \to x^-\}$. Applying the tableau rule to $\neg\neg x \to x^-$ we obtain

$$\begin{array}{c} \neg\neg x \to x^- \\ | \\ \neg\neg x, x^- \end{array}$$

We next apply the tableau rule to $\neg\neg x$ (i.e., $\neg x \to \bot$). We can either add $\neg x^-$ or $\bot$ to the bottom clause. However, $\bot$ will cause a conflict and we would like to construct a demo. Hence we add $\neg x^-$ to the clause.

$$\begin{array}{c} \neg\neg x \to x^- \\ | \\ \neg\neg x, x^-, \neg x^- \end{array}$$

Now we have a new negative implication formula: $\neg x^-$. We add a new clause in which we copy the positive formulas and include x and $\bot^-$.

$$\begin{array}{c} \neg\neg x \to x^- \\ | \\ \neg\neg x, x^-, \neg x^- \\ | \\ \neg\neg x, x, \bot^- \end{array}$$

Finally we apply the implication rule to $\neg\neg x$ in the new clause and add $\neg x^-$ to the new clause.

$$\begin{array}{c} \neg\neg x \to x^- \\ | \\ \neg\neg x, x^-, \neg x^- \\ | \\ \neg\neg x, x, \bot^-, \neg x^- \end{array}$$

We now have a demo and the top clause is a world which does not force $\neg\neg x \to x$.

We can also obtain the following relationship between intuitionistic tableau refutability and the intuitionistic natural deduction system.

For a signed clause C let C^- be $\{s | s^- \in C\}$.

Theorem 12.6.2 If there is an intuitionistic tableau refutation of C, then there is an $u \in C^- \cup \{\bot\}$ such that $C^+ \vdash_{\mathcal{N}} u$.

Proof If $\bot \in C$, then we clearly have $C^+ \vdash_{\mathcal{N}} \bot$. If $x \in C$ and $x^- \in C$, then we clearly have $C^+ \vdash_{\mathcal{N}} x$.

Suppose $s \to t \in C$ and both C, s^- and C, t have intuitionistic tableau refutations. By the inductive hypothesis for C, s^- we either have $C^+ \vdash_{\mathcal{N}} u$ for some $u \in C^- \cup \{s^-, \bot\}$. If $u \in C^- \cup \{\bot\}$, then we are done. Otherwise, u is s and we have $C^+ \vdash_{\mathcal{N}} s$. Since $s \to t \in C^+$ we know $C^+ \vdash_{\mathcal{N}} s \to t$ and so $C^+ \vdash_{\mathcal{N}} t$. By the inductive hypothesis for C, t we have $C^+, t \vdash_{\mathcal{N}} v$ for some $v \in C^- \cup \{\bot\}$. Hence $C^+ \vdash_{\mathcal{N}} t \to v$ and so $C^+ \vdash_{\mathcal{N}} v$ as desired.

Finally, suppose $s \to t^- \in C$ and C^+, s, t^- has an intuitionistic tableau refutation. By the inductive hypothesis either $C^+, s \vdash_{\mathcal{N}} \bot$ or $C^+, s \vdash_{\mathcal{N}} t$. In either case $C^+, s \vdash_{\mathcal{N}} t$ and so $C^+ \vdash_{\mathcal{N}} s \to t$ as desired. ∎

We consider a simple example. We start with the clause with signed formulas $x, x \to y, y^-$ and z^-. The only rule which applies is the implication rule for $x \to y$. Both premises lead to a conflict since adding x^- conflicts with x and adding y conflicts with $y-$. By Theorem 12.6.2 we know that either $x, x \to y \vdash_{\mathcal{N}} \bot$, $x, x \to y \vdash_{\mathcal{N}} y$ or $x, x \to y \vdash_{\mathcal{N}} z$. In fact we know $x, x \to y \vdash_{\mathcal{N}} y$.

12.7 Remarks

Kripke, a philosopher and logician, originally used Kripke models to provide 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.

Independence results have become more prevalent in mathematics in the past century. In 1900 Hilbert gave a list of 23 unsolved problems in Mathematics. Some of these remain unsolved to this day. The first problem was known as the Continuum Hypothesis. From Cantor's theorem, we know that the set of all sets of natural numbers is not countable. The Continuum Hypothesis asserts that there is no collection of a size between the size of the set of all natural numbers (countable) and the size of the set of all sets of natural numbers. The most popular logical system of set theory is first-order Zermelo-Fraenkel. Once one has a logical system in which one can express the Continuum Hypothesis as a formula, then it becomes possible to prove that neither it nor its negation is provable. In 1940 Gödel proved the negation of the Continuum Hypothesis is not provable. In 1963 Cohen proved the Continuum Hypothesis itself is also not provable. Cohen used a technique called forcing. It is due to the similarities between Cohen's forcing and Kripke semantics that the term "forcing" is often used in the context of Kripke semantics for intuitionistic logic. Indeed in Kripke's 1965 paper on models for intuitionistic logic, he discusses the Cohen result in the context of a Kripke model.

The tableau system is due to Fitting 1969.

13 Quotients

In this chapter we study quotients. For example, we form the set of finite sets of natural numbers as the quotient of lists of natural numbers modulo list equivalence. In set-theoretic mathematics, one can always form such quotients. In type theory, the situation is different. One can only form quotients when one can find a computational implementation for the quotient. We will also introduce a new aspect of Coq: modules.

13.1 Sigma Types

In mathematics it is common to separate out a subset from a given set using a given property. Such a set is usually written $\{x \in A | Px\}$. Informally, this is the set of all elements of A satisfying the property P. We can obtain similar types in type theory using **sigma types**.

Sigma types are formed using the inductive type sig defined in the Coq library as follows.

```
Inductive sig (A : Type) (P : A -> Prop) : Type :=
    exist : forall x : A, P x -> sig P
```

The argument A is implicit for both sig and exist. Coq allows the mathematical notation {x:A | P x} to be used for the sigma type sig (fun x:A => P x).

A special case of sigma types are **boolean sigma types**. A boolean sigma type is of the form {x:A |p x} where p:A -> bool. One way boolean sigma types are special is that we can prove the following lemma.

```
Lemma sig_bPI (X:Type) (p:X -> bool) (x y : X) (H : p x) (H' : p y) : x = y
 -> exist (fun z => p z) x H = exist (fun z => p z) y H'.
```

We leave the proof of this as an exercise.

Exercise 13.1.1 Recall the definition of sig (sigma types).

```
Inductive sig (A : Type) (P : A -> Prop) : Type :=
    exist : forall x : A, P x -> sig P
```

Prove the following in Coq.

```
Lemma bPI (x:bool) : forall (H H':x), H = H'.
```

```
Lemma sig_bPI (X:Type) (p:X -> bool) (x y : X) (H : p x) (H' : p y) : x = y
 -> exist (fun z => p z) x H = exist (fun z => p z) y H'.
```

13.2 Equivalence Relations as Modules

One can specify a mathematical structure in Coq by encapsulating a signature of names and types into a **module type**. A **module** encapsulates a signature of names and definitions. Coq can check that a module M implements a given module type Φ by ensuring that for every name c declared as having type A in Φ there is a definition c of type A in M. Since propositions are types module types can also specify properties which must be proven of any module implementing the module type.

As a simple example, we specify equivalence relations as a module type. An equivalence relation on a type X is a binary relation E which is reflexive, symmetric and transitive.

```
Module Type EQUIVRELN.
 Parameter X:Type.
 Parameter E:X -> X -> Prop.
 Axiom Eref : forall x, E x x.
 Axiom Esym : forall x y, E x y -> E y x.
 Axiom Etra : forall x y z, E x y -> E y z -> E x z.
End EQUIVRELN.
```

A simple example of an equivalence relation is the full relation on a type. To be specific, we take the type of natural numbers. Here is a module implementing the module type EQUIVRELN using the natural numbers and the full relation.

```
Module NATMODALL <: EQUIVRELN.
Definition X := nat.
Definition E := fun n m:nat => True.
Lemma Eref : forall x, E x x.
Proof. firstorder. Qed.
Lemma Esym : forall x y, E x y -> E y x.
Proof. firstorder. Qed.
Lemma Etra : forall x y z, E x y -> E y z -> E x z.
Proof. firstorder. Qed.
End NATMODALL.
```

13.3 Quotients

We now consider the mathematical notion of a quotient and determine a corresponding notion in type theory. Suppose X is a set and $\sim$ is an equivalence

relation on X. For each $x \in X$, let $\tilde{x}$ be the set $\{y \in X | x \sim y\}$. By reflexivity we know $x \in \tilde{x}$ and so $\tilde{x}$ is nonempty. Using symmetry and transitivity we know $\tilde{x} = \tilde{y}$ whenever $x \sim y$.

We define $X/\sim$ to be the set $\{\tilde{x} | x \in X\}$ and call this the **quotient set of X modulo $\sim$**. There is clearly a function A mapping X onto $X/\sim$ defined as $A(x) := \tilde{x}$. Since each member of $X/\sim$ is nonempty for each $\tilde{x}$ we can choose an element $C(\tilde{x})$ of $\tilde{x}$. Since each $\tilde{x}$ is a subset of X we know C is a function from $X/\sim$ to X. Note that $x \sim C(\tilde{x})$ since $C(\tilde{x}) \in \tilde{x}$. The function A gives us an abstract version of x in which we forget the particular representative x of the equivalence class $\tilde{x}$. Conversely the function C chooses a concrete representative of an equivalence class $\tilde{x}$.

We now have three objects: $X/\sim$, $A : X \to X/\sim$ and $C : X/\sim \to X$. Let us explore what properties relate these objects. We have already noted that A is surjective. We also know that A maps $\sim$-equivalent elements ($x \sim y$) to equal elements ($A(x) = \tilde{x} = \tilde{y} = A(y)$). We can also prove C is injective relative to the equivalence relation $\sim$ on X. Suppose $C(\tilde{x}) \sim C(\tilde{y})$. In this case we have

$$x \sim C(\tilde{x}) \sim C(\tilde{y}) \sim y$$

and so $x \sim y$ and $\tilde{x} = \tilde{y}$. It is also easy to check that $C(A(x)) \sim x$ and $A(C(\tilde{x})) = \tilde{x}$ (since $C(\tilde{x}) \in \tilde{x}$). Hence A and C act as inverses relative to the equivalence relation $\sim$.

In Coq we cannot copy these mathematical constructions exactly. However, we can consider a quotient of a type X by an equivalence relation $\sim$ to be given by a type Q and two functions $Abs : X \to Q$ and $Con : Q \to X$ These functions should be inverses to one another (relative to $\sim$ on X), Abs should respect the equivalence relation $\sim$ and be surjective and Con should be injective relative to $\sim$. For all these properties to hold it is enough to require the following two properties:

- For all $a, b \in Q$ if $Con(a) \sim Con(b)$, then $a = b$.
- For every $x \in X$ we have $Con(Abs(x)) \sim x$.

13.4 Quotients as Functors

Since quotients depend on equivalence relations, we must consider module types and modules which depend on other modules. Module types and modules which depend on modules are called **functor types** and **functors**, respectively. In Coq we specify a quotient relative to a given equivalence relation as the following functor type.

```
Module Type QUOT (E':EQUIVRELN).
```

```
  Export E'.
  Parameter Q:Type.
  Parameter Abs:X -> Q.
  Parameter Con:Q -> X.
  Axiom C_inj : forall a b, E (Con a) (Con b) -> a = b.
  Axiom CA_id : forall x, E (Con (Abs x)) x.
End QUOT.
```

In this definition E' refers to a module of module type EQUIVRELN (i.e., an equivalence relation). The **Export** E' statement allows us to reference the names X, E, Eref, Esym and Etra from the module E'. When we apply QUOT to a particular equivalence relation module we obtain a module type. For example, QUOT NATMODALL is the module type for the quotient of nat by the full relation. An implementation of the module QUOT NATMODALL is a quotient of nat by the full relation. We can easily implement such a quotient using the unit type.

```
Module NATMODALLQUOT : QUOT NATMODALL.
Export NATMODALL.
Definition Q := unit.
Definition Abs := fun n:nat => tt.
Definition Con := fun _:unit => 0.
Lemma C_inj : forall a b, E (Con a) (Con b) -> a = b.
Proof. intros [] [] H. reflexivity. Qed.
Lemma CA_id : forall x, E (Con (Abs x)) x.
Proof. intros x. unfold E. tauto. Qed.
End NATMODALLQUOT.
```

In the previous section we informally argued that requiring the properties C_inj and CA_id is sufficient to obtain all the properties we expect of a quotient. We can verify this in Coq by proving appropriate theorems. One way to make such results reusable in the future is to define a functor which assumes a quotient module and extends the module with a number of theorems. We call this functor QUOT2.

```
Module QUOT2 (E':EQUIVRELN) (Q':QUOT E').
  Export E'.
  Export Q'.

  Add Parametric Relation : X E
  reflexivity  proved by Eref
  symmetry proved by Esym
  transitivity  proved by Etra
  as Erel.

  Add Parametric Morphism : Abs with
  signature E ==> (@eq Q) as A_eq.
```

```
Proof. intros x y H. apply C_inj. rewrite CA_id. rewrite CA_id. assumption. Qed.

Lemma AC_id : forall a, Abs (Con a) = a.
Proof. intros a. apply C_inj. rewrite CA_id. reflexivity. Qed.

Lemma A_inj : forall x y, Abs x = Abs y -> E x y.
Proof. intros x y H. rewrite <- CA_id. rewrite H. rewrite CA_id. reflexivity. Qed.

Lemma A_fun (Z:Type) (f:X -> Z) : (forall x y, E x y -> f x = f y) ->
  forall  x,  (fun a => f (Con a)) (Abs x) = f x.
Proof. intros H x. simpl. apply H. now apply CA_id. Qed.

End QUOT2.
```

Exercise 13.4.1 For each of the following equivalence relations, implement a quotient.

a) nat modulo $\approx$ where $n \approx m$ if n and m are the same modulo 2. (That is, $n \approx m$ if either both are even or both are odd.)

b) list nat modulo $\approx$ where $l \approx l'$ if l and l' have the same length.

c) bool -> bool modulo functional equivalence.

13.5 Finite Sets of Naturals

We can obtain a type of finite sets of natural numbers by constructing a quotient of finite lists of natural numbers over the list equivalence $\approx$ considered in Section 10.5. Recall $A \approx B$ if and only if A and B have the same elements. The equivalence relation can be implemented as follows.

```
Module NATSETEQUIV <: EQUIVRELN.
Definition X := list nat.
Definition E := fun A B:list nat => equi A B.
Lemma Eref : forall x, E x x.
Proof. exact (@equi_refl nat). Qed.
Lemma Esym : forall x y, E x y -> E y x.
Proof. exact (@equi_sym nat). Qed.
Lemma Etra : forall x y z, E x y -> E y z -> E x z.
Proof. exact (@equi_tran nat). Qed.
End NATSETEQUIV.
```

The facts equi_refl, equi_sym and equi_tran were proven in Section 10.5 using the firstorder tactic.

It will take some time to implement finite sets as a quotient (i.e., as a module of module type QUOT NATSETEQUIV). We can develop the basic theory in

the abstract by assuming we have an implementation. We assume we have an implementation by defining a functor SET.

```
Module SET (M:QUOT NATSETEQUIV).
Export M.
```

The type Q is the assumed type of finite sets. For readability, let us define an abbreviation set for Q.

```
Definition set := Q.
```

Recall that we have a general membership predicate In on lists. On lists of natural numbers we have the boolean predicate inb and the corresponding reflection lemma.

```
Definition inb (x : nat) (xs :  list  nat) :  bool :=
if  in_dec eq_nat_dec x xs then true else false.

Lemma inb_ref x A : inb x A <-> In x A.
```

We can use Con to lift inb to membership on sets.

```
Definition insb n X : bool := inb n (Con X).
```

We can now verify one of the main properties we wanted: two sets are the same if they have the same members. This property is known as **set extensionality**.

```
Lemma setext X Y : (forall n, insb n X <-> insb n Y) -> X = Y.

Proof. intros H. apply C_inj. split.
intros n. specialize (H n). unfold insb in H. rewrite inb_ref in H. rewrite inb_ref in H.
now firstorder.
intros n. specialize (H n). unfold insb in H. rewrite inb_ref in H. rewrite inb_ref in H.
now firstorder. Qed.
```

We next consider how we can construct sets. The empty set $\emptyset$ is the abstract version of the empty list. We can prove the empty set has no elements.

```
Definition empty : set := Abs (@nil nat).

Lemma emptyE n : ~insb n empty.
```

Given any natural number n, we can define the singleton $\{n\}$ using the singleton list. We can prove $n \in \{n\}$ and that $m = n$ for any $m \in \{n\}$.

```
Definition sing n : set := Abs [n].

Lemma singI n : insb n (sing n).

Lemma singE m n : insb m (sing n) -> m = n.
```

Finally we define binary union $X \cup Y$ via concatenation of lists.

```
Definition union X Y : set := Abs (Con X ++ Con Y).
```

For every sets X and Y and natural numbers n we can prove the following facts:

- $n \in X \to n \in X \cup Y$
- $n \in Y \to n \in X \cup Y$
- $n \in X \cup Y \to n \in X \vee n \in Y$

In Coq these lemmas are given as follows.

```
Lemma unionI1 X Y n : insb n X -> insb n (union X Y).

Lemma unionI2 X Y n : insb n Y -> insb n (union X Y).

Lemma unionE X Y n : insb n (union X Y) -> insb n X \/ insb n Y.
```

We finally end the definition of the functor.

```
End SET.
```

13.6 Quotients via Normalization

We can use boolean sigma types to construct many quotients. Suppose we have an equivalence relation with type X : T and relation $E : X \to X \to \mathit{Prop}$. Suppose we further have a boolean predicate $p : X \to \mathit{bool}$ and a normalization function $n : X \to X$ satisfying three properties:

- $nu : \forall xy : X, px \to py \to Exy \to x = y$
- $np : \forall x : X, p(nx)$
- $nE : \forall x : X, E(nx)x$

We can construct a quotient as follows:

```
Definition Q : Type := {x:X |p x}.

Definition Abs : X -> Q := fun x => exist p (n x) (np x).

Definition Con : Q -> X := fun X => let (x,_) := X in x.

Lemma C_inj : forall a b:Q, E (Con a) (Con b) -> a = b.
Proof. intros [x Hx] [y Hy]. unfold Con. intros H1. apply sig_bPI. now apply nu. Qed.

Lemma CA_id : forall x:X, E (Con (Abs x)) x.
Proof. intros x. unfold Con,Abs. apply nE. Qed.
```

This is the process we will use to construct a quotient for lists of natural numbers modulo list equivalence $\approx$. We will construct a boolean predicate

$sortedb : list\ nat \to bool$ recognizing strictly sorted lists of natural numbers (ordered by $<$) and a normalization function $sort : list\ nat \to list\ nat$ which sorts a list of natural numbers. In order to construct the quotient as a boolean sigma type we will prove three facts from above.

- $\forall AB : list\ nat, sortedb\ A \to sortedb\ B \to A \approx B \to A = B$
- $\forall A : list\ nat, sortedb\ (sort\ A)$
- $\forall A : list\ nat, sort\ A \approx A$

13.7 Sorted Lists

Even though we only need sortedb as a boolean predicate, it will be easier to reason about an equivalent inductive predicate sorted.

```
Inductive sorted : list nat -> Prop :=
| sorted_nil : sorted nil
| sorted_sing x : sorted [x]
| sorted_rec x y A : x<y -> sorted (y::A)  -> sorted (x::y::A).
```

Mathematically we can define sorted with the rules

$$\frac{}{sorted\ nil} \qquad \frac{}{sorted\ [x]} \qquad \frac{sorted(x :: A)}{sorted\ (x :: y :: A)}\ x < y$$

By inversion we have the following:

```
Lemma sorted_cons x A :
sorted (x::A) -> sorted A.
```

We also know that if $(x :: A)$ is sorted then x must be less than every element in A. We make an auxiliary relation ltL expressing this and prove the fact.

```
Definition ltL (x : nat) (A : list nat) : Prop :=
forall y, In y A -> x < y.

Lemma sorted_cons_lt x A :
sorted (x::A) -> ltL x A.
```

Also, if x is less than every element of a sorted list A, then $(x :: A)$ is sorted.

```
Lemma sorted_lt x A :
sorted A -> ltL x A -> sorted (x::A).
```

We now can prove that if $(x :: A) \subseteq (y :: B)$ where $(x :: A)$ and $(y :: B)$ are both sorted, then $y \leq x$ and $A \subseteq B$.

```
Lemma sorted_incl_lt x y A B :
sorted (x::A) -> sorted (y::B) -> incl (x::A) (y::B) -> y <= x.
```

```
Lemma sorted_incl x y A B :
sorted (x::A) -> sorted (y::B) -> incl (x::A) (y::B) -> incl A B.
```

Using these lemmas we can prove one of the main properties we will need: If A and B are sorted and $A \approx B$, then $A = B$.

```
Lemma sorted_equi A B :
sorted A -> sorted B -> equi A B -> A=B.
```

We now define a function sort which sorts lists. We use the insertion sort algorithm. We first need a function insert to insert x into a list A such that $insert\, x\, A \approx (x :: A)$ and $insert\, x\, A$ is sorted if A is sorted. We make use of the function

```
lt_eq_lt_dec : forall n m : nat, {n < m} + {n = m} + {m < n}
```

from the Coq library.

```
Fixpoint insert (x : nat) (A : list nat) : list nat :=
match A with
| nil => [x]
| y::A' => match lt_eq_lt_dec x y with
             | inleft (left _) => x::A
             | inleft (right _) => A
             | inright _ => y :: insert x A'
           end
end.
```

The relevant lemmas about insert can be proven by induction on lists.

```
Lemma insert_equi x A :
equi (insert x A) (x :: A).

Lemma insert_sorted x A :
sorted A -> sorted (insert x A).
```

We can now define the sorting algorithm.

```
Fixpoint sort (A : list nat) : list nat :=
match A with
| nil => A
| x::A => insert x (sort A)
end.
```

By induction on lists we can prove the following vital properties.

```
Lemma sort_equi A :
equi (sort A) A.

Lemma sort_sorted A :
sorted (sort A).
```

Note that sort_equi is exactly one of the three properties we need.

In order to complete the realization of the quotient we define a boolean predicate sortedb and prove the corresponding reflection lemma.

```
Fixpoint sortedb (A : list nat) : bool :=
match A with
| nil => true
| x::A' => match A' with
             | nil => true
             | y::_ => if lt_dec x y then sortedb A' else false
           end
end.

Lemma sortedb_ref A :
sortedb A <-> sorted A.
```

We can use this reflection lemma to obtain the other two of the three properties we need.

```
Lemma sortedb_equi A B :
sortedb A -> sortedb B -> equi A B -> A=B.

Proof. rewrite sortedb_ref. rewrite sortedb_ref. now apply sorted_equi. Qed.

Lemma sortedb_sort A :
sortedb (sort A).

Proof. apply sortedb_ref, sort_sorted. Qed.
```

We can now implement the quotient using the boolean sigma type {x: list nat|sortedb x} and hence obtain an implemention of finite sets of natural numbers.

```
Module NATSET <: QUOT NATSETEQUIV.
Definition Q := {l | sortedb l}.
Definition Abs := fun A:list nat => exist sortedb (sort A) (sortedb_sort A).
Definition Con := fun X:Q => let (A,_) := X in A.
Lemma C_inj : forall X Y:Q, equi (Con X) (Con Y) -> X = Y.
Proof. destruct X as [A HA], Y as [B HB] ; simpl ; intros C.
now apply sig_bPI, sortedb_equi. Qed.
Lemma CA_id : forall A:list nat, equi (Con (Abs A)) A.
Proof. simpl. apply sort_equi. Qed.
End NATSET.
```

14 Least Number Search

In this chapter we will define a function *first* taking a boolean predicate $p : nat \to bool$ and a proof of $\exists x : nat, px$ and return the least natural number n such that pn. Writing such a function requires a couple of new ideas. In a programming language, we would ignore the proofs and write a recursive function that starting from 0 searches for the first number satisfying p. In Coq, we face the problem that increasing a number is not structurally recursive. For that reason we will write a recursive function taking as arguments a *counter* and a *permission*. The counter is incremented by 1 by each recursion step. The *permission* is decreased by each recursion step. So the recursion is on the permission. We represent permissions as proof terms for an inductive predicate *safe*.

```
Inductive safe (p : nat -> bool) (n : nat) : Prop :=
| safeI : p n -> safe p n
| safeS : safe p (S n) -> safe p n.
```

If we have a permission $A : safe\ p\ n$, we know that there is a $k \geq n$ satisfying p and that the size of the permission is an upper bound on the number of steps it takes an upward search starting at n to find a number satisfying p.

We define propositions $low\ p\ n$ stating that nothing smaller than n satisfies p and $min\ p\ n$ stating that n is the least natural number satisfying p.

```
Definition low (p : nat -> bool) (n : nat) : Prop :=
forall k, k<=n -> p k -> k = n.

Definition min (p : nat -> bool) (n : nat) : Prop :=
p n /\ low p n.
```

When $p : nat \to bool$ Coq allows us to write `ex p` for `ex (fun x:nat => p x)` (i.e., $\exists x.px$). We will write a function $firstc'$ that given a number n, a permission of type $safe\ p\ n$, and a proof of $low\ p\ n$ returns the least $k \geq n$ satisfying p together with a proof of $min\ p\ k$. We can define this by recursion on the permission A. The type of $firstc'$ will be

```
forall (p : nat -> bool) (n : nat) (A : safe p n), low p n -> {k:nat|min p k}.
```

The first thing the function will do is make a case distinction on whether $p\ n$ is true or false. Since we will also need a proof of $p\ n$ or $\neg p\ n$ in each case, we first obtain the following function.

```
Definition IBC' (x : bool) : {x} + {~ x}.
destruct x. left. exact I. right. tauto. Defined.
```

If $p\ n$, then *firstc'* will return n (with a proof of $min\ p\ n$). If $\neg p\ n$, then we will make a recursive call with Sn and a structurally smaller permission. We will also need a proof of $low\ p\ (Sn)$ to make the recursive call. For this reason we need the following lemma.

```
Lemma lowS p n : low p n -> ~ p n -> low p (S n).
Proof. intros A B k C D. destruct (le_lt_eq_dec k (S n) C) as [E|E].
assert (F:k = n). apply A. omega. assumption.
exfalso. apply B. rewrite <- F. assumption.
exact E. Qed.
```

Note that we have used the function

```
le_lt_eq_dec : forall n m : nat, n <= m -> {n < m} + {n = m}
```

from the Coq library.

We define *firstc'* by recursion.

```
Fixpoint firstc' (p : nat -> bool) n (A:safe p n) : low p n -> {k:nat|min p k} :=
fun B =>
match (IBC' (p n)) with
| left  C => exist (fun k => min p k) n (conj C B)
| right C =>
  firstc' (match A with
             | safeI D => match (C D) with end
             | safeS D => D
             end)
           (lowS B C)
end.
```

We can alternatively define *firstc'* with a script as follows.

```
Fixpoint firstc' (p : nat -> bool) n (A:safe p n) : low p n -> {k:nat|min p k}.
intros B. destruct (IBC' (p n)) as [C|C].
exists n. split. exact C. exact B.
apply firstc' with (n := S n).
destruct A as [D|D].
contradiction (C D).
exact D.
now apply lowS. Defined.
```

Both of these presentations of the definition requires some explanation.

The function *firstc'* is given a counter n, a permission A and a proof B of $low\ p\ n$. It first tests whether the counter satisfies p. If this is the case, it returns the counter n paired with the proof of $pn \wedge low\ p\ n$ (i.e., $min\ p\ n$). Otherwise, the function recurses with the incremented counter and a permission that is

obtained with a match on the given permission A. For the recursion to be structural, the permission obtained with the match must be structurally smaller than the permission A. This is the case if each rule of the match yields a permission that is smaller than A. In case A was formed with *safeI* we must have a proof of $\neg(pc)$, a contradiction. For this reason, the body of the first rule is a match (on a proof of *False*) with no rules and hence yields a permission smaller than A (since each of its rules does). The body of the second rule returns a permission that is obtained from A by stripping off the constructor *safeS*.

An ordinary recursive function first does a match on the argument it recurses on and then recurses in the bodies of some of the rules. The function *first'* modifies this pattern in that it delegates the match to an argument term of the recursive application. We speak of an **eager proof recursion**. With eager proof recursion it is possible to recurse and match on proofs but nevertheless return values that are not proofs. This is impossible with the ordinary recursion pattern since it would violate the elim restriction.

We can now define a function *firstc* of type

```
forall  p:nat −> bool, ex p −> {n:nat|min p n}
```

by calling *firstc'* with initial counter value 0. With the following lemmas we know 0 is safe and low.

```
Lemma safe_O p :  forall n, safe p n −> safe p O.
Proof. induction n. tauto. intros A. apply IHn, safeS, A. Defined.

Lemma ex_safe (p : nat −> bool) :  ex p −> safe p O.
Proof.  intros [n A]. exact (safe_O (safeI A)). Defined.

Lemma lowO p : low p O.
Proof. intros k. omega. Qed.
```

We can now easily define *firstc*.

```
Definition firstc (p :  nat −> bool) :  ex p −> {n:nat|min p n} :=
fun E => firstc'  (ex_safe E) (@lowO p).
```

We can also define the function *first* mapping to *nat* by forgetting the proof of *min p n*.

```
Definition first (p :  nat −> bool) (A : ex p) : nat :=
let  (n,_)  :=  firstc  A in n.
```

Finally we can define a function of type

```
forall  (p:nat −> bool), (exists x:nat, p x) −> {x:nat|p x}.
```

by simply calling *firstc* and forgetting that the return value n satisfies *low p n*. In general we say a **choice operator for boolean predicates** on type X is an element of type

```
forall (p:X -> bool), (exists x:X, p x) -> {x:X|p x}.
```

We record this in the following definition

```
Definition Choice_b (X:Type) : Type :=
 forall (p:X -> bool), (exists x:X, p x) -> {x:X|p x}.
```

As promised, we can use *firstc* to obtain a choice operator for boolean predicates on *nat*.

```
Definition choose_b_nat : Choice_b nat.
intros p A. destruct (firstc A) as [n [B _]]. exists n. exact B. Defined.
```

Exercise 14.0.1 Is there a choice operator for boolean predicates on *bool*?

15 Mathematical Assumptions

In Chapter 3 we briefly discussed excluded middle and the double negation law as propositions that are not provable in Coq. Nevertheless, these are natural assumptions one makes when working in a mathematical context. In this chapter we consider different propositions which are not provable in Coq but are natural mathematical assumptions.

15.1 Classical Assumptions

Recall the propositions *XM* (excluded middle) and *DN* (the double negation law).

```
Definition XM : Prop := forall X : Prop, X \/ ~X.

Definition DN : Prop := forall X : Prop, ~~X -> X.
```

Neither of these are provable in Coq, but both are natural mathematical assumptions. In fact, they are provably equivalent in Coq (see Exercise 3.15.2), so that by assuming either one, we obtain a proof of the other.

Another equivalent proposition is Peirce's law.

```
Definition PEIRCE : Prop := forall X Y : Prop, ((X -> Y) -> X) -> X.
```

After considering *XM*, *DN* and *PEIRCE*, one may get the impression that all classical assumptions are equivalent. This is not the case. An example is the **Gödel-Dummett (*GD*)** proposition:

```
Definition GD :=
 forall  X Y:Prop, (X -> Y) \/ (Y -> X).
```

We can prove *GD* from *XM*, but *XM* is not provable from *GD* in Coq.

Exercise 15.1.1 Prove *GD* follows from *XM*.

```
Goal XM -> GD.
```

15.2 Extensional Assumptions

Extensional assumptions allow us to prove certain objects are equal by proving they have a common property. None of these propositions are provable in Coq.

- **Functional Extensionality (*FuncE*):** Two functions are equal if they have the same value on all arguments.

```
Definition FuncE : Prop :=
 forall X Y : Type, forall f g:X -> Y, (forall x:X, f x = g x) -> f = g.
```

- **Propositional Extensionality (*PropE*):** Two propositions are equal if they are equivalent.

```
Definition PropE : Prop :=
 forall X Y : Prop, (X <-> Y) -> X = Y.
```

- **Predicate Extensionality (*PredE*):** Two predicates are equal if the same elements satisfy the predicates.

```
Definition PredE : Prop :=
 forall X:Type, forall p q:X -> Prop, (forall x:X, p x <-> q x) -> p = q.
```

We prove *PredE* follows from *FuncE* and *PropE*.

```
Lemma FuncE_PropE_PredE : FuncE -> PropE -> PredE.
Proof. intros fe pe X p q A. apply fe. intros x. apply pe. firstorder. Qed.
```

Also, predicate extensionality implies propositional extensionality.

```
Lemma PredE_PropE : PredE -> PropE.
Proof. intros pe X Y A.
assert (B:(fun _:unit => X) = (fun _ => Y)).
apply (pe unit (fun _ => X) (fun _ => Y)). intros _. assumption.
change ((fun _ => X) tt = (fun _ => Y) tt). rewrite B. reflexivity. Qed.
```

15.3 Proof Irrelevance

Another proposition which is not provable in Coq is **proof irrelevance (*PI*)**. Proof irrelevance says that there is at most one proof of any proposition.

```
Definition PI : Prop := forall X:Prop, forall A B:X, A = B.
```

In Coq, one can prove *PropE* implies *PI*. We will prove this result in the rest of this section. One can also prove *XM* implies *PI*. Note that without the elim restriction, we could distinguish proofs. This would allow us to prove the negation of *PI* and hence the negation of both *XM* and *PropE*. One reason the elim restriction is so restrictive is so that Coq remains consistent with *XM* and *PropE*.

In the rest of this section we prove *PI* form *PropE*. We start by defining an inductive proposition P_2 with two proof constructors p_0 and p_1.

```
Inductive P2 : Prop := p0 : P2 | p1 : P2.
```

Propositional extensionality can be used to prove $p_0 = p_1$ as follows. Assume propositional extensionality. Using the identity function from $(\lambda x : P_2.x)$ as witnesses for f and g we can easily prove

$$\exists f : P_2 \to P_2.\exists g : P_2 \to P_2.\forall v : P_2, g(fv) = v.$$

Since P_2 is provable, propositional extensionality implies $(P_2 \to P_2) = P_2$. By rewriting with this equality we know

$$\exists f : (P_2 \to P_2) \to P_2.\exists g : P_2 \to P_2 \to P_2.\forall v : P_2 \to P_2.g(fv) = v.$$

Let f and g be such that $\forall v : P_2 \to P_2.g(fv) = v$. Now let $n : P_2 \to P_2$ be the function such that $n\,p_0 = p_1$ and $n\,p_1 = p_0$. Let $v : P_2 \to P_2$ be $\lambda x.n(gxx)$ and $y : P_2$ be $g(fv)(fv)$. Note that since $g(fv) = v$ we have

$$y = g(fv)(fv) = v(fv) = n(g(fv)(fv)) = ny.$$

That is, $y = ny$. By a final case analysis we know either $y = p_0$ and so $p_0 = np_0 = p_1$ or $y = p_1$ and so $p_1 = np_1 = p_0$.

Here is the proof as a Coq script.

```
Lemma PropE_P2PI : PropE -> p0 = p1.
Proof. intros pe.
assert (A:exists f:(P2 -> P2) -> P2, exists g:P2 -> P2 -> P2, forall v:P2 -> P2, g (f v) = v).
assert (B:(P2 -> P2) = P2). apply pe. split. intros _. exact p0. tauto.
rewrite B. exists (fun x => x). exists (fun x => x). intros x. reflexivity.
destruct A as [f [g B]].
pose (n x := match x with p0 => p1 | p1 => p0 end).
pose (v := (fun x => n (g x x))).
pose (y := g (f v) (f v)).
assert (n y = y). unfold y at 2. rewrite B. reflexivity.
case_eq y; intros C.
rewrite <- C. rewrite <- H. rewrite C. simpl. reflexivity.
rewrite <- C. rewrite <- H. rewrite C. simpl. reflexivity. Qed.
```

We can now prove that propositional extensionality implies proof irrelevance. Assume propositional extensionality and that A and B are both proofs of a proposition X. Let $h : P_2 \to X$ be defined such that $hp_0 = A$ and $hp_1 = B$. By the previous result $p_0 = p_1$ and hence $A = B$.

Here is the proof as a Coq proof script.

```
Lemma PropE_PI : PropE -> PI.
Proof. intros pe X A B.
pose (h x := match x with p0 => A | p1 => B end).
assert (C:A = h p0). reflexivity.
rewrite C. rewrite (PropE_P2PI pe). reflexivity. Qed.
```

15.4 Choice

In Chapter 14 we constructed a choice operator for boolean predicates on *nat*. A **choice operator** on type X is an element of type

```
forall  p:X -> Prop, ex p -> {x:X|p x}
```

We can record this in the following definition.

```
Definition Choice (X:Type) : Type := forall (p:X -> Prop), (exists x:X, p x) -> {x:X|p x}.
```

In mathematics it is sometimes useful to assume there is a choice operator at every type. This is one form of the **axiom of choice**.

If we assume *nat* has a choice operator and we assume predicate extensionality, then we can prove excluded middle. This result is called **Diaconescu's Theorem** and was first proven in the 1970s. We give the informal argument and then give the corresponding Coq proof script.

Assume predicate extensionality and let c be a choice operator on type *nat*. Let X be a proposition. For $x : nat$ let $p\ x$ be the proposition $x = 0 \vee X$ and let $q\ x$ be the proposition $x = 1 \vee X$. Clearly we have proofs $A : \exists x, px$ and $B : \exists x, qx$. We know $c\ p\ A$ yields a natural number n with a proof of pn. Either n is 0 or X holds. If X holds, we are done. Assume $n = 0$. Likewise $c\ q\ B$ yields a natural number m with a proof of qm. Either m is 1 or X holds. If X holds, we are done. Assume $m = 1$. In this final case we will prove $\neg X$. Assume X. Under this assumption it is easy to prove $\forall x.px \leftrightarrow qx$ and so $p = q$ by predicate extensionality. Hence the propositions $\exists x.px$ and $\exists x.qx$ are the same and so A and B are proofs of the same proposition. Predicate extensionality implies propositional extensionality which implies proof irrelevance. Hence A and B are equal. Thus $n = m$ and so $0 = 1$, a contradiction.

Here is the corresponding Coq proof script.

```
Lemma Choice_nat_PredE_XM : Choice nat -> PredE -> XM.
Proof. intros c pe X.
pose (c' := fun p A => let (n,_) := c p A in n).
pose (p x := x = O \/ X).
assert (A:exists x, p x). exists O. left. reflexivity.
case_eq (c p A). intros n [C|C] D.
assert (D':c' p A = n). unfold c'. rewrite D. reflexivity.
pose (q x := x = S O \/ X).
assert (B:exists x, q x). exists (S O). left. reflexivity.
case_eq (c q B). intros m [E|E] F.
assert (F':c' q B = m). unfold c'. rewrite F. reflexivity.
right. intros G.
assert (H:p = q). apply pe. intros x. split; intros _; right; assumption.
cut (n = m). rewrite C. rewrite E. discriminate.
rewrite <- D'. rewrite <- F'. clear D D'. revert A. rewrite H.
```

```
intros A.
assert (AB:A = B). now apply PropE_PI, PredE_PropE.
rewrite AB. reflexivity.
tauto. tauto. Qed.
```

Exercise 15.4.1 Modify the proof above to prove that excluded middle follows from predicate extensionality and a choice operator on *bool*. (Hint: Exactly two changes are required.)

Exercise 15.4.2 Does *unit* have a choice operator?