Introduction to Computational Logic

Lecture Notes SS 2010

July 27, 2010

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

Contents

1 Introduction

1 Introduction

2 Functions and Types

This chapter introduces the calculus of constructions and the proof assistant Coq. The calculus of constructions is a syntactic system that provides functions and types in a very expressive format. With the calculus of constructions one can express mathematical and computational constructions, logical statements, and proofs. It will be fascinating to see how a calculus with so few primitives can serve so many concerns.

We will work with the proof assistant Coq. Coq is a widely used software system that implements a logical system extending the calculus of constructions. Coq has been used for large scale mathematical proofs like the four color theorem. Coq will be your personal teaching assistant for this course. Working with Coq is similar to working with an interactive programming system. With Coq the calculus of constructions turns into a computational reality. Run all the examples you see with Coq.

2.1 Values

The calculus of constructions is concerned with two disjoint kinds of **values** called **functions** and **types**. Every type is associated with a set of values called its **elements**, and every function is associated with a type called its **argument type**. A function assigns a unique value to each element of its argument type.

The functions of the calculus of constructions are different from set-theoretic functions. Functions of the calculus are more like computational procedures. Two functions of the calculus can be different although they have the same argument type and assign the same values to arguments.

There are two disjoint kinds of types, called **function types** and **universes**. The elements of function types are functions, and the elements of universes are types. Every function is an element of some function type, and every type is an element of some universe.

The universes are denoted by U_0, U_1, U_2, ... where there is a own universe U_n for every natural number n. For every $n \in \mathbb{N}$, we have that U_n is an element of U_{n+1}, and that every element of U_n is an element of U_{n+1}. Moreover, U_n is not an element of U_n. Using notation for sets, we have $\mathsf{U}_0 \in \mathsf{U}_1 \in \mathsf{U}_2 \in \cdots$ and $\mathsf{U}_0 \subsetneq \mathsf{U}_1 \subsetneq \mathsf{U}_2 \subsetneq \cdots$. One speaks of a **cumulative hierarchy**.

2.2 Terms

The syntactic expressions of the calculus of constructions are called **terms**. There is a decidable **typing relation** $s : t$ between two terms s and t. If a **typing** $s : t$ holds, we say that s **has type** t or that s **is of type** t. If s has type t, then t denotes a type and s denotes an element of t.

A term s is **well-formed** if there is a term t such that $s : t$. Only well-formed terms denote values. There is a **type checking algorithm** that determines whether a term is well-formed. For every well-formed term s the type checking algorithm computes a well-formed term t such that $s : t$. We call t **the principal type of** s or just **the type of** s.

Terms are obtained with the grammar

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

where U ranges over the universes $\mathsf{U}_0, \mathsf{U}_1, \mathsf{U}_2, \ldots$ and x ranges over symbols called **variables**. We state the following facts about terms.

1. Terms of the form $s\,t$ are called **applications**. A well-formed application $s\,t$ denotes the value the function denoted by s assigns to the value denoted by t.
2. Terms of the forms $\lambda x{:}s.t$ and $\forall x{:}s.t$ are called **abstractions**. An abstraction provides its **body** t with a **local variable** x. One says that the abstraction **binds** x in t. The term s specifies the type of x.
3. A well-formed term $\lambda x{:}s.t$ denotes a function. The term s must denote a type, and t must denote a value for every element x of s. The argument type of the function is s. To an element x of s the function assigns the value the term t denotes for x.
4. A well-formed term $\forall x{:}s.t$ denotes a function type. The terms s and t must both denote types. The elements of the function type are functions whose argument type is s, and that assign to an element x of s an element of the type the term t denotes for x.
5. A term of the form $\forall x{:}s.t$ may be written as $s \to t$ if x does not occur in t. Function types that can be written as $s \to t$ are called **simple**. Function types that cannot be written as $s \to t$ are called **dependent**.
6. If s and t denote types in some universe U_i, then $\forall x{:}s.t$ denotes a type in U_i. We say that the universes are *closed under taking function types*.
7. If s denotes a type and t denotes a type in U_0, then $\forall x{:}s.t$ denotes a type in U_0. We say that U_0 is *closed under quantification*.

To ease our language, we sometimes say just "s" when we really should say "the value denoted by the term s".

Because of the cumulative hierarchy a term may have more than one type. For instance, the term U_0 has the types U_1, U_2, and so on. The type of U_0 determined by the type checking algorithm is U_1.

A function that returns a function can be seen as a multi-argument function. This intuition motivates two notational conventions for terms that save parentheses:

$$s \to t \to u \quad \rightsquigarrow \quad s \to (t \to u)$$
$$s\ t\ u \quad \rightsquigarrow \quad (s\ t)\ u$$

We distinguish between **semantic objects** and **syntactic objects**. Values are seen as semantic objects, and terms are seen as syntactic objects. Syntactic objects serve as representations of semantic objects and can be realized on a computer. Semantic objects serve as intuitive objects that help our understanding of the calculus. The syntactic objects of the calculus will become very concrete once we start to work with Coq. In the mathematical development of the calculus the syntactic objects take priority over the semantic objects.

This section contains too much information for a single reading. The idea is that this section serves as a reference as you read through the chapter.

2.3 Type Checking

It's high time that we use Coq as a teaching assistant. Here are a few things you need to know about Coq's notation:

- *Prop* is Coq's notation for the lowest universe U_0.
- *Type* is Coq's common notation for the higher universes U_1, U_2, U_3, ...
- *fun* $x : s \Rightarrow t$ is Coq's notation for an abstraction $\lambda x : s.t$.
- *forall* $x : s,\ t$ is Coq's notation for an abstraction $\forall x : s.t$.

Coq behaves much like an interactive programming system. With the command *Check* one can submit a term and have it checked. If the term is well-formed in the current environment, Coq prints the type of the term. Otherwise, one gets an error message. One speaks of type checking because most of the checking concerns type conditions. We enter three terms that denote functions.

```
Check fun X : Prop => X.
```
Prop → Prop

```
Check fun X : Prop => fun x : X => x.
```
forall X : Prop, X → X

```
Check fun X : Prop => fun f : X -> X => fun x : X => f (f x).
```
forall X : Prop, (X → X) → X → X

Try to understand the derived types. It is not difficult. The first function is the identity function for *Prop*. The second function is a polymorphic identity function that given a type X from the universe *Prop* returns the identity function for X. The third function takes a type X : *Prop*, a function $f : X \to X$, and a value $x : X$ and returns the value obtained by applying f twice to x. The types of the second and third function are dependent.

We call a function **polymorphic** if it takes types as arguments. Dependent types make it possible to write interesting polymorphic functions.

Every term that is well-formed has a type. Thus there are types for terms that denote types. Here are examples.

```
Check Prop.
```
Type

```
Check Prop -> Prop.
```
Type

```
Check forall X : Prop, X -> X.
```
Prop

The reasoning for the last example is as follows: Since X has type *Prop*, $X \to X$ has type *Prop*; and since $X \to X$ has type *Prop*, $\forall X$: *Prop*. $X \to X$ has type *Prop*. Review statements (6) and (7) of §2.2 to see this.

You can use *Check* to verify whether two terms are related by the typing relation (see §2.2):

```
Check Prop : Type.
Check forall X : Prop, X  :  Prop.
Check forall X : Prop, X  :  Type.
Check (fun X : Type => X) (forall X : Prop, X) : Type.
Check (fun X : Type => X) (forall X : Prop, X) : Prop.
```

All typings but the last are confirmed.

Coq comes with a number of notational conveniences for nested abstractions. Here are examples. Make sure you can write each example just using the core notation.

```
Check fun (X : Prop) (f : X -> X) (x : X) => f (f x).
```
forall X : Prop, (X → X) → X → X

```
Check fun (X Y Z : Prop) (f : X -> Y) (g : Y -> Z) (x : X) => g (f x).
```
forall X Y Z : Prop, (X → Y) → (Y → Z) → X → Z

2.4 Parameters and Definitions

Coq can work with variables whose values are undetermined. Such variables are called **parameters**. We open a section *Getting_Started* and **declare** two local

parameters:

```
Section Getting_Started.
Variable A : Type.
Variable a : A.
```

The **declarations** tell Coq that A is a type and that a is an element of A. However, Coq does not know what kind of type A is and what kind of value a is. We can use the parameters A and a to construct terms:

```
Check fun x : A => a.
```
A → A

Coq also provides for **defined variables**:

```
Definition id : A -> A := fun x : A => x.
Definition twice : (A -> A) -> A -> A := fun  (f : A -> A) (x : A) => f (f x).
Check twice id (id a).
```
A

A **definition** declares a variable, gives it a type, and equates it with a term. This way the variable denotes the same value the term denotes. We say that a defined variable **names** the term it is equated with. When you enter a definition, Coq checks that everything is well-typed and that the term on the right has in fact the type the definition specifies for the variable.

2.5 Notational Conveniences and Type Inference

Coq comes with various notational conveniences for definitions.

```
Definition comp : (A -> A) -> (A -> A) -> A -> A := fun (f g : A -> A) (x:A) => g (f x).
```
comp : (A → A) → (A → A) → A → A

This is the standard form of a definition. All types are given.

```
Definition comp1 := fun (f g : A -> A) (x : A) => g (f x).
```
comp1 : (A → A) → (A → A) → A → A

This definition omits the type of the declared variable. Coq derives the type of the term at the right-hand side of the definition and assigns it to the variable.

```
Definition comp2 (f g : A -> A) (x : A) : A := g (f x).
```
comp2 : (A → A) → (A → A) → A → A

This definition gives all the types but combines the introduction of the argument variables with the type specification.

```
Definition comp3 (f g : A -> A) (x : A) := g (f x).
```
comp3 : (A → A) → (A → A) → A → A

This definition omits the specification of the result type. It is derived by Coq.

```
Definition comp4 : (A -> A) -> (A -> A) -> A -> A := fun f g x => g (f x).
```
comp4 : (A → A) → (A → A) → A → A

This definition omits the types for the argument variable of the term at the right-hand side. They are derived by Coq. One speaks of **type inference**. All five definitions equate their variable to the same term. You can check this with the **print command**.

```
Print comp4.
```
comp4 = fun (f g : A → A) (x:A) ⇒ g (f x)
: (A → A) → (A → A) → A → A

In case of doubt always use the print command for defined variables. It shows you the type of the variable and the term the variable is equated to.

When you enter terms you may introduce local variables without specifying a type. Coq will try to infer the missing types. Here are examples:

```
Check (fun x => x) a.
```
(fun x : A ⇒ x) a : A

```
Check (fun x f => f (f x)) a.
```
(fun (x : A) (f : A -> A) ⇒ f (f x)) a : (A → A) → A

```
Check (fun X (x : X) => x) A.
```
(fun (X : Type) (x : X) ⇒ x) A : A → A

```
Check (forall X, X -> X).
```
forall X : Type, X → X : Type

2.6 Alpha Renaming

A variable x is **free in a term** if it has an occurrence in the term that is not in the body of an abstraction that binds x. Here are examples:

- x is not free in $\lambda x : y . x$.
- x is free in $(\lambda x : y . x) x$.
- x is free in $\lambda x : x . x$.
- x is free in $x \to x$.
- x is not free in $\forall x : y . \; x \to x$.

A term is **closed** if no variable is free in it. Examples of closed terms are U_0, $\mathsf{U}_0 \to \mathsf{U}_0$, $\forall x:\mathsf{U}_0.\, x \to x$, and $\lambda X:\mathsf{U}_0.\, \lambda x:X.\, x$. Terms are called **open** if they are not closed. A term t is **closed up to defined variables** if every variable that is free in t is defined.

Abstractions that are equal up to a renaming of their local variable describe the same value. Here are examples:

- $\lambda x:X.\, x$ can be renamed into $\lambda y:X.\, y$.
- $\lambda x:X.\, \lambda y:X.\, f\, x\, y$ can be renamed into $\lambda z:X.\, \lambda y:X.\, f\, z\, y$.
- $\lambda x:X.\, \lambda y:X.\, f\, x\, y$ *cannot* be renamed into $\lambda y:X.\lambda y:X.\, f\, y\, y$.

Given a term, we may rename the local variable of an abstraction that occurs in the term as a subterm. We speak of **alpha renaming**. Two terms are **alpha equivalent** if they are equal up to finitely many alpha renaming steps. Here are examples:

- $\lambda x:X.\, x$ and $\lambda y:X.\, y$ are alpha equivalent.
- $\lambda x:X.\, \lambda y:X.\, f\, x\, y$ and $\lambda x:X.\, \lambda z:X.\, f\, x\, z$ are alpha equivalent.
- $\lambda x:X.\, \lambda y:X.\, f\, x\, y$ and $\lambda y:X.\, \lambda x:X.\, f\, y\, x$ are alpha equivalent.
- $\lambda x:X.\, \lambda y:X.\, f\, x\, y$ and $\lambda y:X.\, \lambda x:X.\, f\, x\, y$ are not alpha equivalent.

One says that two terms are **equal up to alpha renaming** if they are alpha equivalent. In fact, alpha equivalent terms can be considered equal for almost most purposes. In particular, value denotation and well-formedness are invariant under alpha renaming. When Coq prints a term, it may take the freedom to alpha rename it. For instance:

```
Check fun (x : Prop) (x : x) => x.
fun (x : Prop) (x0 : x) ⇒ x0  :  forall x : Prop, x → x
```

If the local variable of an abstraction does not occur in its body, one may write the **underline character** in place of the variable:

```
Check fun (X : Type) (x : X) (_ : X) => x.
forall X : Type, X → X → X
```

Exercise 2.6.1 Try the command *Check fun X Y : Prop => forall x : X, Y.* It prints the given term as *fun X Y : Prop ⇒ X → Y*. Explain why this is correct.

2.7 Reduction and Normal Forms

A term of the form $(\lambda x:s.t)\, u$ is called a **beta redex**. It represents the application of an explicitly given function to an argument. A beta redex $(\lambda x:s.t)\, u$ can be simplified to the term t^x_u where t^x_u is obtained from t by replacing every free

occurrence of the variable x with the term u. We speak of a **beta reduction**. Clearly, a beta redex $(\lambda x:s.t)\,u$ and the term t^x_u obtained by beta reduction denote the same value. We can use the command *Eval cbv beta in* to make Coq perform beta reductions:

```
Eval cbv beta in (fun x : A => x) a.
```
```
a
```
```
Eval cbv beta in (fun x y : A => y) a.
```
```
fun y : A ⇒ y
```
```
Eval cbv beta in (fun x y : A => x) (id a).
```
```
fun _ : A ⇒ id a
```
```
Eval cbv beta in (fun (x y : A) (f : A -> A -> A) => f x y) (id a).
```
```
fun (y : A) (f : A → A → A) ⇒ f (id a) y
```
```
Eval cbv beta in (fun f x => f x) (fun y : A => y).
```
```
fun x : A ⇒ x
```
```
Eval cbv beta in (fun X (x : X) => x) A.
```
```
fun x : A ⇒ x
```

When we reduce a beta redex $(\lambda x:s.t)\,u$ where s contains an abstraction whose local variable x is free in u, we must make sure that the abstraction does not **capture** the free occurrence of x in u. This can be done by renaming the local variable of the abstraction. Here is an example:

```
Eval cbv beta in (fun (x a : A) (f : A -> A -> A) => f x a) a.
```
```
fun (a0 : A) (f : A → A → A) ⇒ f a a0
```

Besides beta reduction Coq also implements delta reduction. **Delta reduction** simply replaces a defined variable with the term it is equated to.

Beta and delta reduction provide a logical form of computation. Given a term, one can perform beta and delta reduction steps until no further reduction step is possible. The calculus of constructions is carefully designed such that this process always terminates. The reduction steps can be applied in any order, and the term finally obtained is unique up to alpha renaming.

Let s and t be well-formed terms such that $s : t$. If we apply a reduction step to s, we always obtain a well-formed term s' such that $s' : t$ and s' denotes the same value as s. One says that reduction preserves types and denotations.

We say that a term s **reduces** to a term t if t is alpha equivalent to some term that can be obtained from s by beta and delta reduction steps. This definition is to be understood such that s reduces to t also if s and t are alpha equivalent (no reduction step applied). If s reduces to t and s' is alpha equivalent to s, then s' also reduces to t.

A term is **normal** if no reduction step applies to it. This is the case if the term contains neither a beta redex nor a defined variable. A term s is a **normal form** of a term t if s is normal and t reduces to s. Since reduction with beta and delta steps always terminates, every term has a normal form. In fact, a term has a unique normal form up to alpha renaming. To ease our language, we often speak of the normal form of a term.

Two terms are **convertible** if they have a common normal form. Since reduction preserves the denotation of a term, convertible terms denote the same value. If s is well-formed and reduces to t, then s and t are convertible.

In Coq, we can obtain the normal form of a term with the command *Eval cbv in*:

```
Eval cbv in id a.
a
Eval cbv in twice id.
fun x : A ⇒ x
Eval cbv in fun x => id (twice id (id x)).
fun x : A ⇒ x
```

Let us summarize and name the key properties of reduction in the calculus of constructions:

- **Termination** For every well-formed term, reduction with beta and delta steps terminates. In the literature, the termination property is often referred to as *strong normalization*.
- **Confluence** If a term s reduces to two terms t_1 and t_2, then there is always a term u such that both t_1 and t_2 reduce to u. Confluence is a variant of a property known as *Church-Rosser*.
- **Type Preservation** If s has type t and s reduces to u, then u is always well-formed and has type t. Type presevation is also known as the *subject reduction* property.

Taken together, the three properties guarentee that every well-formed term of type t has a normal form of the same type that is unique up to alpha renaming.

A precursor of the calculus of constructions is the **untyped lambda calculus** studied by Church in the 1930s. The terms of the untyped lambda calculus are obtained with the grammar $s ::= x \mid \lambda x.s \mid ss$. Consider the term $\omega := \lambda x.xx$ and note that the term $\omega\omega$ reduces in one beta step to itself. From this we learn that one needs types to make beta reduction terminating. In 1940, Church presented the simply typed lambda calculus, which has simple function types and no universes. It took until the late 1960s until termination of the simply typed lambda calculus was shown by Tait and others.

2.8 Closing a Section

It's time to close the section *Getting_Started* we opened in §2.4.

```
End Getting_Started.
```

This eliminates the local parameters A and a. However, all definitions survive in generalized form:

```
Check id.
forall A : Type, A → A
Check comp.
forall A : Type, (A → A) → (A → A) → A → A
```

2.9 Computational Interpretation

A **canonical term** is a well-formed term that is closed and normal. The type structure of the calculus of constructions is set up such that a canonical term is either an abstraction $\lambda x{:}s.\,t$ or an abstraction $\forall x{:}s.\,t$ or a universe U_n. Thus a canonical term denotes either a function or a function type or a universe. Note that every well-formed term that is closed up to defined variables reduces to a canonical term that is of the same type and denotes the same value.

The **computational interpretation** of the calculus of constructions is determined by two assumptions:

1. Every value is denoted by some canonical term.
2. Two canonical terms that are not alpha equivalent denote different values.

The values of the computational interpretation can be obtained from the canonical terms by taking the quotient with respect to alpha renaming. Given the computational interpretation, we can identify values and canonical terms up to alpha renaming.

So let the computational interpretation be in force. Here are three types:

```
Definition void : Prop := forall X : Prop, X.
Definition unit : Prop := forall X : Prop, X -> X.
Definition bool : Prop := forall X : Prop, X -> X -> X.
```

One can show that *void* has no element, *unit* has exactly one element, and *bool* has exactly two elements. Here are the elements of *unit* and *bool*:

```
Definition I : unit := fun _ x => x.
Definition true : bool := fun _ x _ => x.
Definition false : bool := fun _ _ y => y.
```

Note that we make use of type inference. Use the print cammand to see the complete terms.

Exercise 2.9.1 Convince yourself that you cannot find further elements of *void*, *unit*, and *bool*.

Exercise 2.9.2 Consider the term $(\lambda x:\mathsf{U}_1.x)(\forall x:\mathsf{U}_0.x)$. This term has the type U_1. It reduces to the normal term $\forall x:\mathsf{U}_0.x$, which has the type U_0. Consider the following commands:

```
Definition foo := fun X : Type => X.
Definition arg := forall X : Prop, X.
Check foo.
Check arg.
Definition test : Prop := foo arg.
```

Explain why Coq rejects the definition of *test*.

2.10 Church Numerals

There is a type whose elements are in one-to-one correspondence with the natural numbers:

```
Definition nat : Prop := forall X : Prop, X -> (X -> X) -> X.
Definition zero : nat := fun _ x f => x.
Definition one : nat := fun _ x f => f x.
Definition two : nat := fun _ x f => f (f x).
Definition three : nat := fun _ x f => f (f (f x)).
```

Note that the number n is represented as the polymorphic function that takes a value x and a function f and applies f n times to x. The functional representation of the natural numbers was discovered by Church in the 1930s in the untyped setting of the lambda calculus. The functions representing the natural numbers are known as **Church numerals**. We can say that Church numerals represent natural numbers as iterators.

It is now straightforward to define functions that compute successors, sums, products, and powers of Church numerals:

```
Definition succ (n : nat) : nat := fun X x f => n X (f x) f.
Definition add (m n : nat) : nat := m nat n succ.
Definition mul (m n : nat) : nat := m nat zero (add n).
Definition exp (m n : nat) : nat := n nat one (mul m).
Eval cbv in exp two three.
fun (X : Prop) (x : X) (f : X → X) ⇒ f (f (f (f (f (f (f (f x)))))))
```

We can also write a function that tests whether a numeral is zero:

```
Definition iszero (n : nat) : bool := n bool true (fun x => false).
```

It takes a new idea to write a function that yields the predecessor of a numeral. Given a number n, we can apply the function

$$(m, k) \mapsto (m + 1, m)$$

n times to the pair $(0, 0)$. For $n = 3$ this yields

$$(0, 0) \mapsto (1, 0) \mapsto (2, 1) \mapsto (3, 2)$$

Clearly, the second component of the pair so obtained is the predecessor of n. For 0 we get 0 as predecessor. It remains to find a representation for pairs of numerals. Again the idea is due to Church.

```
Definition pair : Prop := forall X : Prop, (nat -> nat -> X) -> X.
Definition cons (m n : nat) : pair := fun _ f => f m n.
Definition fst (p : pair) : nat := p nat (fun x _ => x).
Definition snd (p : pair) : nat := p nat (fun _ y => y).
Definition pred (n : nat) : nat := snd (n pair (cons zero zero)
                                       (fun p => cons (succ (fst p)) (fst p))).
```

Exercise 2.10.1 Write a function that subtracts two numerals.

Exercise 2.10.2 Write a function that computes the factorial of a numeral. Do not use the predecessor function but work directly with pairs.

Exercise 2.10.3 Here is an alternative representation for pairs of naturals:

```
Definition pair' : Prop := (nat -> nat -> nat) -> nat.
```

a) Define functions *cons′*, *fst′*, *snd′* that construct and decompose pairs.

b) In contrast to *pair*, *pair′* has **unwanted elements** that cannot be obtained with *cons′*. Define a function $unwanted : nat \to pair'$ that yields a different unwanted element for every Church numeral.

Exercise 2.10.4 Consider the following variant of the type *nat*:

```
Definition nat' : Prop := forall X : Prop, (X -> X) -> X -> X.
```

It has the problem that it has two elements that represent 1. Find the extra element.

2.11 Unspecified and Implicit Arguments

We can define polymorphic pairs as follows:

```
Definition Pair (X Y : Prop) : Prop
:= forall Z : Prop, (X -> Y -> Z) -> Z.
Definition Cons' (X Y : Prop) : X -> Y -> Pair X Y
:= fun x y _ f => f x y.
Definition Fst' (X Y : Prop) : Pair X Y -> X
:= fun p => p X (fun x _ => x).
Definition Snd' (X Y : Prop) : Pair X Y -> Y
:= fun p => p Y (fun _ y => y).
```

We can now describe the pair that consists of *one* and *true*.

```
Check Cons' nat bool one true.
```
Cons' nat bool one true : Pair nat bool

Coq can automatically derive type arguments that are determined by other arguments. The first and second argument of *Cons'* are derivable. Arguments that are derivable can be given as an underline "_".

```
Check Cons' _ _ one true.
```
Cons' nat bool one true : Pair nat bool

Arguments that are given with underline are called **unspecified arguments**. Here is the definition of a function that given a pair returns the pair obtained by swapping the components.

```
Definition swap' (X Y : Prop) : Pair X Y -> Pair Y X
:= fun p => Cons' _ _ (Snd' _ _ p) (Fst' _ _ p).
```

You can use the print command to see which types Coq derives for the unspecified arguments.

We can go one step further and get rid of the underlines for the type arguments. To this purpose we define the operations for pairs such that the type arguments are implicit. This can be done by specifying the type arguments with curly braces.

```
Definition Cons {X Y : Prop} : X -> Y -> Pair X Y
:= fun x y _ f => f x y.
Definition Fst {X Y : Prop} : Pair X Y -> X
:= fun p => p X (fun x _ => x).
Definition Snd {X Y : Prop} : Pair X Y -> Y
:= fun p => p Y (fun _ y => y).
```

```
Check Cons one true.
```
Cons one true : Pair nat bool

```
Definition swap {X Y : Prop} : Pair X Y -> Pair Y X
:= fun p => Cons (Snd p) (Fst p).

Eval cbv in fun (X : Prop) (x y : X) => Fst (swap (Cons x y)).
fun (X : Prop) (_ y : X) ⇒ y
```

Arguments that are specified with curly braces are called **implicit arguments**. When a variable x is defined with implicit arguments, Coq's frontend replaces every occurrence of x in a term by the underspecified term $@x\ \_\ldots\_$ where every implicit argument contributes an underline and $@x$ is the name for the real variable. For instance, the term *Cons* by itself does not type check, but *@Cons* does.

```
Check @Cons.
forall X Y : Prop, X → Y → Pair X Y
```

Keep in mind that unspecified and implicit arguments are just notational devices that don't affect the logical representation of terms.

2.12 Typing Rules

We will now say more about the typing relation (see §2.2). We do this by means of typing rules. Each typing rule says that certain typings $s : t$ hold if certain conditions are satisfied. The typing rules are complete in the sense that every typing that holds can be established with the typing rules. We start with an obvious typing rule for universes (see §2.1).

$$\textbf{Uni}\quad \frac{}{\mathsf{U}_n : \mathsf{U}_{n+1}}$$

The rule says that U_n has type U_{n+1} for every natural number n. The next rule concerns the typing of λ-abstractions.

$$\textbf{Lam}\quad \frac{s : \mathsf{U}_n \qquad x : s \Rightarrow t : u}{\lambda x{:}s.\,t : \forall x{:}s.\,u}$$

It says that a term $\lambda x{:}s.\,t$ has type $\forall x{:}s.\,u$ if s has type U_n for some n and t has type u under the assumption that the variable x has type s. The next rule concerns the typing of applications.

$$\textbf{App}\quad \frac{u : \forall x{:}s.\,t \qquad v : s}{uv : t_v^x}$$

It says that an application uv has type t_v^x if u has type $\forall x{:}s.\,t$ and v has type s.

We give two rules that describe the typing of ∀-abstractions. The rules correspond to items (6) and (7) in §2.2.

$$\textbf{Fun}\quad \frac{s : \mathsf{U}_n \qquad x : s \Rightarrow t : \mathsf{U}_n}{\forall x{:}s.\, t : \mathsf{U}_n} \qquad\qquad \textbf{Prop}\quad \frac{s : \mathsf{U}_n \qquad x : s \Rightarrow t : \mathsf{U}_0}{\forall x{:}s.\, t : \mathsf{U}_0}$$

Rule Prop sets U_0 apart from the higher universes by stating that U_0 is closed under taking function types whose argument type is in an arbitrary universe. One says that U_0 is **impredicative** and that the higher universes are **predicative**. The impredicativity of U_0 was exploited in §2.10 when we defined the arithmetic operations on Church numerals (e.g., *add*). Making one of the higher universes impredicative or collapsing the hierarchy with $\mathsf{U}_0 : \mathsf{U}_0$ is not an option since it would forsake termination of reduction and the existence of normal forms.

The remaining typing rules concern alpha renaming, reduction, and the cumulativity of the universes (i.e., $\mathsf{U}_0 \subseteq \mathsf{U}_1 \subseteq \mathsf{U}_2 \subseteq \cdots$).

$$\textbf{Pre}\quad \frac{s : t}{s' : t'}\quad s \text{ alpha equivalent to } s' \text{ and } t \text{ reduces to } t'$$

Rule Pre says that the validity of a typing is preserved by alpha renaming on either side and by reduction on the right-hand side (recall that reduction subsumes alpha renaming). The next rule says that the validity of a typing is preserved under inverse reduction of the right-hand side:

$$\textbf{Red}\quad \frac{t : \mathsf{U}_n \qquad s : t'}{s : t}\quad t \text{ reduces to } t'$$

Rule Red tells us that we can establish the validity of a typing $s : t$ as follows: First establish $t : \mathsf{U}_n$ for some n, then reduce t to some t', and finally establish $s : t'$. This rule is essential when we work with defined types (e.g., *nat* in §2.10).

Finally, there is a rule that accounts for the cumulativity of the universes:

$$\textbf{Cum}\quad \frac{s : t}{s : t'}\quad t \prec t'$$

The **subtyping relation** employed by the rule is defined as follows:

1. $\mathsf{U}_m \prec \mathsf{U}_n$ if $m < n$.
2. $\forall x{:}s.\, t \prec \forall x{:}s.\, t'$ if $t \prec t'$.

The typing rules constitute a definition of the typing relation. Given this definition, one can prove the following properties:

1. If $s : t$, then $t : \mathsf{U}_n$ for some n.
2. If $s : t$ and s reduces to s', then $s' : t$ (type preservation).

2.13 Remarks

The original version of the calculus of constructions was formulated in the 1980s by Gérard Huet and his student Thierry Coquand. Systems with dependent function types where first studied in the early 1970s by Jean-Yves Girard and Per Martin-Löf. In 1971, Martin-Löf presented a system with a single universe $\mathsf{U}_0 : \mathsf{U}_0$. Girard showed that the assumption $\mathsf{U}_0 : \mathsf{U}_0$ leads to a non-terminating system. Martin-Löf then came up with the cumulative hierarchy of universes to have a terminating system where every type has a type. The original calculus of constructions had just the universes U_0 and U_1 and did not give a type to U_1. The hierarchy of universes was added by Zhaohui Luo in his extended calculus of constructions.

The notion of type was invented by Bertrand Russell in 1908 to avoid the inconsistencies (paradoxes) present in Georg Cantor's set theory and and Gottlob Frege's Begriffsschrift.

Work on Coq started in the 1980s under the direction of Gérard Huet. Two important precursors of Coq are LCF and Automath. The LCF proof assistant was developed in the 1970s under the lead of Robin Milner. LCF is based on a simple type theory without dependent types. The Automath project started in the late 1960s under the lead of Nicholas de Bruijn. The Automath project designed and implemented a dependently typed language to represent mathematics and to computer-verify mathematical proofs.

The programming language ML, which pioneered polymorphic types and type inference, was designed by Robin Milner in the 1970s for use with LCF. At this time Milner did not know about the work of Martin-Löf and Girard. Coq is implemented in OCaml, a variant of the ML language.

3 Natural Deduction

In this chapter we take a first look at propositions and proofs, two notions familiar from mathematics. A proposition is a statement that may be true or false (e.g., "there are infinitely many prime numbers"), and a proof is a rigorous argumentation that a statement is true. We are interested in those apects of propositions and proofs that are not specific to a particular mathematical theory (e.g., number theory). These aspects are known as *logical aspects* and are organized around syntactical operations called *logical operations*.

The title of the chapter, natural deduction, refers to the proof theory we outline in this chapter. Natural deduction originated with the work of Gerhard Gentzen in the early 1930s and was conceived as a formulation of the basic rules of mathematical reasoning. In a fascinating process of discovery that started in the 1960s and led to the calculus of constructions, it turned out that propositions and proofs as modeled by natural deduction are intimately linked with types and functions in computational systems.

This chapter outlines our view of propositions and proofs so that you have an idea where we want to go. The necessary technical details and careful explanations will follow in later chapters. You will profit from rereading this chapter when you work through the technical chapters.

3.1 Propositions and Logical Operations

A proposition is a statement that expresses a property. In mathematics one assumes that a proposition is either true or false. Propositions are typically expressed as a mix of natural language and symbolic notation. The structure of propositions is determined by logical operations that combine propositions into more complex propositions. Here is a summary of commonly used **logical operations**:

- *Conjunction* $s \wedge t$ s is true and t is true
- *Disjunction* $s \vee t$ s is true or t is true
- *Implication* $s \to t$ if s is true, then t is true
- *Negation* $\neg s$ s is not true
- *Equivalence* $s \leftrightarrow t$ s is true if and only if t is true

- *Universal quantification* $\forall x : s.\, t$ t is true for every x in s
- *Existential quantification* $\exists x : s.\, t$ t is true for some x in s

For a systematic study of propositions it is useful to fix two **logical constants** that provide canonical notations for the two **truth values**:

- *Falsity* $\bot$ false
- *Truth* $\top$ true

To ease our language, we count $\bot$ and $\top$ as logical operations. We also count equality as a logical operation:

- *Equality* $s = t$ s equals t

The above list gives you an idea of the meaning of the logical operations we consider. Your mathematical experience will provide you with intuitions and examples. As we continue, we will become more precise.

Syntactically, propositions will be modelled as terms and logical operations will appear as functions (with the exception of $\bot$ and $\top$). In fact, we will represent propositions in the calculus of constructions as terms of type *Prop*. Quantifications will be represnted with abstractions so that the local variables of quantifications will appear as local variables of abstractions.

For now, we will not say more about the notion of truth. Instead, we will say more about proofs.

3.2 Proof Rules for Implications

We are interested in a system of proof rules that establishes a notion of provability that is faithful to mathematical reasoning. We require that the proof rules are based on the syntactic structure of propositions and do not make use of the notion of truth. We are aiming at a system that can be realized on a computer. There should be algorithms that check whether a proposition is well-formed and whether the steps of a proof are validated by a given set of proof rules.

A proof rule is a syntactical device for establishing the truth of propositions. Here is a prominent example of a proof rule:

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

This rule is known as **modus ponens** and was already formulated by the old Greeks. It says that t is true if both $s \to t$ and s are true. Read differently, it says that t is provable if both $s \to t$ and s are provable. We now see that we can establish a notion of provability by fixing a set of proof rules. To find the right proof rules, we can look at mathematical proofs and rely on our intuitive idea of truth.

If we look at the modus ponens rule, we see that it makes use of an already proven implication $s \to t$. Thus modus ponens tells us how to make use of implications. What we need in addition is a rule that allows us to establish the provability of an implication. This rule takes the form

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

It says that an implication $s \to t$ can be proven by assuming that the premise s is provable and by showing that the conclusion t is provable. So the task of proving $s \to t$ is reduced to the task of proving t under the assumption that s is provable.

We now have two proof rules for implications. The rule that introduces the premise of an implication as an assumption is called an **introduction rule**, and the rule that applies an implication (modus ponens) is called an **elimination rule.** Here is an example that shows how the two rules can prove a complex implication.

$$\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{x}{X}\,6 \quad f}{Y}\,5 \quad g}{Z}\,4}{X \to Z}\,3 \quad x : X}{(Y \to Z) \to (X \to Z)}\,2 \quad g : Y \to Z}{(X \to Y) \to (Y \to Z) \to (X \to Z)}\,1 \quad f : X \to Y$$

Its best to read the proof backwards.

0. Show $(X \to Y) \to (Y \to Z) \to (X \to Z)$.
1. Assume that f is a proof of $X \to Y$. Show $(Y \to Z) \to (X \to Z)$.
2. Assume that g is a proof of $Y \to Z$. Show $X \to Z$.
3. Assume that x is a proof of X. Show Z.
4. Since g is a proof of $Y \to Z$, it suffices to prove Y.
5. Since f is a proof of $X \to Y$, it suffices to prove X.
6. Since x is a proof of X, we are done.

Note that steps (1), (2), and (3) are validated by the introduction rule for implications, and that steps (4) and (5) are validated the elimination rule for implications.

We will see that proofs can be represented as terms in the calculus of constructions. The above proof will be represented as the following term:

```
fun (f : X -> Y) (g : Y -> Z) (x : X) => g (f x)
```

Relate the term with the verbal version of the proof. The term introduces the assumptions with lambda abstractions where the argument variables serve as names of the assumptions and the types serve as propositions. Modus ponens is realized in a natural way as function application. The term is a more compact presentation of the proof than the verbal presentation. Moreover, the term is a formal representation of the proof that can be checked and manipulated by a machine.

3.3 Basic Intuitionistic Proof Rules

It turns out that the two rules we have given for implications fully characterize provability of implications. We can now look at the other logical operations and think about their basic proof rules. It turns out that we can characterize provability of each logical operation with just a few proof rules. For some logical operations it is preferable to define them in terms of other logical operations:

$$\begin{aligned} \top &:= \bot \to \bot \\ \neg s &:= s \to \bot \\ s \leftrightarrow t &:= (s \to t) \wedge (t \to s) \end{aligned}$$

Note that negation is defined with $\to$ and $\bot$. This means that we prove $\neg s$ by assuming s and proving $\bot$. Since there is no direct proof of $\bot$, we can prove $\neg s$ only if the assumption s leads to a contradiction.

The basic proof rules for the logical operations that are not accomodated by definition appear in Figure 3.1. These rules are known as the **basic intuitionistic proof rules**. Each line gives the rules for one particular logical operation. The **introduction rules** appear on the left and the **elimination rules** appear on the right. With the exception of $\bot$ and $\vee$, each logical operation appearing in the figure has one introduction rule and one elimination rule. There are two introduction rules for $\vee$, and there is no introduction rule for $\bot$. The elimination rule for $\bot$ says that every proposition is provable if $\bot$ is provable. We need such a strong elimination rule for $\bot$ so that we can express negation with $\to$ and $\bot$. The notation t_v^x is to be understood as in the calculus of constructions: It denotes the term that is obtained from t by replacing every free occurrence of the variable x with the term v. The rules in Figure 3.1 are accompanied by two additional proof rules:

1. A proposition s is provable if there is an assumption s.
2. A proposition s is provable if it is provable after alpha renaming.

In addition, one needs typing rules that establish the premise $v : s$ of the elimination rule for $\forall$ and the introduction rule for $\exists$.

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

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

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

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

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

$$\frac{v : s \quad t_v^x}{\exists x : s.\, t} \qquad \frac{\exists x : s.\, t \quad x : s, t \Rightarrow u}{u}\ x \notin \mathrm{FV}\, u$$

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

$x \notin \mathrm{FV}\, u$ means that x is not free in u

Figure 3.1: Basic intuitionistic proof rules

3.4 Quantification over Propositions

In mathematics one quantifies over numbers, vectors, functions and other mathematical objects. It also makes sense to quantify over propositions. Quantification over propositions makes it possible to express proof rules as propositions. This feature gives a tremendous boost to a proof system since it is now possible to use as many proof rules as one likes. To make a new proof rule available, one formulates it as a proposition and proves it. Proving the proposition formulating the rule establishes that the rule is derivable from the basic rules. As an example,

we consider a proof rule known as **modus tollens**:

$$\frac{s \to t \qquad \neg t}{\neg s}$$

The rule translates into the following proposition:

$$\forall X\, Y : \mathit{Prop}.\ (X \to Y) \to (\neg Y \to \neg X)$$

Note that the letters s and t in the rule represent propositions while the letters X and Y in the proposition represent variables. Here is a proof of the proposition.

$$\dfrac{\dfrac{\dfrac{\dfrac{g(fx)}{X \to \bot}\ x : X}{\neg Y \to \neg X}\ g : Y \to \bot}{(X \to Y) \to (\neg Y \to \neg X)}\ f : X \to Y}{\forall X\, Y : \mathit{Prop}.\ (X \to Y) \to (\neg Y \to \neg X)}\ X\, Y : \mathit{Prop}$$

With quantification over propositions it is possible to express the logical operations $\bot$, $\wedge$, $\vee$, and $\exists$ with the operations $\forall$ and $\to$ in such a way that one can derive their basic intuitionistic proof rules.

$$\begin{aligned}
\bot &:= \forall X : \mathit{Prop}.\ X \\
s \wedge t &:= \forall X : \mathit{Prop}.\ (s \to t \to X) \to X \\
s \vee t &:= \forall X : \mathit{Prop}.\ (s \to X) \to (t \to X) \to X \\
\exists x : s.\, t &:= \forall X : \mathit{Prop}.\ (\forall x : s.\, t \to X) \to X
\end{aligned}$$

The trick consists in expressing a proposition u built with a logical operation o as the elimination rule for o specialized to u. This means that the reduction to $\forall$ and $\to$ is already present in the elimination rule for o.

3.5 Leibniz Equality

Two objects x and y are equal if every property that holds for x also holds for y. This characterization of equality is attributed to Leibniz. The asymmetry of the characterization is balanced by the presence of sufficiently strong properties. We can express equality with $\forall$ and $\to$ if we can quantify over predicates (functions that yield propositions):

$$s = t \ := \ \forall p : u \to \mathit{Prop}.\ pt \to ps \qquad \text{where } s : u$$

If a logical system provides equality through the above definition, one says that the system provides Leibniz equality.

We now see that it suffices to give a proof system for just the logical operations $\rightarrow$ and $\forall$, provided one can quantify over propositions and predicates. This observation simplifies the construction of proof systems considerably.

3.6 Representation in the Calculus of Constructions

If we now look at the calculus of constructions, we realize that the terms of type *Prop* give us all the propositions we could ask for. In addition we make the discovery that given a proposition t every term s of type t can be seen as a proof of t that is validated by the basic intuitionistic proof rules for $\rightarrow$ and $\forall$.

1. A variable represents the use of an assumption.
2. A lambda abstraction $\lambda x{:}s.t$ represents an application of an introduction rule where $x{:}s$ is the assumption made and t is the remaining proof. This can be seen from the typing rule for lambda abstractions (§2.12).

$$\text{Lam}\quad \frac{s : \mathsf{U}_n \qquad x : s \Rightarrow t : u}{\lambda x{:}s.\,t : \forall x{:}s.\,u}$$

3. An application st represents an application of an elimination rule. This becomes clear if we look at the typing rule for applications (§2.12).

$$\text{App}\quad \frac{u : \forall x{:}s.\,t \qquad v : s}{uv : t_v^x}$$

3.7 Remarks

The systematic study of popositions and proofs started in the second half of the 19th century with George Boole and Gottlob Frege. The notion of type was invented by Bertrand Russell in 1908 to avoid the inconsistencies present in Cantor's set theory and in Frege's Begriffsschrift. In the early 1930s, Gehard Gentzen developed the first system of natural deduction. His system is first order in that it can only quantify over a single type of individuals. The development of higher order systems that can quantify over propositions and functions started with Prawitz in the 1960s. The relationship between natural deduction and typed lambda terms was discoved by Curry and Howard. The technical development of this discovery was first pursued by Martin-Löf and Girard and finally led to the development of the calculus of constructions.

4 Propositions and Proofs

We will now see that the calculus of constructions is an expressive logical system. Propositions (i.e., logical statements) are represented as terms that denote function types, and proofs (i.e., logical argumentations) are represented as terms that denote functions. Given this representation, a proof s proves a proposition t if and only if s has type t. Proof checking is thus obtained as type checking.

4.1 The Logic

A **proposition** is a term that has type *Prop*. A **proof** is a term s such that there is a proposition t such that s has type t. In this case we say that s **proves** t or that s **is a proof of** t. A proposition t is **provable** if there is a proof s that proves t. Propositions of the form $s \to t$ are called **implications**, and propositions of the form $\forall x{:}s.\,t$ are called **universal quantifications** if x is free in t.

If we need the word proof with its general meaning, we will refer to the proofs of the calculus of constructions as **proof terms**.

Given our definitions, the typing rules of the calculus of constructions turn into proof rules. The typing rule for applications, for instance, yields a proof rule for universal quantifications:

$$\frac{u : \forall x{:}s.\,t \qquad v : s}{uv : t_v^x}$$

It tells us that given a proof of a proposition $\forall x{:}s.\,t$ and a term v of type s, we can obtain a proof of the proposition t_v^x. As a special case of this rule we have a proof rule for implications:

$$\frac{u : s \to t \qquad v : s}{uv : t}$$

It tells us that given proofs of the propositions $s \to t$ and s, we can obtain a proof of the proposition t. A simplified version of this rule

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

that omits the proofs is known as **modus ponens** and dates back to the ancient Greeks. It should be read as follows: If $s \to t$ is provable and s is provable, then t is provable.

4.2 First Steps

We start with a proposition that is easy to prove:

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

True states that every proposition that is provable is provable. With the definition

```
Definition I : True
:= fun _ x => x.
```

we make Coq check that I is defined as a proof of *True*. Note that we rely on type inference. We can ask Coq for the full proof:

```
Print I
```
```
I = fun (X : Prop) (x : X) ⇒ x
```

Here are proofs of interesting properties of implication.

```
Definition K {X Y : Prop} : X -> Y -> X
:= fun x y => x.

Definition C {X Y Z : Prop} : (X -> Y) -> (Y -> Z) -> X -> Z
:= fun f g x => g (f x).

Definition com {X Y Z : Prop} : (X -> Y -> Z) -> Y -> X -> Z
:= fun f y x => f x y
```

Proof K tells us that $Y \to X$ is provable if X is provable. Proof C tells us that $X \to Z$ is provable if $X \to Y$ and $Y \to Z$ are provable. Proof *com* tells us that $Y \to X \to Z$ is provable if $X \to Y \to Z$ is provable. This means that it doesn't matter for provability how we order the **premises** $X_1, \ldots, X_n$ of an implication $X_1 \to \cdots \to X_n \to X$.

Every proposition can be seen as a proof rule, and a proof of a proposition establishes the soundness of the corresponding proof rule. The proofs I, K, C, and *com* establish the soundness of the following proof rules (we use $\top$ as notation for *True*):

$$\frac{}{\top} \qquad \frac{s}{t \to s} \qquad \frac{s \to t \qquad t \to u}{s \to u} \qquad \frac{s \to t \to u}{t \to s \to u}$$

Given a rule, we call the propositions above the line **premises**, and the proposition below the line **conclusion**. Note that a soundness proof for a proof rule describes a function that constructs a proof of the conclusion of the rule from proofs of the premises of the rule.

We use proof rules to convey an operational understanding of propositions. There is no need to be precise about proof rules. If we want to be precise, we look at the proposition and its proof, not at the proof rule.

Exercise 4.2.1 Prove the following propositions with Coq.

a) $\forall\, X\, Y : \mathit{Prop}.\ X \to (X \to Y) \to Y$

b) $\forall\, X\, Y : \mathit{Prop}.\ (X \to X \to Y) \to X \to Y$

c) $\forall\, X\, Y\, Z : \mathit{Prop}.\ (X \to Y \to Z) \to (X \to Y) \to X \to Z$

d) $\forall\, X\, Y : \mathit{Prop}.\ X \to Y \to \forall\, Z : \mathit{Prop}.\ (X \to Y \to Z) \to Z$

e) $\forall\, X\, Y : \mathit{Prop}.\ (\forall\, Z : \mathit{Prop}.\ (X \to Y \to Z) \to Z) \to X$

f) $\forall\, X\, Y : \mathit{Prop}.\ X \to \forall\, Z : \mathit{Prop}.\ (X \to Z) \to (Y \to Z) \to Z$

4.3 Falsity and Negation

We define *False* as the canonical example of a proposition that is not provable:[1]

```
Definition False : Prop := forall X : Prop, X.
```

It is common to refer to *False* as **falsity**. The characteristic property of falsity is that every proposition is provable if falsity is provable. Here is a proof of the characteristic property:

```
Definition False_elim {X : Prop} : False -> X
:= fun f => f X.
```

We define **negation** as follows:

```
Definition not (X : Prop) : Prop := X -> False.
```

If s is a proposition, the negated proposition *not s* is provable if we can obtain a proof of falsity from a proof of s. Given the fact that falsity is unprovable, a proof of *not s* establishes the fact that s is unprovable. Here are proofs that establish basic properties of negation:

```
Definition not_elim {X : Prop} : not X -> X -> False
:= fun f x => f x.
```

[1] For now you have to believe us that *False* is not provable. A proof of this claim is quite involved. One first has to show that reduction is terminating, confluent, and type preserving.

```
Definition not_dn {X : Prop} : X -> not (not X)
:= fun x f => f x.
Definition mt {X Y : Prop} : (X -> Y) -> not Y -> not X
:= fun f g x => g (f x).
```

To check the proofs, Coq must make use of the definition of the variable *not*. This happens by reducing the specified propositions with delta and beta as justified by the typing rule Red (see § 2.12). Here is a proof that makes use of the previously established proofs *mt* and *not_dn*.

```
Definition not_tn {X : Prop} : not (not (not X)) -> not X
:= mt not_dn.
```

This proof relies on syntactic sugar and type inference. The full proof looks as follows:

```
Print not_tn.
not_tn = fun X : Prop ⇒ @mt X (not (not X)) (@not_dn X)
```

We use the notations $\bot$ and $\neg s$ for *False* and *not s*. Here are the proof rules that correspond to the proofs *False_elim*, *not_elim*, *not_dn*, *not_tn*, and *mt*:

$$\frac{\bot}{s} \qquad \frac{\neg s \quad s}{\bot} \qquad \frac{s}{\neg\neg s} \qquad \frac{\neg\neg\neg s}{\neg s} \qquad \frac{s \to t \quad \neg t}{\neg s}$$

The rightmost rule is known as **modus tollens**.

Exercise 4.3.1 (Triple) Find a proof for $\forall X : \mathit{Prop}.\ \neg\neg\neg X \to \neg X$ that doesn't use defined proofs.

Exercise 4.3.2 Find a proof for $\forall X : \mathit{Prop}.\ \neg X \to (\neg X \to X) \to \bot$.

Exercise 4.3.3 (Circuit) Find a proof for $\forall X : \mathit{Prop}.\ (X \to \neg X) \to (\neg X \to X) \to \bot$.

Exercise 4.3.4 Prove that the proposition $\neg\neg\bot$ is not provable.

Exercise 4.3.5 Determine normal forms for the following propositions.

a) $\forall X : \mathit{Prop}.\ \neg X \to X \to \bot$

b) $\forall X : \mathit{Prop}.\ X \to \neg\neg X$

c) $\forall X\, Y : \mathit{Prop}.\ (X \to Y) \to \neg Y \to \neg X$

4.4 Proof Diagrams and Proof Scripts

Finding proof terms can be difficult, and proof terms for interesting propositions may get large. One can ease the construction of proof terms by drawing so-called **proof diagrams**. A proof diagram keeps track of the assumptions we have made (i.e., local variables and their types) and the propositions we still have to prove. Here is an example of a complete proof diagram for the proposition $\forall X : Prop.\ \neg\neg\neg X \to \neg X$.

$$\cfrac{\cfrac{f \qquad \cfrac{\cfrac{gx}{\bot}\,4}{\neg\neg X}\,3 \quad g : \neg X}{\bot}\,2}{\forall X : Prop.\ \neg\neg\neg X \to \neg X}\,1 \quad X : Prop,\ \ f : \neg\neg\neg X,\ \ x : X$$

Proof diagrams are drawn bottom up. One starts with the initial claim. Then one either *introduces assumptions* or *applies a proof term* obtained with the assumptions. Each step is identified by a horizontal line. In the above diagram, steps 1 and 3 are introduction steps, and steps 2 and 4 are application steps. The assumptions introduced appear to the right of the line, and the proof terms applied appear above the line to the left. The proof term described by a proof diagram is obtained by translating the introduction steps into λ-abstractions and the application steps into applications. The above diagram describes the proof term

```
fun (X : Prop) (f : not (not (not X))) (x : X) => f (fun g : not X => g x)
```

Coq comes with a scripting language that makes it possible to construct proof terms interactively. We can see the commands of the scripting language as commands that construct proof diagrams. Here is a proof script that constructs the proof diagram and the proof term shown above.

```
Definition triple {X : Prop} : not (not (not X)) -> not X.
intros X f x. apply f. intros g. apply (g x). Defined.
```

You can step through the script command by command. At each step you see the available assumptions and the proposition to prove. One contructs the proof script interactively by entering command after command. Coq's user interface makes it possible to undo and redo commands. This way you can walk through the proof diagram. Check this out with the above example. Once the proof term is constructed, you can look at it with the print command.

```
Print triple.
triple = fun (X : Prop) (f : not (not (not X))) (x : X) ⇒ f (fun g : not X ⇒ g x)
```

The commands for proof scripts are known as **tactics**. So far we have seen two tactics, *intros* and *apply*.

With Coq, complex proofs are usually obtained with proof scripts, and the proof scripts are constructed in interaction with Coq.

Here is a proof diagram for the proposition $\forall X : Prop.\ \neg\neg\neg X \to \neg X$ that uses the proofs *mt* and *not_dn* defined in §4.3. It consists of one introduction and two application steps.

$$\dfrac{\dfrac{mt \qquad \dfrac{not\_dn}{X \to \neg\neg X}}{\neg\neg\neg X \to \neg X}}{\forall X : Prop.\ \neg\neg\neg X \to \neg X}\ X : Prop$$

Note that the diagram uses *mt* and *not_dn* with implicit arguments. The translation of the diagram into a proof script looks as follows:

```
Definition triple' {X : Prop} : not (not (not X)) -> not X.
intros X. apply mt. apply not_dn. Defined.

Print triple'.
triple' = fun X : Prop ⇒ mt not_dn
```

Exercise 4.4.1 Consider the propositions of Exercise 4.2.1. Draw proof diagrams and write proof scripts. Run all proof scripts with Coq. Understand the correspondence between proof diagrams, proof scripts, and proof terms.

4.5 Let and Assert

We will now discuss different proofs of the proposition of Exercise 4.3.3:

```
Definition Circuit : Prop := forall X : Prop,  (X -> not X) -> (not X -> X) -> False.
```

Here is a proof diagram for *Circuit*.

$$\dfrac{\dfrac{f \qquad \dfrac{g \qquad \dfrac{\dfrac{fxx}{\bot}}{\neg X}\ x:X}{X} \qquad \dfrac{g \qquad \dfrac{\dfrac{fxx}{\bot}}{\neg X}\ x:X}{X}}{\bot}}{\forall X : Prop.\ (X \to \neg X) \to (\neg X \to X) \to \bot}\ X : Prop,\ f : X \to \neg X,\ g : \neg X \to X$$

Note that the diagram contains the subproof for X twice. The diagram corresponds to the following proof script:

```
Definition circuit_straight : Circuit.
intros X f g.  apply f.
apply g. intros x. apply (f x x).
apply g. intros x. apply (f x x).
Defined.
```

The proof term constructed looks as follows:

```
Print circuit_straight.
fun (X : Prop) (f : X → not x) (g : not X → X) ⇒ f (g (fun x : X ⇒ f x x)) (g (fun x : X ⇒ f x x))
```

When we write the proof term, the duplication of the subproof for X can be avoided with the let notation:

```
Definition circuit_let : Circuit
:= fun X f g => let x := g (fun x' => f x' x') in f x x.
```

The **let notation** is an abbreviation for a beta redex:

$$let\ x := s\ in\ t \quad \rightsquigarrow \quad (fun\ x \Rightarrow t)\ s$$

The **assert tactic** gives us the equivalent of the let notation for proof scripts:

```
Definition circuit_assert : Circuit.
intros X f g.
assert (x := g (fun x' => f x' x')).
apply (f x x).
Defined.
```

It is also possible to construct the proof term for the local variable with a script.

```
Definition circuit_assert' : Circuit.
intros X f g.
assert (x : X). apply g. intros x'. apply (f x' x').
apply (f x x).
Defined.
```

The proof terms *circuit_let*, *circuit_assert*, and *circuit_assert'* are all identical. Use the print command to convince yourself.

Here is a proof diagram that simulates the let notation with an explicit beta redex.

$$\dfrac{\dfrac{\dfrac{\dfrac{f\,x\,x}{\bot}}{X \to \bot}\ x:X \qquad \dfrac{g \qquad \dfrac{\dfrac{f\,x'\,x'}{\bot}}{\neg X}\ x':X}{X}}{\bot}}{\forall X : Prop.\ (X \to \neg X) \to (\neg X \to X) \to \bot}\ X : Prop,\ \ f : X \to \neg X,\ \ g : \neg X \to X$$

There is another elegant proof of *Circuit* that uses an auxiliary proof R.

```
Definition R {X Y : Prop} : (X -> X -> Y) -> X -> Y
:= fun f x => f x x.
Definition circuit_R : Circuit
:= fun X f g => R f (g (R f)).
```

4.6 Conjunction

We will define the **conjunction** $s \wedge t$ of two propositions s and t as a proposition such that the following proof rules are sound:

$$\frac{s \qquad t}{s \wedge t} \qquad\qquad \frac{s \wedge t \qquad s \to t \to u}{u}$$

The left rule is called an **introduction rule** since it establishes a conjunction as provable. It says that $s \wedge t$ is provable if s and t are provable. The right rule is called an **elimination rule** since it makes use of a proven conjunction. It says that given a provable conjunction $s \wedge t$, we can prove a proposition u by proving the implication $s \to t \to u$. This means that we can prove u under the assumption that both s and t are provable.

We define conjunction as a function that given two propositions X and Y yields the elimination rule for $X \wedge Y$:

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

Given this definition, we prove the soundness of the proof rules for conjunction.

```
Definition conj {X Y : Prop} : X -> Y -> and X Y
:= fun x y _ f => f x y.
```

```
Definition and_elim {X Y Z: Prop} : and X Y -> (X -> Y -> Z) -> Z
:= fun f => f Z.
```

Next we prove the soundness of two specialized elimination rules:

$$\frac{s \wedge t}{s} \qquad\qquad \frac{s \wedge t}{t}$$

```
Definition proj1 {X Y : Prop} : and X Y -> X
:= fun f => f X (fun x _ => x).
Definition proj2 {X Y : Prop} : and X Y -> Y
:= fun f => f Y (fun _ y => y).
```

We can think of the proof of a conjunction $s \wedge t$ as a pair (u, v) consisting of a proof u of s and a proof v of t. Under this view, the introduction rule constructs such a pair, and the elimination rules make the components of such a

pair available. In fact, the proofs *conj*, *proj1*, and *proj2* are the operations *cons*, *fst*, and *snd* for polymorphic pairs (see §2.11).

We prove that conjunction is a commutative operation.

```
Definition and_com {X Y : Prop} : and X Y -> and Y X.
intros X Y p. apply p. intros x y. apply (conj y x). Defined.
```

Step carefully through the proof script to understand how it works. The corresponding proof diagram looks as follows.

$$\dfrac{\dfrac{p\ \_ \qquad \dfrac{\dfrac{conj\ y\ x}{and\ Y\ X}}{X \to Y \to and\ Y\ X}\ \ x:X,\ y:Y}{and\ Y\ X}}{\forall X\,Y : Prop.\ and\ X\ Y \to and\ Y\ X}\ \ X\ Y : Prop,\ p : and\ X\ Y$$

The diagram reveals the interesting fact that the apply tactic puts in underlines automatically. Without this feature we would have to write *apply*$(p\ \_)$ in the proof script. To see the full truth, look at the proof term constructed by the script:

```
Print and_com.
```
and_com = fun (X Y : Prop) (p : and X Y) ⇒ p (and Y X) (fun (x : X) (y : Y) ⇒ conj y x)

Next we prove that conjunction is associative.

```
Definition and_asso {X Y Z: Prop} : and X (and Y Z) -> and (and X Y) Z.
intros X Y Z p.  apply p. intros x q. apply q. intros y z.
apply (conj (conj x y) z). Defined.
```

Constructing the proof script in interaction with Coq is straightforward. Constructing the proof term by hand is tedious since one has to supply many type arguments. Print the proof term to see this.

To save parentheses, we follow the common convention that $\wedge$ takes its arguments before $\to$. Thus $s \wedge t \to u \wedge v$ is to be read as $(s \wedge t) \to (u \wedge v)$.

Exercise 4.6.1 Prove the following propositions.

a) $\forall X : Prop.\ X \to X \wedge X$

b) $\forall X\,Y\,Z : Prop.\ (X \wedge Y) \wedge Z \to X \wedge (Y \wedge Z)$

4.7 Equivalence

We define **equivalence** as follows:

```
Definition iff (X Y : Prop) := and (X -> Y) (Y -> X).
```

We write $s \leftrightarrow t$ for a proposition *iff* s t. Note that an equivalence $s \leftrightarrow t$ is provable if and only if the implications $s \to t$ and $t \to s$ are provable. Thus, if $s \leftrightarrow t$ is provable, s is provable whenever t is provable, and t is provable whenever s is provable. We say that two propositions are **provably equivalent** if the equivalence $s \leftrightarrow t$ is provable.

We adopt the convention that $\leftrightarrow$ takes its arguments after $\to$. Thus we read $s \to t \to u \leftrightarrow t \to s \to u$ as $(s \to t \to u) \leftrightarrow (t \to s \to u)$.

It's time to use Coq' notation command. The commands

```
Notation "~ s" := (not s) : type_scope.
Notation "s /\ t" := (and s t) : type_scope.
Notation "s <-> t"  := (iff s t) : type_scope.
```

establish convenient notations for *not*, *and*, and *iff* that corresponds to the mathematical notations we are already using. Here are two proofs involving equivalences. Step through them to see the fine points. Note the use of proof K defined in §4.2.1.

```
Definition and_idempotence {X : Prop} : X /\ X <-> X.
intros X. apply conj.
intros p. apply p. apply (K x).
intros x. apply (conj x x).
Defined.
```

```
Definition iff_circuit {X : Prop} : ~(X <-> ~X).
intros X e. apply e. intros f g.
assert (x : X). apply g. intros x. apply (f x x).
apply (f x x).
Defined.
```

Exercise 4.7.1 Prove the following propositions.

a) $\forall X : Prop.\ \bot \to X \leftrightarrow \top$

b) $\forall X : Prop.\ \top \to X \leftrightarrow X$

c) $\forall X : Prop.\ X \to \top \leftrightarrow \top$

d) $\forall X : Prop.\ (X \leftrightarrow \top) \leftrightarrow X$

e) $\forall X : Prop.\ (X \leftrightarrow \bot) \leftrightarrow \neg X$

f) $\forall XY : Prop.\ X \to \neg Y \leftrightarrow Y \to \neg X$

g) $\forall X : Prop.\ X \wedge \bot \leftrightarrow \bot$

h) $\forall X : Prop.\ X \wedge \top \leftrightarrow X$

4.8 Theorem, Lemma, and Qed

Coq offers the commands *Theorem* and *Qed* as alternatives to the commands *Definition* and *Defined*. Here is an example.

```
Theorem example_theo (X Y Z : Prop) :
X -> Y -> Z <-> X /\ Y -> Z.
intros X Y Z.  apply conj.
intros f g.  apply g. apply f.
intros f x y. apply (f (conj x y)).
Qed.
```

If you use *Theorem* rather than *Definition* you must give the proof with a script. This is the only difference between *Definition* and *Theorem*. There is also a difference between *Defined* and *Qed*. While *Defined* introduces a *transparent name* that is subject to delta reduction, *Qed* introduces an *opaque name* that cannot be delta reduced.

```
Eval cbv in example_theo.
```

example_theo

With the print command you can display the proof term *example_theo* is equated to. Coq offers the keywords *Lemma* and *Corollary* as synonyms for the keyword *Theorem*. From now on we will state and prove propositions using the keywords *Theorem*, *Lemma*, *Corollary*, and *Qed*.

4.9 Disjunction

We will define the **disjunction** $s \vee t$ of two propositions s and t as a proposition such that the following proof rules are sound:

$$\frac{s}{s \vee t} \qquad \frac{t}{s \vee t} \qquad \frac{s \vee t \qquad s \to u \qquad t \to u}{u}$$

This time there are two introduction rules saying that we can prove a disjunction $s \vee t$ if we can prove s or t. The third rule is an elimination rule that provides for case analysis. It says that given a proof of a disjunction $s \vee t$, we can prove a proposition u by proving the implications $s \to u$ and $t \to u$. We define disjunction as follows:

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

This definition represents a disjunction $X \vee Y$ as the elimination rule for $X \vee Y$. The soundness proofs for the proof rules are straightforward.

```
Lemma or_introl {X Y : Prop} : X -> or X Y.
intros X Y x Z f g. apply (f x). Qed.

Lemma or_intror {X Y : Prop} : Y -> or X Y.
intros X Y y Z f g. apply (g y). Qed.

Lemma or_elim {X Y Z : Prop} : or X Y -> (X -> Z) -> (Y -> Z) -> Z.
intros X Y Z p. apply (p Z). Qed.
```

We can think of the proof of a disjunction $s \vee t$ as a pair (i, u) where either $i = 1$ and u is a proof of s or $i = 2$ and u is a proof of t. We speak of a *tagged proof* and call i the *tag* of the proof. Under this view, the introduction rules construct tagged proofs, and the elimination rule decomposes tagged proofs.

We adopt the notational convention that the operator symbol $\vee$ takes its arguments before $\to$ and after $\wedge$. Thus $s \vee t \wedge u \to s$ is read as $(s \vee (t \wedge u)) \to s$. We also define a notation with analogous properties in Coq.

```
Notation "s \/ t" := (or s t) : type_scope.
```

Here is a proof diagram verifying the commutativity of disjunction:

$$\dfrac{\dfrac{p\ \_ \qquad \dfrac{or\_intror}{X \to Y \vee X} \qquad \dfrac{or\_introl}{Y \to Y \vee X}}{Y \vee X}}{\forall X\, Y : Prop.\ X \vee Y \to Y \vee X} \quad X\ Y : Prop,\ \ p : X \vee Y$$

The corresponding proof script and proof term look as follows:

```
Lemma or_com {X Y : Prop} : X \/ Y -> Y \/ X.
intros X Y p. apply p. apply or_intror. apply or_introl. Qed.

Print or_com.
fun (X Y : Prop) (p : X \/ Y) ⇒ p (Y \/ X) or_intror or_introl
```

Here is a proof of the De Morgan law for disjunction:

```
Lemma or_DeMorgan {X Y : Prop} : ~(X \/ Y) <-> ~X /\ ~Y.
intros X Y. apply conj.
intros f. apply conj.
intros x. apply f. apply (or_introl x).
intros y. apply f. apply (or_intror y).
intros p q. apply p. apply q. Qed.
```

Exercise 4.9.1 Prove the following propositions.

a) $\forall X : Prop.\ X \vee \bot \leftrightarrow X$

b) $\forall X : Prop.\ X \vee \top \leftrightarrow \top$

c) $\forall X : Prop.\ X \vee X \leftrightarrow X$

d) $\forall X\, Y : \mathit{Prop}.\ X \vee Y \to Y \vee X$

e) $\forall X\, Y\, Z : \mathit{Prop}.\ X \vee (Y \vee Z) \to (X \vee Y) \vee Z$

f) $\forall X\, Y\, Z : \mathit{Prop}.\ X \wedge (Y \vee Z) \leftrightarrow X \wedge Y \vee X \wedge Z$

g) $\forall X\, Y\, Z : \mathit{Prop}.\ X \vee Y \wedge Z \leftrightarrow (X \vee Y) \wedge (X \vee Z)$

h) $\forall X\, Y : \mathit{Prop}.\ \neg X \vee \neg Y \to \neg(X \wedge Y)$

4.10 Canonicity of Proof Rules

We have defined the logical operations conjunction and disjunction such that they validate certain proof rules. It turns out that the proof rules determine conjunction and disjunction up to equivalence. This means that two disjunctions that both validate the characteristic rules cannot be distinguished with respect to provability. We establish the canonicity of the proof rules with the following section.

```
Section Canonicity_Disjunction.
Variable or' : Prop -> Prop -> Prop.
Variable  or'_introl : forall X Y : Prop,  X -> or' X Y.
Variable or'_intror : forall X Y : Prop,  Y -> or' X Y.
Variable or'_elim : forall X Y Z : Prop, or' X Y -> (X -> Z) -> (Y -> Z) -> Z.
Lemma canonicity_disjunction (X Y : Prop) : X \/ Y <-> or' X Y.
intros X Y. apply conj.
intros H. apply (or'_elim _ _ _ H). apply or_introl. apply or_intror.
intros H. apply H. apply or'_introl. apply or'_intror. Qed.
End Canonicity_Disjunction.
```

Lemma *canonicity_disjunction* states that every disjunction operation *or'* that validates the characteristic proof rules for disjunction is equivalent to the disjunction operator we have defined. Use the print command to see the externalized form of the lemma.

Exercise 4.10.1 Prove that the characteristic proof rules for conjunction determine conjunction up to equivalence.

5 Excluded Middle and Basic Laws

We study the classical assumption of excluded middle and list the basic laws for the propositional operations implication, negation, conjunction, disjunction, and equivalence.

5.1 Excluded Middle

Here are four prominent propositions.

- **Law of Excluded Middle** $\forall X : Prop.\ X \vee \neg X$
- **Double Negation Law** $\forall X : Prop.\ \neg\neg X \to X$
- **Contraposition Law** $\forall X\, Y : Prop.\ (\neg Y \to \neg X) \to X \to Y$
- **Peirce's Law** $\forall X\, Y : Prop.\ ((X \to Y) \to X) \to X$

None of these propositions is provable. Neither are the negations of these propositions provable. However, any two of these propositions are provably equivalent.

The equivalence proofs for the laws are interesting and provide us with excellent exercises. The proofs are more difficult than what we have seen so far. If you can synthesize the proof terms without using pencil and paper, you are a wizard. The rest of us either draws a proof diagram on paper or constructs a proof script in interaction with Coq. The latter is mor fun since Coq will do the bookkeeping and ensure the correctness of the proofs obtained. Below are some of the proofs. To understand the proof scripts, you can either step through them with Coq or draw the corresponding proof diagrams by hand. You may also ask Coq to print the proof terms synthesized.

```
Definition XM : Prop := forall X : Prop, X \/ ~X.
Definition DN : Prop := forall X : Prop, ~~X -> X.
Definition CP : Prop := forall X Y : Prop, (~Y -> ~X) -> X -> Y.
Definition Peirce : Prop := forall X Y : Prop, ((X -> Y) -> X) -> X.

Lemma DN_XM : DN -> XM.
intros dn X. apply dn.
intros f. apply f. apply or_intror.
intros x. apply f. apply (or_introl x). Qed.
```

```
Lemma XM_DN : XM -> DN.
intros xm X f. apply (xm X).
apply I.
intros g. apply (f g X). Qed.

Lemma DN_Peirce : DN -> Peirce.
intros dn X Y f. apply dn.  intros g. apply g.
apply f. intros x. apply (g x Y). Qed.

Lemma Peirce_DN : Peirce -> DN.
intros p X. apply (p _ False).
intros f g. apply g. intros x.  apply f.apply (K x). Qed.
```

Note that the command *apply*(*p _ False*) in the proof of *Peirce_DN* contains a proof term with an unspecified argument. Coq derives the proposition $\neg\neg X \to X$ for this argument.

In mathematics, one assumes that every well-formed statement is either true or false. One also assumes that the law of excluded middle holds and can be used in proofs. We can make this assumption in Coq by declaring a parameter that serves as a proof of the law of excluded middle.

One has to be very careful if one assumes provability of propositions. For instance, if we assume that the law of excluded middle and also the negation of Peirce's law are provable, our logic collapses since we can now prove all propositions (by *not_elim* since excluded middle and Peirce are provably equivalent). We call a sequence of parameter declarations **consistent** if falsity remains unprovable. One can show that declaring excluded middle as provable is consistent.

If we want to prove $\bot$, a case analysis as provided by excluded middle can be made without using excluded middle. Here is the proof providing for this case analysis.

```
Lemma xm_false (X : Prop) : (X -> False) -> (~X -> False) -> False.
intros X f g. apply (g f).  Qed.
```

An interesting use of *xm_false* will appear in §7.2.

Exercise 5.1.1 Prove $DN \leftrightarrow CP$.

Exercise 5.1.2 Prove $XM \leftrightarrow \forall X\, Y : Prop.\ (X \to Y) \to \neg X \vee Y$.

Exercise 5.1.3 Prove $XM \leftrightarrow \forall X\, Y : Prop.\ \neg(\neg X \wedge \neg Y) \to X \vee Y$.

Exercise 5.1.4 Prove $XM \leftrightarrow \forall X\, Y : Prop.\ X \vee Y \to X \vee \neg X \wedge Y$.

Exercise 5.1.5 Prove $XM \leftrightarrow \forall X\, Y : Prop.\ (X \to Y) \to (\neg X \to Y) \to Y$.

Exercise 5.1.6 (Proof by Contradiction) Proof by contradiction is the following proof principle: To prove s, assume $\neg s$ and derive a contradiction. Under a contradiction we understand the situation that there is a proposition t such that both t and $\neg t$ can be proven. We can formulate the proof-by-contradiction principle with the following proposition:

$$\mathit{Contra} := \forall X\, Y : \mathit{Prop}.\ (\neg X \to Y) \to (\neg X \to \neg Y) \to X$$

Prove that *DN* and *Contra* are equivalent.

Exercise 5.1.7 Prove $\mathit{XM} \to \forall X\, Y : \mathit{Prop}.\ \neg(X \to Y) \leftrightarrow X \wedge \neg Y$.

Exercise 5.1.8 Prove the following proposition:
$\mathit{DN} \to \forall X : \mathit{Type}\ \forall p : X \to \mathit{Prop}.\ \neg\neg(\forall x : X.\ px) \to \forall x : X. \neg\neg px$.

5.2 Basic Laws

In this section we state provable propositions that formulate important properties of the logical operations we have introduced so far. We refer to the propositions stated in this section as *laws*.

We distinguish between **intuitionistic laws** and **classical laws**. While intuitionistic laws can be shown without using excluded middle, the proofs of classical laws may use excluded middle. Every intuitionistic law is also a classical law but not vice versa. Non-intuitionistic laws (i.e., laws that require excluded middle) are highlighted.

The most basic laws are shown in Figure 5.1. They are all direct consequences of the definitions of the logical operations. For $\top$, $\bot$, $\wedge$, and $\vee$ we state laws that formulate the characteristic proof rules. The propositions $\top$ and $\bot$ have only one characteristic rule each, which for $\top$ is an introduction rule and for $\bot$ is an elimination rule.

For readability the laws are stated without the universal quantifications for the propositional variables x, y, z. The quantifiers for the variables will be added automatically if the laws are proven in a section like this:

```
Section Laws.
  Variable xm : XM.
  Variable x y z : Prop.
  prove laws here
End Laws.
```

This will also add the assumption *XM* to laws whose proof uses the variable *xm*.

Figures 5.2 and 5.3 show laws for implications. The laws in Figure 5.3 are non-intuitionistic. For each of them, the direction from left to right requires

$$
\begin{aligned}
x \to y &\leftrightarrow \forall \_ : x.\ y \\
\top &\leftrightarrow \forall x : \mathit{Prop}.\ x \to x \\
\bot &\leftrightarrow \forall x : \mathit{Prop}.\ x \\
x \wedge y &\leftrightarrow \forall z : \mathit{Prop}.\ (x \to y \to z) \to z \\
x \vee y &\leftrightarrow \forall z : \mathit{Prop}.\ (x \to z) \to (y \to z) \to z \\
\neg x &\leftrightarrow x \to \bot \\
(x \leftrightarrow y) &\leftrightarrow (x \to y) \wedge (y \to x)
\end{aligned}
$$

$$
\begin{aligned}
&\top \\
&\bot \to x \\
&x \to y \to x \wedge y \\
&x \wedge y \to (x \to y \to z) \to z \\
&x \to x \vee y \\
&y \to x \vee y \\
&x \vee y \to (x \to z) \to (y \to z) \to z
\end{aligned}
$$

Figure 5.1: Definitional laws

excluded middle while the other direction is intuitionistic. Figure 5.4 shows laws for equivalence.

Figure 5.5 shows laws for the Boolean connectives that are known from Boolean algebra. Three of the laws require excluded middle.

Exercise 5.2.1 Try to prove all laws stated. None of the proofs is really difficult.

$$
\begin{array}{rcl}
\bot \to x & \leftrightarrow & \top \\
\top \to x & \leftrightarrow & x \\
x \to \bot & \leftrightarrow & \neg x \\
x \to \top & \leftrightarrow & \top \\
x \to x & \leftrightarrow & \top \\
x \to x \to y & \leftrightarrow & x \to y \\
x \to y \to z & \leftrightarrow & y \to x \to z \\
x \to \neg y & \leftrightarrow & y \to \neg x \\
x \wedge y \to z & \leftrightarrow & y \to x \to z \\
x \wedge (x \to y) & \leftrightarrow & x \wedge y \\
x \vee y \to z & \leftrightarrow & (x \to z) \wedge (y \to z) \\
x \to (y \wedge z) & \leftrightarrow & (x \to y) \wedge (x \to z) \\
x \to y & \leftrightarrow & (x \leftrightarrow x \wedge y) \\
x \to y & \leftrightarrow & (y \leftrightarrow x \vee y)
\end{array}
$$

Figure 5.2: Laws for implication

$$
\begin{array}{rcl}
x \to y & \leftrightarrow & \neg x \vee y \\
\neg(x \to y) & \leftrightarrow & x \wedge \neg y \\
\neg y \to \neg x & \leftrightarrow & x \to y \\
(x \to y) \to y & \leftrightarrow & x \vee y \\
x \wedge y \to z & \leftrightarrow & x \to \neg y \vee z \\
x \wedge \neg y \to z & \leftrightarrow & x \to y \vee z
\end{array}
$$

Figure 5.3: Laws for implication that require *XM* (from left to right)

$x \leftrightarrow x$	Refexivity
$(x \leftrightarrow y) \leftrightarrow (y \leftrightarrow x)$	Symmetry
$(x \leftrightarrow y) \to (y \leftrightarrow z) \to (x \leftrightarrow z)$	Transitivity
$(x \leftrightarrow \bot) \leftrightarrow \neg x$	
$(x \leftrightarrow \top) \leftrightarrow x$	
$(x \leftrightarrow \neg x) \leftrightarrow \bot$	
$(x \leftrightarrow y) \leftrightarrow (x \wedge y) \vee (\neg x \wedge \neg y)$	
$(x \leftrightarrow y) \leftrightarrow (\neg x \vee y) \wedge (x \vee \neg y)$	
$(\neg x \leftrightarrow \neg y) \leftrightarrow (x \leftrightarrow y)$	
$\neg(x \leftrightarrow y) \leftrightarrow (\neg x \leftrightarrow y)$	
$(\neg x \leftrightarrow y) \leftrightarrow (x \leftrightarrow \neg y)$	

Figure 5.4: Laws for equivalence (highlighted laws require *XM*)

$\neg\bot$	$\leftrightarrow$	$\top$	
$\neg\top$	$\leftrightarrow$	$\bot$	
$\top \wedge x$	$\leftrightarrow$	x	Identity
$\bot \vee x$	$\leftrightarrow$	x	
$\bot \wedge x$	$\leftrightarrow$	$\bot$	Dominance
$\top \vee x$	$\leftrightarrow$	$\top$	
$x \wedge y$	$\leftrightarrow$	$y \wedge x$	Commutativity
$x \vee y$	$\leftrightarrow$	$y \vee x$	
$x \wedge (y \wedge z)$	$\leftrightarrow$	$(x \wedge y) \wedge z$	Associativity
$x \vee (y \vee z)$	$\leftrightarrow$	$(x \vee y) \vee z$	
$x \wedge x$	$\leftrightarrow$	x	Idempotence
$x \vee x$	$\leftrightarrow$	x	
$x \wedge (y \vee z)$	$\leftrightarrow$	$(x \wedge y) \vee (x \wedge z)$	Distributivity
$x \vee (y \wedge z)$	$\leftrightarrow$	$(x \vee y) \wedge (x \vee z)$	
$x \wedge (x \vee y)$	$\leftrightarrow$	x	Absorption
$x \vee (x \wedge y)$	$\leftrightarrow$	x	
$x \wedge \neg x$	$\leftrightarrow$	$\bot$	Complement
$x \vee \neg x$	$\leftrightarrow$	$\top$	
$\neg(x \wedge y)$	$\leftrightarrow$	$\neg x \vee \neg y$	De Morgan
$\neg(x \vee y)$	$\leftrightarrow$	$\neg x \wedge \neg y$	
$\neg\neg x$	$\leftrightarrow$	x	Double negation
$\neg\neg\neg x$	$\leftrightarrow$	$\neg x$	

Figure 5.5: Boolean laws (highlighted laws require *XM*)

5 Excluded Middle and Basic Laws

6 Existential Quantification

In this chapter we study existential quantification and list the most important quantifier laws.

6.1 Functional Representation of Quantification

Given a list of definitions in Coq, can we rewrite this list such that universal quantification is used exactly once? Yes, we can. We simply start the list with the definition of a function *all* and then replace every universal quantification $\forall x : s.t$ with the term $all\,(\lambda x : s.t)$. Note that the use of the lambda abstraction preserves the bindings that come with the local variable x.

How do we define *all*? Let's first think about the type of *all*. Its clear that *all* returns a proposition and takes a function of type $X \to Prop$ as argument, where X is the type we quantify over. This gives us the type

$$\forall X : Type.\ (X \to Prop) \to Prop$$

for all. Provided we declare the argument X as implicit (see § 2.11), the term $all\,(\lambda x : s.t)$ obtained form a quantification $\forall x : s.t$ will in fact be well-typed. The definition of *all* is now straightforward:

```
Definition all {X : Type} (p : X -> Prop) : Prop := forall x : X, p x.
```

We can now write $all\,(\lambda x : s.\,t)$ in place of $\forall x : s.\,t$ since the former reduces with delta and beta to the latter. Let's consider this reduction in detail:

$all\,(\lambda x : s.\,t)$	the term we start with
$= @all\ \_\ (\lambda x : s.\,t)$	making the implicit argument explicit
$= @all\ s\ (\lambda x : s.\,t)$	replacing underline with an inferred type
$= (\lambda X p.\forall x : X.\,p x)\ s\ (\lambda x : s.\,t)$	delta reduction of $@all$
$= (\lambda p.\forall x : s.\,p x)\ (\lambda x : s.\,t)$	beta reduction
$= \forall x : s.\,(\lambda x : s.\,t)\,x$	beta reduction
$= \forall x : s.\,t$	beta reduction

For concrete examples you can do the reduction with Coq.

```
Eval cbv in (all (fun X : Prop, X -> X)).
```
forall x : Prop, X → X

Exercise 6.1.1 Proof the following lemmas.

```
Lemma inst X (p : X -> Prop) (x : X) : all p -> p x.
Lemma gen X (p : X -> Prop) (q : Prop) : (forall x, q -> p x) -> q -> all p.
```

6.2 Existential Quantification

Given the natural deduction rules for existential quantification

$$\frac{v : s \qquad t_v^x}{\exists x : s.\, t} \qquad\qquad \frac{\exists x : s.\, t \qquad \forall x : s.\, t \to u}{u}$$

and the functional representation of quantification, the definition of existential quantification is a matter of routine:

```
Definition ex {X : Type} (p : X -> Prop) : Prop :=
forall Z : Prop, (forall x : X, p x -> Z) -> Z.
```

The definition of the deduction rules for existential quantification is now straightforward.

```
Lemma ex_intro {X : Type} {p : X -> Prop} (x : X) :
p x -> ex p.
```

```
intros X p x a Z f. apply (f x a). Qed.
```

```
Lemma ex_elim (X : Type) (p : X -> Prop) (Z : Prop) :
ex p -> (forall x : X, p x -> Z) -> Z.
```

```
intros X p Z f.  apply f. Qed.
```

We define convenient notations for existential quantifications.

```
Notation "'exists' x : s , t" := (ex (fun x : s => t)) : type_scope.
Notation "'exists' x , t" := (ex (fun x => t)) : type_scope.
```

Existential quantification generalizes conjunction. This can be seen from the natural deduction rules. We formulate this observation with a lemma.

```
Lemma ex_and (X Y : Prop) : X /\ Y <-> exists x : X, Y.
intros X Y. apply conj. apply I. apply I. Qed.
```

Note that the proof is so straightforward since the terms $X \wedge Y$ and $\exists x : X.Y$ reduce to the same normal form.

Exchange Law

The order of existential quantifications does not matter.

```
Lemma ex_exchange (X Y :Type) (p : X -> Y -> Prop) :
(exists x, exists y, p x y) -> exists y, exists x, p x y.

intros X Y p f.
apply f. intros x g. apply g. intros y a.
apply (ex_intro y). apply (ex_intro x). apply a. Qed.
```

Exercise 6.2.1 Prove that natural deduction determines $\exists$ up to equivalence.

```
Section Canonicity_Exists.
Variable ex' : forall X : Type, (X -> Prop) -> Prop.
Variable ex'_intro : forall (X : Type) (p : X -> Prop) (x : X),
  p x -> ex' X p.
Variable ex'_elim : forall (X : Type) (p : X -> Prop) (Z : Prop),
  ex' X p -> (forall x : X, p x -> Z) -> Z.
Lemma canonicity_exists (X : Type) (p : X -> Prop) : ex p <-> ex' X p.
  (* your proof *)
End Canonicity_Exists.
```

Russell's Law

Russell's law has many interesting applications. Russell used it to point out inconsistencies in the theories of Cantor and Frege.

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

intros X p f. apply f. intros x g. apply (iff_circuit (g x)). Qed.
```

Lemma *iff_circuit* was defined in §4.7. From Russell's law we immediately get the following:

1. *Halting problem.* There is no Turing machine that halts on the coding of a Turing machine if and only if this machine does not halt on its own coding. Take for X the set of Turing machines and for pxy the relation "x halts on the coding of y".
2. *Russell's problem.* Given a set X whose elements are sets, there is no $x \in X$ such that $y \in x$ iff $y \notin y$ for all $y \in X$. Take for pxy the relation $y \in x$.
3. *Barber paradox.* Suppose there is a town with just one male barber; and that every man in the town keeps himself clean-shaven: some by shaving themselves, some by attending the barber. It seems reasonable to imagine that the barber obeys the following rule: He shaves all and only those men in town who do not shave themselves. From Russell's law it follows that such a barber cannot exist. Take for X the set of men and for pxy the relation "x shaves y".

Exercise 6.2.2 Prove $\forall X : Type\ \forall p : X \to X \to Prop\ \forall x : X\ \exists y : X.\ \neg(pxy \leftrightarrow \neg pyy)$.

Exercise 6.2.3 Prove the following lemma, which says that a barber who shaves exactly those males who don't shave themseves cannot be male.

```
Lemma Barber (X : Type) (m : X -> Prop) (s : X -> X -> Prop) (b :X) :
(forall x,  m x -> (s b x <-> ~ s x x)) -> ~m b.
```

De Morgan Law for ∃

A negated existential quantification is equivalent to a universal quantification.

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

```
intros X p. apply conj.
intros f x a. apply f. apply (ex_intro x a).
intros f g. apply g. intros x a. apply (f x a). Qed.
```

De Morgan Law for ∀

A negated universal quantification is equivalent to an existential quantification provided we have excluded middle. We need excluded middle since we have to establish an existential quantification. For convenience we use *DN* rather than *XM*.

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

```
intros X p dn. apply conj.
intros f.  apply dn. intros g. apply f. intros x. apply dn.
  intros h. apply g. apply (ex_intro x). apply h.
intros f g. apply f. intros x h. apply (h (g x)). Qed.
```

Note that *DN* is used twice for the direction from left to right. For the other direction *DN* is not needed.

Exercise 6.2.4 Assume a section with $X\,Y : Type$, $p : X \to Prop$, and $r : X \to Y \to Prop$. Prove the following propositions.

a) $(\forall x.\, px) \to \neg\exists x.\, \neg px$

b) $(\forall x\, \exists y.\, \neg rxy) \to \neg\exists x \forall y.\, rxy$

c) $\forall x\, y.\;\; px \vee py \to \exists x.\, px$

d) $XM \to \forall x\, y\, \exists z.\;\; px \vee py \to pz$

Exercise 6.2.5 Prove that the proposition $\exists X : Prop.\; X \wedge \neg X$ is not provable.

Exercise 6.2.6 (Irrational Numbers) We prove that there exist two irrational numbers x and y such that x^y is rational. We know that 2 is rational and that

$\sqrt{2}$ is irrational. Since $\left(\sqrt{2}^{\sqrt{2}}\right)^{\sqrt{2}} = \sqrt{2}^{\sqrt{2}\sqrt{2}} = \sqrt{2}^2 = 2$, we know that $\left(\sqrt{2}^{\sqrt{2}}\right)^{\sqrt{2}}$ is rational. The claim now follows by case analysis: If $\sqrt{2}^{\sqrt{2}}$ is a rational number, then $x = y = \sqrt{2}$, otherwise $x = \sqrt{2}^{\sqrt{2}}$ and $y = \sqrt{2}$. Prove the following Lemma, which captures part of the argument:

```
Lemma irrat (X : Type) (rat : X -> Prop) (exp : X -> X -> X) (s : X) :
XM -> ~rat s -> rat (exp (exp s s) s) ->
exists x, exists y, ~rat x /\ ~rat y /\ rat (exp x y).
```

6.3 Inhabitation

We define inhabitation and voidness of types as follows:

```
Definition inhab (X : Type) : Prop := exists x : X, True.
Definition void (X : Type) : Prop := forall x : X, False.
```

Inhabitation generalizes provability:

```
Lemma inhab_provable (X : Prop) :
inhab X <-> X.

intros X. apply conj.
intros f. apply f. intros x _. apply x.
intros x.  apply (ex_intro x I). Qed.
```

Moreover, *void* generalizes negation:

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

intros X. apply conj.
intros f.  apply f. intros f. apply f. Qed.
```

A type is void if and only if it is not inhabited.

```
Lemma void_not_inhab (X : Type) :
void X <-> ~ inhab X.

intros X.  apply conj.
intros f g. apply g. intros x _. apply (f x).
intros f x. apply f. apply (ex_intro x I). Qed.
```

We need excluded middle to show that a type is inhabited if and only if it is not void. The asymmetry is due to the fact that we have to establish an existential quantification.

```
Lemma equiv_not (X Y : Prop) :
XM -> (X <-> ~Y) -> (Y <-> ~X).

intros X Y xm f.  apply f. intros g h. apply conj.
intros y x. apply (g x y).
intro i. apply (xm Y). intros j.  apply j. intros j. apply (i (h j)). Qed.

Lemma inhab_not_void (X : Type) :
XM -> (inhab X <-> ~ void X).

intros X xm. apply (equiv_not _ _ xm). apply void_not_inhab. Qed.
```

Note that the lemma *equiv_not* is a straightforward consequence of two equivalence laws in Figure 5.4.

Deletion Laws

Here are two quantifier laws that depend on inhabitation.

```
Lemma all_delete (X : Type) (q : Prop) :
inhab X -> ((forall x : X, q) <-> q).

intros X P inh. apply conj.
apply inh. intros x _ f. apply (f x).
intros a _.  apply a. Qed.

Lemma ex_delete (X : Type) (q : Prop) :
inhab X -> ((exists x : X, q) <-> q).

intros X q inh. apply conj.
intros f. apply f. intros _ a. apply a.
apply inh. intros x _ a. apply (ex_intro x a). Qed.
```

Drinker's Paradox

Consider a bar that is inhabited by at least one person. Using excluded middle, we show that we can pick some person in the bar such that everyone in the bar drinks if this person drinks.

```
Lemma drinker (X : Type) (d : X -> Prop):
DN -> inhab X -> exists x, d x -> forall x, d x.

intros X d dn inh. apply inh. intros x _.
apply dn. intros f. apply f.
apply (ex_intro x). intros H y. apply dn. intros g.
apply f. apply (ex_intro y). intros a. apply (g a). Qed.
```

The informal explanation of the drinker's lemma is straightforward. If everyone in the bar drinks, the claim is trivial. If not everyone in the bar drinks, we pick a person x that does not drink. Since x does not drink, the claim is again trivial.

Exercise 6.3.1 Prove the following propositions.

```
Section Quantifier_Laws.
  Variable xm : XM.
  Variable X Y : Type.
  Variable inh : inhab X.
  Variable p p' : X -> Prop.
  Variable r : X -> Y -> Prop.
  Variable q : Prop.
  . . .
  Lemma all_imp : (forall x, q -> p x) <-> q -> forall x, p x.
  apply conj.
  intros f a x. apply (f x a).
  intros f x a. apply (f a).
  Qed.
  Lemma ex_imp : (exists x, q -> p x) <-> q -> exists x, p x.
  apply conj.
  intros H a. apply H. intros x f. apply (ex_intro _ (f a)).
  intros f. apply (xm q).
    intros a. apply (f a). intros x b. apply (ex_intro x). apply (K b).
    intros g. apply inh. intros x _.  apply (ex_intro x). intros a. apply (g a).
  Qed.
  . . .
End Quantifier_Laws.
```

Figure 6.1: Section for quantifier laws

a) $\forall X\, Y : Type.\ \ inhab\ Y \to inhab\,(X \to Y)$.

b) $\forall X\, Y : Type.\ \ inhab\ X \to inhab\,(X \to Y) \to inhab\ Y$.

c) $\forall X : Type.\ \ void\ X \leftrightarrow \forall p : X \to Prop\ \forall x : X.\ p\,x$

d) $\forall X : Type.\ \ inhab\ X \leftrightarrow \forall p : X \to Prop.\ (\forall x : X.\ p\,x) \to \exists x : X.\ p\,x$

6.4 Quantifier Laws

Figure 6.2 shows the basic laws for universal and existential quantification. The quantifier laws are to be proven in a section as shown in Figure 6.1. Note that with mathematical notation one writes a nested quantification such as $\forall x : s.\ \exists y : t.\ u$ without the first dot as $\forall x : s\ \exists y : t.\ u$.

Exercise 6.4.1 Prove all the quantifier laws. Use a section as shown in Figure 6.1. The laws that don't require assumptions are all easy.

$(\exists x{:}X.\ px) \leftrightarrow \forall Z{:}\mathit{Prop}.\ (\forall x{:}X.\ px \to Z) \to Z$			Definition
$\forall x{:}X.\ (\forall x{:}X.\ px) \to px$			Instantiation
$\forall x{:}X.\ px \to \exists x{:}X.\ px$			
$(\forall x{:}X\ \forall y{:}Y.\ rxy) \leftrightarrow \forall y{:}Y\ \forall x{:}X.\ rxy$			Exchange
$(\exists x{:}X\ \exists y{:}Y.\ rxy) \leftrightarrow \exists y{:}Y\ \exists x{:}X.\ rxy$			
$(\forall x{:}X.\ px \wedge p'x) \leftrightarrow (\forall x{:}X.\ px) \wedge \forall x{:}X.\ p'x$			Distributivity
$(\exists x{:}X.\ px \vee p'x) \leftrightarrow (\exists x{:}X.\ px) \vee \exists x{:}X.\ p'x$			
$(\forall x{:}X.\ px \vee q) \leftrightarrow (\forall x{:}X.\ px) \vee q$		X	
$(\exists x{:}X.\ px \wedge q) \leftrightarrow (\exists x{:}X.\ px) \wedge q$			
$(\forall x{:}X.\ q) \leftrightarrow q$		I	Deletion
$(\exists x{:}X.\ q) \leftrightarrow q$		I	
$\neg(\forall x{:}X.\ px) \leftrightarrow \exists x{:}X.\ \neg px$		X	De Morgan
$\neg(\exists x{:}X.\ px) \leftrightarrow \forall x{:}X.\ \neg px$			
$(\forall x{:}X.\ q \to px) \leftrightarrow q \to \forall x{:}X.\ px$			Implication
$(\exists x{:}X.\ q \to px) \leftrightarrow q \to \exists x{:}X.\ px$		XI	
$(\forall x{:}X.\ px \to q) \leftrightarrow (\exists x{:}X.\ px) \to q$			
$(\exists x{:}X.\ px \to q) \leftrightarrow (\forall x{:}X.\ px) \to q$		XI	
$(\exists x{:}X.\ px \to p'x) \leftrightarrow (\forall x{:}X.\ px) \to (\exists x{:}X.\ p'x)$		X	

Highlighted laws require excluded middle (X) and/or inhabitation (I)
as specified on the right

Figure 6.2: Quantifier laws

7 Equality

In this chapter we study equality. We will see several proofs of Cantor's theorem. We discuss propositional and functional extensionality, two basic assumptions commonly made in mathematics. As proofs get more involved, we introduce more features of Coq's tactic language to ease the synthesis of proof terms.

7.1 Definition and Basic Rules

Once more we start from the natural deduction rules.

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

The rules state basic mathematical assumptions: An equation $s = s$ always holds, and given that an equation $s = t$ holds, we can replace t with s. One speaks of replacement of equals with equals. We realize equality as follows.

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

Notation "s = t" := (eq s t) : type_scope.
Notation "s <> t" := (~eq s t) : type_scope.

Lemma eq_intro {X : Type} (x : X) : x=x.
intros X x p. apply I. Qed.

Lemma eq_elim {X : Type} {x y: X} {p : X -> Prop} : x=y -> p y -> p x.
intros X x y p e. apply e. Qed.
```

Equality is reflexive, as stated by *eq_intro*. It is also symmetric and transitive.

```
Lemma eq_sym {X : Type} {x y : X} : x=y -> y=x.
intros X x y f. apply f. apply eq_intro. Qed.

Lemma eq_trans {X : Type} {x z : X} (y : X) : x=y -> y=z -> x=z.
intros X x y z f g. apply f. apply g. Qed.
```

When you step through the proofs, you will see that the application of an equation $s = t$ replaces all occurrences of the term s in the claim with the term t, provided the claim does not have the form $s = t$ (application of g in proof *eq_trans*) or $ut \to us$ (application of e in proof *eq_elim*). Look at the proof terms *eq_sym* and *eq_trans* and identify the predicates the apply tactic has synthesized.

Here are proofs of further key properties of equality.

```
Lemma eq_lc {X Y : Type} {f g : X -> Y} (x : X) : f=g -> f x = g x.
intros X Y x f g e. apply e. apply eq_intro. Qed.

Lemma eq_rc {X Y : Type} {x y : X} (f : X -> Y) : x=y -> f x = f y.
intros X Y x y f e. apply e. apply eq_intro. Qed.

Lemma eq_iff {x y : Prop} : x=y -> (x<->y).
intros x y e. apply e. apply conj. apply I. apply I. Qed.

Lemma eq_not {X : Type} {x y: X} (p : X -> Prop) : p x -> ~p y -> x<>y.
intros X x y p a f e. apply f. apply (eq_sym e). apply a. Qed.
```

The lemmas *eq_lc* and *eq_rc* state that equality is left and right compatible with application. Lemma *eq_iff* states that equality is stronger than equivalence. Lemma *eq_not* is useful for proving disequations (i.e., negated equations). Here is another useful lemma.

```
Lemma True_neq_False : True <> False.
intros H. apply (eq_sym H). apply I. Qed.
```

Exercise 7.1.1 Prove the contrapositions of *eq_lc* and *eq_rc*. That is, prove the following propositions for $(X\,Y : \mathit{Type})$, $(x\,y : X)$, and $(f\,g : X \to Y)$:

a) $fx \neq gx \to f \neq g$

b) $fx \neq fy \to x \neq y$

Exercise 7.1.2 Prove the contraposition of *eq_iff*. That is, prove the proposition $\neg(x \leftrightarrow y) \to x \neq y$ for $x, y : \mathit{Prop}$.

Exercise 7.1.3 Prove *eq_lc* using *eq_rc*.

Exercise 7.1.4 Prove the following variations of *eq_lc* using *eq_lc*.

```
Lemma eq_forall (X Y : Type) (f g : X -> Y) :
f=g -> forall x, f x = g x.

Lemma eq_forall' (X Y : Type) (f g : X -> Y) :
(fun x => f x) = (fun x => g x) -> forall x, f x = g x.
```

Exercise 7.1.5 Prove the following lemma. If you don't succeed, try the proof of *eq_sym*. Explain why the proof *eq_sym* carries over.

```
Lemma eq_hidden_sym X (x y : X) :
(forall p : X -> Prop, p y -> p x) -> forall p : X -> Prop, p x -> p y.
```

Exercise 7.1.6 Below is another characterization of equality. Find a proof.

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

Exercise 7.1.7 Prove that natural deduction determines equality.

```
Section Canonicity_Equality.
Variable eq' : forall (X : Type), X -> X -> Prop.
Variable eq'_intro : forall (X : Type) (x : X), eq' X x x.
Variable eq'_elim : forall (X : Type) (x y : X) (p : X -> Prop), eq' X x y -> p y -> p x.
Lemma canonicity_equality (X : Type) (x y : X) : x=y <-> eq' X x y.
  (* your proof *)
End Canonicity_Equality.
```

7.2 Cantor's Theorem

Cantor's theorem says that the power set of a set is always larger than the set. More precisely, Cantor's theorem states that there is no surjective function from a set to its power set. One consequence of Cantor's theorem is the result that the power set of the natural numbers is not countable. Another consequence is the result that the real numbers are not countable. When Cantor published his results around 1890, they came as a complete surprise. Here is a statement and a proof of Cantor's theorem.

```
Theorem cantor (X : Type) (f : X -> X -> Prop) : exists g, forall x, f x <> g.
intros X f. apply (ex_intro (fun z => ~f z z)). intros x.
apply (mt (eq_lc x)). apply (mt eq_iff). apply iff_circuit. Qed.
```

The power set of X is represented as the function type $X \to \mathit{Prop}$. The proof reduces the claim to *iff_circuit* from §4.7, a proof we used before for Russell's theorem (§6.2). For the existentially quantified g the reduction chooses a function that represents the *diagonal set* $\{\, x \in X \mid x \notin f\,x \,\}$ used in Cantor's proof. The rest of the reduction applies *eq_lc* and *eq_iff*. To account for the negation, *mt* (modus tollens, §4.3) is used. Step through the proof to understand the fine points.

Here is an alternative proof that doesn't use modus tollens.

```
Theorem cantor' (X : Type) (f : X -> X -> Prop) : exists g, forall x, f x <> g.
intros X f. apply (ex_intro (fun z => ~f z z)). intros x H.
apply (@iff_circuit (f x x)). apply eq_iff. apply (eq_lc x H). Qed.
```

Let's now look at an informal proof of Cantor's theorem as it may appear in a mathematical textbook. The proof technique employed is known as *Cantor's diagonal argument*.

Claim. Let X be a set. Then there exists no surjective function $X \to \wp X$.

Proof. Let f be a function $X \to \wp X$. Let $D = \{\, x \in X \mid x \notin f\,x \,\}$. Let $x \in X$. It suffices to show $f\,x \neq D$. Assume $f\,x = D$. Case analysis.

1. $x \in D$. Then $x \notin fx$. Contradiction since $fx = D$.
2. $x \notin D$. Then $x \in fx$. Contradiction since $fx = D$.

The informal proof is different from the formal proofs. It seems that the informal proof uses excluded middle to do the case analysis. This is not the case since the claim is $\bot$ and thus the case analysis is validated by *xm_false* from §5.1. There is also a use of double negation in case (2) that can be avoided easily.

To give a formal proof of Cantor's theorem that is faithful to the informal proof, we use the tactics *pose* and *unfold*. With *pose* we can realize a local definition, and with *unfold* we can delta reduce a given name in the claim. In addition, we use a proof *push* to add a provable premise to the claim. The details will become clear as you step through the proof. Note that *unfold* also beta reduces the claim.

```
Lemma push {X Y : Prop} : X -> (X -> Y) -> Y.
intros X Y x f.  apply (f x). Qed.

Theorem cantor'' (X : Type) (f : X -> X -> Prop) : exists g, forall x, f x <> g.
intros X f. pose (D := fun z => ~f z z). apply (ex_intro D). intros x e.
apply (xm_false (D x)).
intros B. apply (push B). unfold D. apply e. intros C. apply (C B).
intros B. apply (push B). unfold D. apply e. intros C. apply (C B). Qed.
```

A shorter version of the proof appears as *cantor'''* in §7.4.

Exercise 7.2.1 At the proof term level the use of *pose* translates into a let while the use of *unfold* has no effect. The unfold tactic is only needed to enable the apply tactic to do the right thing. For instance, proof *cantor''* uses *unfold* so that *apply* can apply the equation *e*. Use the following example to see what is happening at the proof term level.

```
Lemma pose_unfold_demo (X : Type) (p : X -> Prop) (x : X) :  p x -> exists x : X, p x.
intros X p x H. pose (a := x). apply (ex_intro a). unfold a. apply H. Qed.
```

7.3 Propositional Extensionality

Here are two prominent propositions.

- **Propositional Extensionality (*PE*)** $\quad \forall X\,Y : Prop.\ (X \leftrightarrow Y) \to X = Y$
- **Propositional Case Analysis (*PCA*)** $\quad \forall X : Prop.\ X = \bot \vee X = \top$

In mathematics, one commonly assumes *PCA*, which says that every proposition denotes one of the two truth values. It is known that $XM \to PCA$ is not provable. We will prove the equivalence $PCA \leftrightarrow XM \wedge PE$.

```
Definition PE : Prop := forall X Y : Prop, (X <-> Y) -> X=Y.
Definition PCA : Prop := forall X : Prop, X=False \/ X=True.
```

We first establish two useful consequences of *PE*.

```
Lemma eq_true {X : Prop} : PE -> X -> X=True.
intros X pe x. apply pe.  apply (conj (K I) (K x)). Qed.

Lemma eq_false {X : Prop}: PE -> ~X -> X=False.
intros X pe f. apply pe.  apply (conj f False_elim). Qed.
```

Now we prove that *PCA* follows from *XM* and *PE*.

```
Lemma xm_pe_pca : XM -> PE -> PCA.
intros xm pe X. apply (xm X).
intros H. apply or_intror. apply (eq_true pe H).
intros H. apply or_introl. apply (eq_false pe H). Qed.
```

Before we proceed we define a useful tactic. You may have observed that the sequence

$\cdots$ intros H. apply H. $\cdots$

appears quite often in our proofs. We capture this pattern by defining a tactic:

```
Ltac inap := (intros Inap; apply Inap; clear Inap).
```

The command defines a tactic *inap*, which introduces an assumption *Inap*, applies it, and discards it so that the name *Inap* can be reused.

We now show that *PCA* implies *XM*. The proof uses the tactic *inap* three times.

```
Lemma pca_xm : PCA -> XM.
intros pca X. apply (pca X).
inap. apply or_intror. inap.
inap. apply (or_introl I). Qed.
```

Two tactics t_1 and t_2 can be combined with a semicolon into a tactic $t_1;t_2$. Execution of $t_1;t_2$ first executes t_1 and then executes t_2 on each subgoal generated by t_1. This combination is of interest if t_1 generates more than one subgoal. We can shorten the above proof by using the **sequencing operation** expressed by the semicolon.

```
Lemma pca_xm' : PCA -> XM.
intros pca X. apply (pca X) ; inap.
apply or_intror. inap.
apply (or_introl I). Qed.
```

Step through the script to see the effect.

Next we prove that *PCA* implies *PE*. Using *PCA*, the proof simply enumerates the possible values of the two propositional variables X and Y in *PE*. Once this is done, we are facing four cases each of which is easy to prove.

```
Lemma pca_pe : PCA -> PE.
intros pca X Y.
apply (pca X) ; inap ; apply (pca Y) ; inap ; intros H.
apply eq_intro.
apply (proj2 H I).
apply (proj1 H I).
apply eq_intro. Qed.
```

Step through the script to see what is happening. Replace the semicolons through dots to see what exactly happens in the first case.

Exercise 7.3.1 Prove $PCA \leftrightarrow \forall p : Prop \to Prop\ \forall X : Prop.\ \ p\bot \to p\top \to pX$.

Exercise 7.3.2 Prove the following propositions.

a) $PE \to \forall X\,Y : Prop.\ \ X \wedge Y \leftrightarrow X{=}\top \wedge Y{=}\top$

b) $PE \to \forall X\,Y : Prop.\ \ X \wedge Y \leftrightarrow \forall p : Prop \to Prop \to Prop.\ \ p\top\top \to pXY$

7.4 More about Tactics

We introduces more features of Coq's tactic language that help us with the synthesis of proof terms.

Auto and Generalize

The auto tactic solves trivial goals that don't require lemmas. Here is an example.

```
Lemma ex_auto (X Y : Prop) (p : Prop -> Prop) : PE -> X -> p(X -> Y) -> p(Y).
intros X Y p pe x. apply pe. apply conj ; auto. Qed.
```

Print the proof term to see the proof terms synthesized by *auto*. Make sure you understand how the proof works, it demonstrates an interesting pattern.

The next lemma is a variation of *ex_auto*. There is the difficulty that the claim is not in the form required for the application of the proof of *PE*. The problem can be solved with the proof *push* from §7.2 or with the tactic *assert*. Another possibility is the use of the tactic *generalize*.

```
Lemma ex_generalize (X Y : Prop) (p : Prop -> Prop) : PE -> p(X -> Y) -> X -> p(Y).
intros X Y p pe H x. generalize H. apply pe. apply conj ; auto. Qed.
```

Print the proof term to see the part synthesized by *generalize*.

The proof of the next lemma illustrates the interplay of semicolon and auto. Do the proof without auto and without semicolon to see how an explicit construction of the proof looks like. Moreover, look at the lemma itself: It characterizes propositional extensionality without equality.

```
Lemma ex_PE_Leibniz :
PE <-> forall (p : Prop -> Prop) (X Y : Prop),  (X -> Y) -> (Y -> X) -> p X -> p Y.
apply conj.
intros pe p X Y A B. apply pe. apply conj; auto.
intros H X Y A. intros p. apply (H p) ; apply A; auto. Qed.
```

We now give a shortened version of the proof *cantor''*. Since both branches of the case analysis have the same proof, we use semicolon to avoid the repetition. Moreover, we use *auto* to finish off the proof. Finally, we use the tactic *generalize* in place of the lemma *push*.

```
Theorem cantor''' (X : Type) (f : X -> X -> Prop) : exists g, forall x, f x <> g.
intros X f. pose (D := fun z => ~f z z). apply (ex_intro D). intros x e.
apply (xm_false (D x)) ; intros B ; generalize B ; unfold D ; apply e ; auto.  Qed.
```

Repeat

Our next example is *Kaminski's equation*[1], which takes the form

$$f(f(fx)) = fx$$

where $f : Prop \to Prop$ and $x : Prop$. This equation holds for all f and all x. This can be shown by a simple case analysis. There are only three possibilities for f: Either f is a constant function, or the identity, or the negation function. In each case the equation holds. Given our current lemmas, formalizing this proof requires work. There is, however, another, more low level proof that simply applies *PCA* to the terms x, $f\bot$, and $f\top$. This leaves us with eight cases, each of which can be solved with the following strategy: First apply the equation for x, then apply the equations for $f\bot$ and $f\top$ until the case is solved. Since Coq's tactic language can express this proof plan, we obtain a short but informative proof script.

```
Lemma Kaminski (x : Prop) (f :  Prop -> Prop) : PCA -> f (f (f x)) = f x.
intros x f pca.
apply (pca x); apply (pca (f False)); apply (pca (f True));
  intros c b a; apply a; repeat (apply b || apply c).
Qed.
```

[1] The equation was proposed by Mark Kaminski in 2004 as a challenge problem for an equational proof system. This led to the discovery of a decidable fragment of simply typed higher order logic.

Keep in mind that the real proof is the proof term constructed by the proof script. Use the print command to see the proof term, which is quite large.

Unfold and Exact

We have already seen the unfold tactic in the proof *cantor''* in §7.2. Here is a cooked up example that once more demonstrates the use of the unfold tactic.

```
Section Unfold.
Variable A B C : Prop.
Definition hidden := B=A.
Lemma unfold_test : A=C -> B=C -> hidden.
intros e e'. unfold hidden. apply e. apply e'. Qed.
End Unfold.
```

The unfold tactic delta reduces the name given. If we remove the command *unfold hidden* in the proof above, the command *apply e* fails to synthesize the predicate needed for the application of *e*. Step through the script to see what happens.

The unfold tactic is only needed when we exploit the smart features of the apply tactic. The proof

```
intros e e'.  apply (e (fun A => B=A) e').
```

goes through as is since it employs a proof term that gives *e* the predicate needed. One would now expect that the proof

```
apply (fun e e' => e (fun A => B=A) e').
```

goes through as well. This not the case since apply tries to be smart where it shouldn't. If we replace *apply* with *exact*, the proof goes through.

```
exact (fun e e' => e (fun A => B=A) e').
```

Use the exact tactic if you think you have a proof term that solves the goal and the apply tactic fails.

Pattern

With the pattern tactic you can factor a claim s with a given term t into $(\lambda x.u)t$ where u is obtained from s by replacing all occurrences of t with the variable x. The pattern tactic may help you to understand how the apply tactic synthesizes the predicates needed for rewriting with an equation. Here is an example.

```
Lemma pattern_demo (X : Type) (x y : X) :
(forall p : X -> Prop, p x -> p y) -> forall p : X -> Prop, p y -> p y -> p x.
intros X x y H. pattern y. apply H. auto. Qed.
```

7.5 Functional Extensionality

In mathematics one commonly assumes that two functions are equal if and only if they are equal for all arguments. This assumption is known as **functional extensionality**. We formulate functional extensionality as follows:

```
Definition FE : Prop := forall (X Y : Type) (f g : X -> Y), (forall x, f x = g x) -> f = g.
```

It is known that $PCA \to FE$ is not provable. So we have to assume FE if we want to use it. Note that FE is the converse of eq_lc. The combination of FE and eq_lc looks as follows.

```
Lemma felc {X Y : Type} {f g : X -> Y} : FE -> (f=g <-> forall x, f x = g x).
intros X Y f g fe. apply conj.
inap. intros x. apply eq_intro.
apply fe. Qed.
```

A prominent consequence of FE is the **eta law**:

```
Lemma eta {X Y : Type} {f : X -> Y} : FE -> f = fun x => f x.
intros X Y f. inap. intros x. apply eq_intro. Qed.
```

Example 7.5.1 Suppose you know that p is a commutative operation. Then it's clear that $qp \to q(\lambda xy.pyx)$ holds. The proof of this fact requires FE.

```
Lemma ex_com X (q : (X -> X -> Prop) -> Prop) (p : X -> X -> Prop) :
FE -> (forall x y, p x y = p y x) -> q p -> q (fun x y => p y x).
intros X q p fe H. apply fe. intros x. apply fe. auto. Qed.
```

Exercise 7.5.2 Following Henkin, Andrews studies a logic where all logical operations are expressed with equality. Here are some of his definitions:

```
Definition htrue : Prop := @eq Prop = @eq Prop.
Definition hfalse : Prop := (fun x : Prop => htrue) = fun x : Prop => x.
Definition hnot  : Prop -> Prop := eq hfalse.
Definition hand (X Y : Prop) : Prop := (fun p => p htrue htrue) = fun p => (p X Y : Prop).
Definition hall X  : (X -> Prop) -> Prop := eq (fun x => htrue).
```

Note the **type cast** $(pXY:Prop)$ in the definition of *hand*. It tells the type checker that the result type of p is *Prop*. We use the cast so that we don't have to give the full type of p where it is introduced as an argument variable. Prove the following lemmas. The unfold tactic may be helpful.

```
Lemma Htrue : htrue <-> True.
Lemma Hfalse : hfalse <-> False.
Lemma Hnot X : PE -> (hnot X <-> ~X).
Lemma Hand X Y : PE -> (hand X Y <-> X /\ Y).
Lemma Hall X p : PE -> FE -> (hall X p <-> all p).
```

Exercise 7.5.3 Prove the following propositions:

a) $\forall q : Prop\ \forall X : Type\ \forall p : X \to Prop.\ \ XM \to (q \leftrightarrow \forall x.\, px) \to (\neg q \leftrightarrow \exists x.\neg px)$.

b) $\forall X\, Y : Type\ \forall f\, g : X \to Y.\ \ XM \to FE \to (f \neq g \leftrightarrow \exists x.\ fx \neq gx)$.

8 Examples from Set Theory

In this chapter we represent sets and relations as functions and show some properties that are familiar from set theory. We study choice functions, Skolem functions, inverse functions, and transitive closure. We make frequent use of the extensionality assumptions.

8.1 Sets

A **predicate** is a function that yields a proposition once it is given all arguments. Predicates of a type $s \to Prop$ can be understood as sets whose elements are elements of s, and predicates of a type $s_1 \to \cdots \to s_n \to Prop$ where $n \geq 2$ can be understood as relations whose elements are n-ary tuples with components from $s_1, \ldots, s_n$. We define sets, set membership, set union, and set emptyness as follows.

```
Definition set (X : Type) : Type := X -> Prop.
Definition mem {X : Type} : X -> set X -> Prop := fun x A => A x.
Definition union {X : Type} : set X -> set X -> set X := fun A B x => A x \/ B x.
Definition empty {X : Type} : set X -> Prop := fun A => ~ ex A.
```

This give us a typed representation of sets. We can think of the elements of *set X* as the sets whose elements are in X. An important assumption about sets is **set extensionality**: Two sets are equal if they have the same elements.

```
Definition SE : Prop := forall (X : Type) (A B : set X),
(forall x, mem x A <-> mem x B) -> A=B.
```

From set extensionality it follows that sets don't come with an order for their elements. Set extensionality follows from propositional and functional extensionality.

```
Lemma pe_fe_se : PE -> FE -> SE.
intros pe fe X A B H. apply fe. intros x. apply pe. apply H. Qed.
```

Conversely, propositional extensionality follows from set extensionality.

```
Lemma se_pe : SE -> PE.
intros se X Y H.
```

```
assert (A : (fun _ : Prop => X) = fun _ => Y).
apply se. intros x. apply H.
apply (eq_lc True A). Qed.
```

We take the opportunity and introduce the tactic *refine*. The refine tactic generalizes the exact tactic in that it introduces a subgoal if it cannot derive an underspecified argument. Using *refine*, we can prove $SE \to PE$ as follows.

```
Lemma se_pe' : SE -> PE.
intros se X Y H.
refine (eq_lc True (_ : (fun _ => X) = fun _ => Y)).
apply se. intros x. apply H. Qed.
```

Here is a function that yields for x the singleton set $\{x\}$:

```
Definition  sing {X : Type} : X -> set X  := eq.
```

Note that the singleton function is exactly the equality function. It is still useful to have the separate name *sing* since it makes explicit that the predicates used are seen as sets. We prove a little fact about singletons and set union.

```
Lemma set_sing_union {X : Type} (x : X) : SE -> union (sing x) (sing x) = sing x.
intros X x se. apply se. intros y. apply conj.
inap ; auto.
inap. apply or_introl. apply eq_intro. Qed.
```

What is remarkable about the proof is that the many delta and beta reductions needed to unfold the definitions for sets are done automatically. Step through the proof to appreciate this fact. That the reduction steps happen automatically is due to the typing rules *Pre* and *Red* of the calculus of constructions (cf. §2.12). When one work on a proof like this, it is sometimes helpful to use the unfold tactic so that one sees what to do next.

Exercise 8.1.1 Define the following set operations.

```
Definition intersection {X : Type} : set X -> set X -> set X
Definition complement {X : Type} : set X -> set X
Definition difference {X : Type} : set X -> set X -> set X
Definition subset {X : Type} : set X -> set X -> Prop
Definition disjoint {X : Type} : set X -> set X -> Prop
```

Exercise 8.1.2 Define power set and set union and show $A = \bigcup_{X \in \wp A} X$.

```
Definition power {X : Type} : set X -> set (set X)
Definition set_union {X : Type} : set (set X) -> set X
Lemma power_union {X : Type} (A : set X) : PE -> FE -> A = set_union (power A).
```

8.2 Choice Functions and Skolem Functions

The axiom of choice was formulated in 1904 by Ernst Zermelo. It has some amazing consequences such as the well-ordering theorem. Although originally controversial, the axiom of choice is now used without reservation by most mathematicians. We formulate the axiom of choice as follows.

```
Definition CF X := exists C: (X->Prop)->X, forall p : X -> Prop, ex p -> p(C p).
Definition AC := forall X, inhab X -> CF X.
```

CF X says that there is a **choice function** C that yields for every predicate p on X an element Cp of X such that p holds for Cp if p holds for some x in X. In set-theoretic language one says that C selects an element of every nonempty subset p of X. The axiom of choice is the proposition AC. AC is not provable but assuming AC is consistent.

There are variations of the theme. The proposition *SF X Y* says that for every total relation from X to Y there exists a **Skolem function** from X to Y.

```
Definition SF X Y := forall r: X->Y->Prop,
(forall x, exists y, r x y) -> exists f, forall x, r x (f x).
```

The existence of a choice function for Y implies the existence of Skolem functions from X to Y.

```
Lemma CF_SF X Y : CF Y -> SF X Y.
intros X Y cf. apply cf. intros C A r H.
apply (ex_intro (fun x => C(r x))). intros x. apply A. apply H. Qed.
```

Note that the proof idea is straightforward: Given a choice function C, $\lambda x.C(rx)$ is a Skolem function as required.

The existence of Skolem functions from $(X \to Prop)$ to X implies the existence of a choice function for X provided X is inhabited and excluded middle is assumed.

```
Lemma SF_CF X : inhab X -> XM -> SF (X->Prop) X -> CF X.
intros X inh xm sf. apply (sf (fun p x => ex p -> p x)).
intros p. apply (proj2 (ex_imp xm _ inh _ _)). apply I. Qed.
```

Recall that proof *ex_imp* is defined in Figure 6.1. The proof obtains the choice function as the Skolem function for the total relation $\lambda px.\, ex\, p \to px$.

The proofs *CF_SF* and *SF_CF* demonstrate the power of Coq's approach to logic. Try to redo the proofs on your own. Doing this kind of abstract proofs with Coq is easier than doing them on paper. Also it is more rewarding since Coq makes sure that the proofs are correct. Here is one more example.

```
Lemma AC_Skolem : XM -> (AC <-> forall X, inhab X -> SF (X -> Prop) X).
intros xm. apply conj.
intros ac X inh. apply CF_SF. apply (ac X inh).
intros H X inh. apply (SF_CF X inh xm). apply (H X inh). Qed.
```

Exercise 8.2.1 Prove the following propositions.

a) $\forall X.\ CF\,X \to inhab\,X$

b) $\forall X\ Y\ \forall r{:}X \to Y \to Prop.\ \ AC \to inhab\,Y \to ((\forall x\,\exists y.\ rxy) \leftrightarrow \exists f\,\forall x.\ rx(fx))$

8.3 Inverse Functions

We define injectivity and surjectivity of functions.

```
Definition injective {X Y : Type} (f : X -> Y) : Prop
:= forall x x', f x = f x' -> x=x'.

Definition  surjective {X Y : Type} (f : X -> Y) : Prop
:= forall y, exists x, f x = y.
```

A function g is inverse to a function f if g undoes f.

```
Definition inverse {X Y : Type} (f : X -> Y) (g : Y -> X) : Prop
:= forall x, g (f x) = x.

Definition invertible {X Y : Type} (f : X -> Y) : Prop
:= exists g, inverse f g.
```

We can now prove some lemmas that relate the existence of inverse functions to injectivity and surjectivity. To start with, inverse functions are surjective, and invertible functions are injective.

```
Lemma inv_surj (X Y : Type) (f : X -> Y) (g : Y -> X) : inverse f g -> surjective g.
intros X Y f g inv x. apply (ex_intro (f x)). apply inv. Qed.

Lemma inv_inj (X Y : Type) (f : X -> Y) : invertible f -> injective f.
intros X Y f. inap. intros g inv x x' e.
apply (eq_sym (inv x)). apply (eq_sym (inv x')). apply (eq_rc g e). Qed.
```

Next we show that surjective functions have at most one inverse function.

```
Lemma inv_unique  (X Y : Type) (f : X -> Y) (g : Y -> X) (h : Y -> X) : FE ->
surjective f -> inverse f g -> inverse f h -> g=h.
intros X Y f g h fe surj A B. apply fe. intros y.
apply (surj y). intros x e. apply (eq_sym e).
apply A. apply B. apply eq_intro. Qed.
```

If Skolem functions exist, every surjective function is an inverse function.

```
Lemma surj_inv (X Y : Type) (f : X -> Y) : SF Y X ->
surjective f -> exists g, inverse g f.
intros X Y f sf surj. apply (sf _ surj). Qed.
```

If Skolem functions exist and f is an injective and surjective function, then there exists a function g such that g is inverse to f and f is inverse to g.

```
Lemma bijection (X Y : Type) (f : X -> Y) : SF Y X ->
injective f -> surjective f -> exists g, inverse f g /\ inverse g f.
intros X Y f sf inj surj.
apply (sf _ surj). intros g H. apply (ex_intro g). apply conj.
intros x. apply inj. apply H.
exact H. Qed.
```

Finally, we show that every injective function is invertible provided some standard assumptions are satisfied. The proof is involved since the construction of the inverse function as a Skolem function requires a clever relation r.

```
Lemma inj_inv (X Y : Type) (f : X -> Y) : XM -> SF Y X -> inhab X ->
injective f -> invertible f.
intros X Y f xm sf inh inj.
pose (r y x := f x = y \/ ~exists x', f x' = y).
assert (A : forall y, exists x, r y x).
intros y. apply (xm (exists x', f x' = y)).
  inap. intros x H. apply (ex_intro x). apply (or_introl H).
  intros H. apply inh. intros x _. apply (ex_intro x). apply (or_intror H).
apply (sf r A). intros g B.
  apply (ex_intro g). intros x. apply (B (f x)).
    apply inj.
    inap. apply (ex_intro x). apply eq_intro. Qed.
```

8.4 Transitive Closure

The transitive closure of a binary relation r on a set X is the least transitive relation that contains r. Given a graph, the transitive closure of the edge relation of the graph is the relation that holds for two nodes x and y if and only if there is a path from x to y that contains at least one edge.

Let's carefully analyse the notion of transitive closure. Let X be a type. We can think of X as a set and of predicates $X \to X \to \mathit{Prop}$ as binary relations on X. We will do our proofs without classical assumptions (i.e., *XM*, *PE*, *FE*, *AC*). This means that our proofs are also valid for predicates that don't respect the classical assumptions. We start with the following definitions.

```
Section Transitive_Closure.
Variable X : Type.
```

```
Definition pred : Type := X -> X -> Prop.
Definition trans (p : pred) : Prop := forall x y z, p x y -> p y z -> p x z.
Definition implies (p q : pred) : Prop := forall x y, p x y -> q x y.
Definition equivalent (p q : pred) : Prop := forall x y, p x y <-> q x y.
```

If we assume *PE* and *FE*, equivalent predicates are equal.

```
Lemma equi_eq (p q : pred) : PE -> FE -> equivalent p q -> p = q.
intros p q pe fe H. apply fe. intros x. apply fe. intros y. apply pe. apply H. Qed.
```

We assume a predicate r and define two properties of predicates:

```
Variable r : pred.
Definition strong (p : pred) : Prop := forall q : pred,  trans q -> implies r q -> implies p q.
Definition tc (p : pred) : Prop := trans p /\ implies r p /\ strong p.
```

If a predicate p satisfies *tc*, we see it as a transitive closure of r. This is justified since p is transitive, is implied by r, and implies every other predicate with this two properties. There are two questions now:

1. *Existence*: Does there exist a predicate that satisfies *tc*?
2. *Uniqueness*: Are two predicates equivalent if they both satisfy *tc*?

We will answer both questions positively. This tells us that the notion of transitive closure is well-defined. Following an idea that already appears in the work of Frege, we settle the existence question with the following definition and proof.

```
Definition ftc : pred := fun x y => forall p : pred, trans p -> implies r p -> p x y.

Lemma existence : tc ftc.
apply conj.
(* trans ftc *)  intros x y z A B p tp ip. apply (tp x y z). apply A ; auto. apply B ; auto.
apply conj.
(* implies r ftc *)  intros x y A p _. inap ; auto.
(* strong ftc *)  intros p tp irp x y. inap ; auto. Qed.
```

Finally, we show that all predicates that satisfy *tc* are equivalent.

```
Lemma uniqueness (p q : pred) : tc p -> tc q -> equivalent p q.
intros p q.
inap. intros tp. inap. intros ip sp.
inap. intros tq. inap. intros iq sq.
intros x y. apply conj. apply sp ; auto. apply sq ; auto. Qed.

End Transitive_Closure.
```

9 Inductive Definitions

Inductive definitions provide a mechanism for extending the calculus of constructions with new types called inductive types. For instance, one can define an inductive type *bool* with two new values *true* and *false*, or an inductive type *nat* that has a new value for every natural number. The elements of inductive types are obtained with primitive values called constructors. Inductive definitions are accompanied by new terms that provide for constructor-based case analysis and structural recursion. Inductive definitions are a prominent feature of functional programming languages like ML or Haskell.

We have seen in §2.10 that the pure calculus of constructions comes with function types whose elements can serve as representations of the boolean values and the natural numbers. While these representations work computationally, they do not work logically. For instance, one cannot prove that *false* is different from *true*. With inductive types one gets representations that are adequate both computationally and logically. The logical features of inductive types are all obtained from their computational features.

In this chapter we restrict our interest to inductive definitions that don't require recursion. Our first example is a two-valued type bool. Then we continue with inductive definitions that realize logical operations. This prepares the switch to Coq's predefined logical operations, which are realized inductively.

9.1 Bool and Match

We start with the **inductive definition**

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

Here are the most important facts about this definition:

- The definition extends the calculus of constructions with the values *bool*, *true*, and *false*.
- *bool* is an **inductive type** that is an element of U_1.
- *true* and *false* are **constructors**.
- The elements of *bool* are exactly *true* and *false*.

- The definition extends the calculus of constructions with all values that can be obtained with *bool*, *true*, and *false* (e.g., $bool \to bool$ and $\lambda x{:}\,bool.\, x$).

Inductive types are accompanied by terms called **matches**. Matches provide for constructor-based case analysis. There is a reduction rule for matches. We demonstrate the use of matches with a function that negates boolean values.

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

```
Eval cbv in negb false.
```

true

Note that the match has two **clauses**, one for *true* and one for *false*. Here is a first proof involving an inductive type.

```
Example E1 : negb (negb true) = negb false.
apply eq_intro. Qed.
```

The keyword *Example* is a synonym for *Definition*. The proof is straightforward since both sides of the equation reduce to *true*. To make this fact visible, we insert the tactic *simpl*.

```
Example E1' : negb (negb true) = negb false.
simpl. apply eq_intro.  Qed.
```

The application of *simpl* reduces the claim to $true = true$. Note that this equation is further reducible since = is defined. You may use the tactic *cbv* to reduce the claim to normal form. You have seen *cbv* before in §2.7. The tactic *simpl* reduces matches and beta redexes and also performs delta reduction steps that introduce matches. Use the print command to see that *E1* is equated to the term *eq_intro* (*negb false*) and that *E1′* is equated to the term *eq_intro true*.

Next we define a function that embeds *bool* into *Prop*.

```
Definition boolp (x : bool) : Prop := match x with true => True | false => False end.
```

With *boolp* we show that the terms *true* and *false* denote different values.

```
Lemma true_neq_false : true <> false.
intros e. apply True_neq_False. apply (eq_rc boolp e). Qed.
```

Here is a shorter proof that exploits the definition of equality:

```
Lemma true_neq_false' : true <> false.
intros e. apply (eq_sym e boolp I). Qed.
```

If we assume *PCA*, *boolp* is surjective.

```
Lemma boolp_surjective : PCA -> forall X : Prop, exists x : bool, boolp x = X.
intros pca X. apply (pca X) ; inap.
apply (ex_intro false). apply eq_intro.
apply (ex_intro true). apply eq_intro. Qed.
```

In the next section, we will show that *boolp* is injective. Thus *boolp* is a bijection from *bool* to *Prop* if *PCA* is assumed (cf. §8.3).

We define an exclusive or function to demonstrate two notational features of Coq.

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

The definition uses a derived notation for a **cascaded match** on two boolean terms. Coq translates this notation into nested matches, as you can verify with the print command.

```
xorb = fun x y : bool => match x with
| true => match y with true => false | false => true end
| false => match y with true => true | false => false end
end
```

The vertical bar "|" in front of the first clause is optional. We only use it if we write the clauses of a match line by line.

Exercise 9.1.1 Prove $false \neq true$.

9.2 Destruct

The tactic *destruct* synthesizes matches.[1] We use *destruct* to prove the so-called induction lemma for bool.

```
Lemma bool_induction (p : bool -> Prop) : p true -> p false -> forall x : bool, p x.
intros p a b x. destruct x. apply a. apply b. Qed.
```

The lemma states that given a predicate p and proofs of $p\,false$ and $p\,true$, we can obtain a function that gives us a proof of $p\,x$ for every x in *bool*. The print command tells us that *bool_induction* is defined as the term

```
fun (p : bool -> Prop) (a : p true) (b : p false) (x : bool) =>
match x as y return p y with true => a | false => b end
```

[1] *destruct* stands for *destructure*.

The match in this term comes with a **return type specification** *as y return p y* that is required for type checking. It states the following:

1. The match has type *p x*.
2. The body of the clause for *true* has type *p true*.
3. The body of the clause for *false* has type *p false*.

The auxiliary variable *y* is local and is needed for the general case where the term the match is done on is not a variable. One speaks of a **dependent match** since the types of the clause bodies depend on the value of the term the match is done on.

Here is second example for the use of the destruct tactic.

```
Lemma negb_negb : forall x, negb (negb x) = x.
destruct x. apply eq_intro. apply eq_intro. Qed.
```

The example tells us that *destruct x* automatically inserts *intros x* if *x* is not yet introduced. The proof term constructed looks as follows (check with the print command):

```
fun x : bool => match x as y return negb (negb y) = y
with true => eq_intro true | false => eq_intro false end
```

Here are two further examples for the use of the destruct tactic.

```
Lemma negb_neq : forall x, negb x <> x.
destruct x ; simpl. apply (mt eq_sym). apply true_neq_false. apply true_neq_false. Qed.
```

```
Lemma boolp_injective (x y : bool) : boolp x = boolp y -> x=y.
destruct x ; destruct y ; simpl ; intros H.
apply eq_intro.
apply (True_neq_False H).
apply (True_neq_False (eq_sym H)).
apply eq_intro. Qed.
```

The induction lemma provides us with a proof rule that can replace the use of the tactic *match*.

```
Lemma negb_negb' : forall x, negb (negb x) = x.
apply bool_induction. apply eq_intro. apply eq_intro. Qed.
```

We define an **if-then-else notation** and prove a basic property.

```
Notation "'if' s 'then' t 'else' u" := (match s with true => t | false => u end) (at level 99).
```

```
Lemma bool_if (X : Type) (x : X) (b : bool) : (if b then x else x) = x.
intros X x b. destruct b ; apply eq_intro. Qed.
```

Exercise 9.2.1 Prove $\forall x : bool.\ \ negb\ x = negb(negb(negb\ x))$.

Exercise 9.2.2 Prove $\forall x : bool.\ \ boolp\ x = True \ \rightarrow\ x = true$.

Exercise 9.2.3 Prove the following lemmas.

```
Lemma if_negb (x : bool) :  negb x = if x then false else true.
Lemma if_xorb (x y : bool) :  xorb x y = if x then negb y else y.
```

Exercise 9.2.4 Prove the lemma *boolp_injective* with *bool_induction* in place of *match*. Without help *apply bool_induction* fails since it cannot synthesize the predicate p. This can be fixed by writing *intros x*; *pattern x* in front of the apply.

Exercise 9.2.5 Define boolean or $orb : bool \rightarrow bool \rightarrow bool$ and prove that it is commutative (i.e., $\forall x\ y.\ orb\ x\ y = orb\ y\ x$).

Exercise 9.2.6 Prove the following lemmas.

```
Lemma BCA (x : bool) : x=true \/ x=false.
Lemma bool_Kaminski (f : bool -> bool) ( x: bool) : f (f (f x)) = f x.
```

9.3 Rules for Matches on Bool

Here is the typing rule for matches on *bool*.

$$\frac{s : bool \qquad x : bool \Rightarrow t : \mathsf{U}_n \qquad u : t^x_{true} \qquad v : t^x_{false}}{\mathsf{match}\ s\ \mathsf{as}\ x\ \mathsf{return}\ t\ \mathsf{with}\ true \Rightarrow u \mid false \Rightarrow v\ \mathsf{end} : t^x_s}$$

Matches without return type specification are automatically completed with a specification as _ return t. The full expressivity of matches on *bool* is captured by the following function.

```
Definition bool_match (f : bool -> Type) (u : f true) (v : f false) (x : bool) : f x
:= match x as x' return f x' with true => u | false => v end.
```

The function *bool_induction* defined in §9.2 can be obtained as a special case.

```
Check fun p : bool -> Prop => bool_match p.
forall p : bool → Prop, p true → p false → forall x : bool, p x
```

The example type checks with the typing rule *Cum* and the second clause of the definition of the subtyping relation (see §2.12).

Here are the reduction rules for matches on *bool*:

$$\mathsf{match}\ true\ \mathsf{as}\ x\ \mathsf{return}\ t\ \mathsf{with}\ true \Rightarrow u \mid false \Rightarrow v\ \mathsf{end} \ \rightsquigarrow\ u$$
$$\mathsf{match}\ false\ \mathsf{as}\ x\ \mathsf{return}\ t\ \mathsf{with}\ true \Rightarrow u \mid false \Rightarrow v\ \mathsf{end} \ \rightsquigarrow\ v$$

9.4 Propositional Conditional

We have already seen that match can express boolean conditionals (see if-then-else notation in §9.2). A **propositional conditional** is like a boolean conditional but takes its decision with respect to the truth value of a proposition. Here is a proposition that states that there is a function that provides propositional conditionals.

```
Definition IFP : Prop := exists ifp : forall X : Type, Prop -> X -> X -> X,
forall (X : Type) (x y : X), ifp X True x y = x /\ ifp X False x y = y.
```

We show that such a function *ifp* exists if and only if the embedding *boolp* is invertible. We use two generally useful lemmas *P1* and *P2*.

```
Lemma P1 {X Y : Type}{p q : X -> Y -> Prop} : (forall x y, p x y /\ q x y) -> forall x y, p x y.
intros X Y p q H x y. apply (H x y). auto. Qed.

Lemma P2 {X Y : Type}{p q : X -> Y -> Prop} : (forall x y, p x y /\ q x y) -> forall x y, q x y.
intros X Y p q H x y. apply (H x y). auto. Qed.

Lemma IFP_inv_boolp : IFP <-> invertible boolp.
apply conj.
inap. intros ifp A.
  apply (ex_intro (fun p => ifp bool p true false)).
  red. destruct x. apply (P1 (A bool)). apply (P2 (A bool)).
inap. intros propb A.
  apply (ex_intro (fun _ p x y => if (propb p) then x else y)).
  intros X x y. apply conj.
    assert (B : propb True = true). apply (A true). apply B. apply eq_intro.
    assert (B : propb False = false). apply (A false). apply B. apply eq_intro. Qed.
```

Next we show that *boolp* is invertible if we assume excluded middle and Skolem functions from *Prop* to *bool*.

```
Lemma SF_inv_boolp : XM -> SF Prop bool -> invertible boolp.
intros xm sf. apply (sf (fun p x => p /\ x=true \/ ~p /\ x=false)).
intros p. apply (xm p) ; intros A.
  apply (ex_intro true). apply or_introl. apply conj. apply A. apply eq_intro.
  apply (ex_intro false). apply or_intror. apply conj. apply A. apply eq_intro.
intros propb A. apply (ex_intro propb). red. destruct x.
  apply (A True).  inap. auto. inap. inap. apply I.
  apply (A False). inap. inap. inap. auto. Qed.
```

Note the use of the tactic *red*. This tactic reduces the claim until it turns into an implication or a universal quantification. The reduction is sometimes needed so that the tactics *destruct* and *fix* become applicable. The lemma can also be obtained with the lemmas *boolp_injective* and *inj_inv*. Finally we show that Skolem functions from *Prop* to *bool* exist if we assume *PCA* and the invertibility of *boolp*.

```
Lemma inv_boolp_SF : PCA -> invertible boolp-> SF Prop bool.
intros pca. inap. intros propb A r B.
assert (AT := A true). simpl in AT.
assert (AF := A false). simpl in AF.
apply (ex_intro (fun p => propb (r p true))).
intros p. apply (B p). intros x.
destruct x ; apply (pca (r p true)) ; intros D C.
  apply False_elim. apply (eq_sym D). apply C.
  apply D. apply AT. apply C.
  apply D. apply AF. apply C.
  apply D. apply AT. apply D. apply I. Qed.
```

9.5 Polymorphic Pairs

Here is an inductive definition that realizes polymorphic pairs:

```
Inductive pair (X Y : Type) : Type :=
| cons : X -> Y -> pair X Y.
```

The definition introduces the **inductive function** *pair* and the constructor *cons*:

$$pair \;:\; Type \to Type \to Type$$
$$cons \;:\; \forall X\, Y : Type.\; X \to Y \to pair\, X\, Y$$

Given two types X and Y, the inductive function yields an inductive type $pair\, X\, Y$. Thus the definition introduces a family of inductive types rather than a single inductive type. We declare the first two arguments of *cons* to be implicit.

```
Implicit Arguments cons [X Y].
```

The definitions of the projection functions are straightforward.

```
Definition fst {X Y : Type} (p : pair X Y) : X := match p with cons x y => x end.
Definition snd {X Y : Type} (p : pair X Y) : Y := match p with cons x y => y end.
```

Here is a lemma about pairs. It says that one can reconstruct a pair from its first and second component.

```
Lemma cons_fst_snd (X Y : Type) (p : pair X Y) : cons (fst p) (snd p) = p.
intros X Y. destruct p as [x y]. apply eq_intro. Qed.
```

Note the “as” clause that appears with the destruct tactic. It specifies the local variables of the clause of the match. The synthesized proof term looks as follows.

```
fun (X Y : Type) (p : pair X Y) =>
match p as p' return cons (fst p') (snd p') = p'
with cons x y => eq_intro (cons x y) end
```

Here is a proof that shows that the constructor *cons* is injective in both arguments.

```
Lemma cons_injective (X Y : Type) (x x' : X) (y y' : Y) :
cons x y = cons x' y' -> x=x' /\ y=y'.
intros X Y x x' y y' e.  apply conj. apply (eq_rc fst e). apply (eq_rc snd e). Qed.
```

9.6 Inductive Predicates

Recall that we are in a setting where most logical operations are defined. It turns out that the derived logical operations can also be realized with inductive predicates (i.e., inductive functions that eventually yield an inductive proposition). To realize a logical operation, we need the following:

- A function that realizes the operation syntactically. For conjunction this is a predicate $and : Prop \to Prop \to Prop$.
- Functions that realize the introduction rules of the operation. For conjunction this is a function $conj : \forall X\, Y : Prop.\ X \to Y \to and\, X\, Y$.
- A function that realizes the elimination rule for the operation. For conjunction this is a function $and\_elim : \forall X\, Y\, Z : Prop.\ and\, X\, Y \to (X \to Y \to Z) \to Z$.

It turns out that every inductive definition that introduces an inductive predicate realizes some logical operation (not necessarily one we have seen so far):

- The inductive predicate realizes the operation syntactically.
- The constructors realize the introduction rules of the operation.
- The match accompanying the definition realizes the elimination rule of the operation.

9.6.1 Conjunction

We start with an inductive definition that realizes conjunction. Since we will use the names we have used so far, our previous definitions should not be active.

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

The definition introduces the inductive function

$$and\ :\ Prop \to Prop \to Prop$$

and the constructor

$$conj \ : \ \forall X\, Y : Prop.\ \ X \to Y \to and\ X\ Y$$

Clearly, *and* realizes conjunctions syntactically, and *conj* realizes the introduction rule for conjunctions. Note that *and* is a function that yields inductive types in *Prop*. The elimination function for conjunctions can be expressed with the match accompanying the definition.

```
Definition and_elim (X Y Z : Prop) : and X Y -> (X -> Y -> Z) -> Z
:= fun a f => match a with conj x y => f x y end.
```

The match exploits the fact that every value of a type $and\,X\,Y$ must be obtained with the constructor *conj*. It is also possible to obtain the elimination function with a script.

```
Lemma and_elim' (X Y Z : Prop) : and X Y -> (X -> Y -> Z) -> Z.
intros X Y Z a f. destruct a as [x y]. apply (f x y). Qed.
```

Use the print command to see that *and_elim* and *and_elim'* are equated to the same proof term. We prove that *and* is commutative:

```
Lemma and_com (X Y : Prop) : and X Y -> and Y X.
intros X Y a. destruct a as [x y]. apply (conj _ _ y x). Qed.
```

The underlines in the application of *conj* are necessary since the respective arguments are not declared to be implicit.

The intros tactic can synthesize matches. The idea is to immediately match on a variable once it is introduced. To this purpose one writes a **pattern** specifying the local variables of the clauses in place of the variable to be matched on.

```
Lemma and_com' (X Y : Prop) : and X Y -> and Y X.
intros X Y [x y]. apply (conj _ _ y x). Qed.
```

Use the print command to see that *and_com* and *and_com'* are equated to the same proof term.

9.6.2 Disjunction

Here is an inductive definition that realizes disjunction.

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

The constructors *or_introl* and *or_introl* provide the introduction rules for disjunctions. The elimination rule can be obtained as follows.

```
Definition or_elim (X Y Z : Prop) : or X Y -> (X -> Z) -> (Y -> Z) -> Z
:= fun o f g => match o with or_introl x => f x | or_intror y => g y end.
```

Alternatively, we can use a script

```
Lemma or_elim' (X Y Z : Prop) : or X Y -> (X -> Z) -> (Y -> Z) -> Z.
intros X Y Z o f g. destruct o as [x|y]. apply (f x). apply (g y). Qed.
```

This time the as clause of the destruct command specifies local variables for two clauses since there are two constructors for disjunctions. The variables for the two clauses are separated by a vertical bar "|". Again it is possible to obtain the match with the intros tactic.

```
Lemma or_elim'' (X Y Z : Prop) : or X Y -> (X -> Z) -> (Y -> Z) -> Z.
intros X Y Z [x|y] f g. apply (f x). apply (g y). Qed.
```

Here is a proof that *or* is commutative.

```
Lemma or_com (X Y : Prop) : or X Y -> or Y X.
intros X Y [x|y]. apply (or_intror _ _ x). apply (or_introl _ _ y). Qed.
```

9.6.3 True and False

Here are inductive definitions that realize the logical constants $\top$ and $\bot$.

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

Recall that there is no elimination rule for $\top$. The elimination rule for $\bot$ can be obtained as follows.

```
Definition False_elim (X : Prop) : False -> X
:= fun f => match f with end.
```

While this looks strange at first it is actually straightforward. Since *False* has no constructors, a match on a term of type *False* has no clauses. Thus every return type is possible (since every clause satisfies the return type).

We can synthesize the elimination function with a very short proof script.

```
Lemma False_elim' (X : Prop) : False -> X.
intros X []. Qed.
```

9.6.4 Existential Quantification

The inductive realization of existential quantification is now routine.

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

Definition ex_elim (X : Type) (p : X -> Prop) (Z : Prop) : ex X p -> (forall x, p x -> Z) -> Z
:= fun e f => match e with ex_intro x a => f x a end.
```

Here is a proof script that synthesizes the elimination function.

```
Lemma ex_elim' (X : Type) (p : X -> Prop) (Z : Prop) : ex X p -> (forall x, p x -> Z) -> Z.
intros X p Z [x a] A. apply (A x a). Qed.
```

9.6.5 Equality

Finally, we give an inductive definition that realizes equality.[2]

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

The inductive predicate and the constructor introduced by this definition have the right types.

$$\begin{aligned} eq &: \forall X : Type.\ X \to X \to Prop \\ refl\_equal &: \forall X : Type\ \forall x : X.\ eq\, X\, x\, x \end{aligned}$$

We realize an elimination function with a proof script.[3]

```
Lemma eq_elim (X : Type) (x y : X) (p : X -> Prop) : eq X x y -> p x -> p y.
intros X x y p e a. destruct e. apply a. Qed.
```

Step through the script to see what is happening. The equational constraint imposed by the return type of *refl_equal* gets realized once the clause of the match is entered. The proof term synthesized looks as follows:

```
fun (X : Type) (x y : X) (p : X -> Prop) (e : eq X x y) (a : p x) =>
match e in (eq _ _ z) return pz
with refl_equal => a end
```

The match comes with a new form of return type specification, which takes care of the equational constraint in the return type of the constructor *refl_equal*. The variable z is local. The specification has the effect that the return type appears as px in the clause. Don't worry about the details, it suffices if you understand intuitively what is happening when you step through the proof script.

[2] We use the name *refl_equal* rather than *eq_intro* to agree with Coq's predefined equality.

[3] Compared with the elimination lemma in §7.1 we have switched px with py. This way we get the simpler proof term. We can obtain the elimination lemma defined in §7.1 from the elimination lemma defined here with $p := \lambda z.\, pz \to px$.

Exercise 9.6.1 Prove the following lemma in three ways: With a script using *destruct*, with a script applying *eq_elim* defined above, and with a proof term.

```
Lemma eq_elimr (X : Type) (x y : X) (p : X -> Prop) : eq X x y -> p y -> p x.
```

9.7 Coq's Predefined Logical Operations

We now abandon our definitions of the derived logical operations and switch to the definitions provided by Coq. Coq's definitions agree with the definitions discussed in the previous section. In addition, Coq defines negation and equivalence as follows.

```
Definition not (X : Prop) : Prop := X -> False.
Definition iff (X Y : Prop) : Prop := and (X -> Y) (Y -> X).
```

Moreover, all the notations we used for the logical operations are predefined in Coq. For reasons of convenience and readability, Coq provides tactics for all introduction rules except I.

split	⇝	apply conj
left	⇝	apply or_introl
right	⇝	apply or_intror
exists t	⇝	apply (ex_intro t)
reflexivity	⇝	apply refl_equal

The inductively defined operations can all be eliminated with the destruct tactic. In addition, the following tactics are provided.

rewrite t	⇝	elimination of =, applies equation from left to right
rewrite<- t	⇝	elimination of =, applies equation from right to left
absurd t	⇝	creates subgoals for $\neg t$ and t

The command *rewrite*<- *t* is equivalent to *destruct t*.

It's time for examples. Open a new Coq shell. Check with the print command that Coq predefines the derived logical operations as we have claimed. Then step through the following examples.

```
Example E1 ( X Y Z : Prop) : X /\ (Y \/ Z) -> X /\ Y \/ X /\ Z.
intros X Y Z [x [y|z]].
left. apply (conj x y).
right. apply (conj x z). Qed.
```

Note the **nested pattern** in the intros command. It synthesizes a nested match. The next example uses a nested pattern to eliminate two existential quantifications:

```
Example E2 (X Y : Type) (p : X -> Y -> Prop) :
(exists x, exists y, p x y) -> exists y, exists x, p x y.
intros X Y p [x [y A]]. exists y. exists x. apply A. Qed.
```

The following example eliminates an equation with a match.

```
Lemma E3 (X : Type) (x y : X) :  x=y <-> forall r : X -> X -> Prop, (forall z, r z z) -> r x y.
intros X x y. split.
intros [] r H. apply H.
intros H. apply H. reflexivity. Qed.
```

The tactic *tauto* knows about the predefined logical constants and solves goals whenever they can be solved with intros, reflexivity, and the introduction and elimination rules for implication, conjunction, and disjunction. In contrast to *auto*, *tauto* fails if it cannot solve a goal. From now on, *tauto* will be our preferred automation tactic. Here are two proofs of Cantor's theorem that use *tauto*. Neither of the proofs uses a lemma.

```
Definition Cantor : Prop := forall (X : Type) (f : X -> X -> Prop), exists g, forall x, f x <> g.

Theorem cantor : Cantor.
intros X f.
assert (A : forall x, ~ (f x x <-> ~f x x)). tauto.
pose (g x := ~ f x x). exists g. intros x e.
apply (A x).  pattern (f x) at 1. rewrite e. unfold g. tauto. Qed.

Theorem cantor' : Cantor.
intros X f.
pose (g x := ~ f x x). exists g. intros x e.
absurd (~ g x).
intros A. generalize A. unfold g. rewrite e. tauto.
intros A. generalize A. unfold g. rewrite e. tauto. Qed.
```

The first proof first establishes the *iff_circuit* lemma to convey the main idea of the proof. Note that the pattern tactic is used such that only the first occurrence of fx is patterned out. The second proof follows the informal presentation of the proof in §7.2.

Here are further examples involving equality.

```
Definition XM : Prop := forall X : Prop, X \/ ~X.

Definition PE : Prop := forall X Y : Prop, (X <-> Y) -> X=Y.

Definition PCA : Prop := forall X : Prop, X=False \/ X=True.
```

```
Lemma pca_xm : PCA -> XM.
intros pca X. destruct (pca X) as [e|e] ; rewrite e ; tauto. Qed.

Lemma pca_pe : PCA -> PE.
intros pca X Y.
destruct (pca X) as [a|a] ; rewrite a ;
destruct (pca Y) as [b|b] ; rewrite b ;
tauto. Qed.
```

The tactic *subst* checks the assumptions for equations of the form $x = t$ or $t = x$ where the variable x is not free in t and eliminates x by replacing it with t. By using *subst* we can shorten the proof just given.

```
Lemma pca_pe' : PCA -> PE.
intros pca X Y. destruct (pca X) ; destruct (pca Y) ; subst ; tauto. Qed.
```

Note that the script does not specify local variables for the clauses introduced by *destruct*. This way we leave the choice of the local variables to Coq.

The type *bool* is predefined in Coq. For reasons of backward compatibility, Coq places *bool* into a special universe *Set* reserved for inductive types. *Set* is contained in U_1 but does not contain *Prop*.

Here is a proof of Kaminski's equation.

```
Lemma bca (x : bool) : x=true \/ x=false.
intros [] ; tauto. Qed.

Lemma Kaminski (x : bool) (f : bool -> bool) : f (f (f x)) = f x.
intros [] f ; destruct (bca (f true)) as [a|a] ; destruct (bca (f false)) as [b|b] ;
repeat (reflexivity || rewrite a || rewrite b). Qed.
```

The proof can be shortened by using the tactic *congruence*, which tries to solve a goal by rewriting with equations that appear as assumptions.

```
Lemma Kaminski' (x : bool) (f :  bool -> bool) : f (f (f x)) = f x.
intros [] f ; destruct (bca (f true)) ; destruct (bca (f false)) ; congruence. Qed.
```

Exercise 9.7.1 It is enlightening to prove that Coq's predefined logical operations are equivalent to the operations we used before. Prove the following.

```
Lemma L1 :  False <-> forall X : Prop, X.
Lemma L2 (X Y : Prop) :  X /\ Y <-> forall Z : Prop, (X ->Y -> Z) -> Z.
Lemma L3 (X Y : Prop) :  X \/ Y <-> forall Z : Prop, (X -> Z) -> (Y -> Z) -> Z.
Lemma L4 (X : Type) (p : X -> Prop) :  ex p <-> forall Z : Prop, (forall x, p x -> Z) -> Z.
Lemma L5 (X : Type) (x y : X) :  x=y <-> forall p : X -> Prop, p y -> p x.
```

9.8 Remarks

We have shown before that different realizations of the logical operations are equivalent if they support the intuitionistic introduction and elimination rules (see §3.3 and §4.10). Thus the **inductive logical operations** predefined in Coq are equivalent to the **impredicative logical operations** we have used so far. The term impredicative refers to the fact that the definitions used so far rely on the impredicativity of the universe *Prop* (see §2.12).

10 Natural Numbers

In this chapter we realize the natural number with an inductive type that comes with structural recursion. With structural recursion we can express arithmetic operations and inductive proofs.

10.1 Definition

We define an inductive type whose elements represent the natural numbers.

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

The elements of *nat* are the values we can obtain with the constructors O and S:

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

We say that the constructor S yields the **successor** of a natural number. We define a function that yields the **predecessor** of a natural number.

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

Note that the second clause involves a local variable x' that is equated to the predecessor of x in case the second clause applies. We prove that the predecessor function is inverse to the successor function.

```
Lemma inverse_S_pred (x : nat) : pred (S x) = x.
reflexivity. Qed.
```

Next we prove that the successor function is injective.

```
Lemma S_injective {x y : nat} : S x = S y -> x=y.
intros x y e. apply (f_equal pred e). Qed.
```

Note the use of the predefined proof *f_equal*, which is equivalent to the proof *eq_rc* defined in §7.1. Next we define a function that tests whether a number is O and prove that O is not in the range of S.

```
Definition iszero (x : nat) : Prop := match x with O => True | S _ => False end.

Lemma S_neq_O (x : nat) : S x <> O.
intros x e.
assert (A : False = True). apply (f_equal iszero e).
rewrite A. apply I. Qed.
```

An **inductive data type** is an inductive type that is not an element of the universe *Prop*. The constructors of inductive data types are always injective and mutually exclusive. The tactics *injection* and *discriminate* try to synthesize the respective proofs. Here are examples.

```
Goal false <> true. discriminate. Qed.
Goal S(S O) <> S O. discriminate. Qed.
Goal forall x y, S (S x) = S (S (S y)) -> x = S y. intros x y e. injection e. tauto. Qed.
```

The command *Goal* is like *lemma* but leaves it to Coq to generate a name for the proof. It is enlightening to prove $S(S\,O) \neq O$ by hand.

```
Goal S(S O) <> S O.
intros e.
assert (A : False = True).
apply (f_equal (fun x => match x with S O => True | _ => False end) e).
rewrite A. apply I. Qed.
```

The next proof shows that *bool* is different from *nat*.

```
Lemma bool_neq_nat :  bool <> nat.
intros e.
assert (A : forall x y z : bool, x=y \/ x=z  \/ y=z). intros [] [] [] ; tauto.
revert A. pattern bool. rewrite e. intros A.
destruct (A O (S O) (S (S O))) as [B|[B|B]] ; discriminate B. Qed.
```

The proof first assumes $bool = nat$. It then establishes a property A of *bool*. Using the assumption $bool = nat$ property A is rewritten into a property for *nat*. Since the property doesn't hold for *nat*, *False* can be shown.

Exercise 10.1.1 Prove $\forall x : nat.\ x = O \vee \exists y.\ x = S\,y$.

Exercise 10.1.2 Prove *surjective pred* and $\neg$ *injective pred*.

Exercise 10.1.3 Prove $\forall x\,y : nat.\ S(S\,x) = S(S(S\,y)) \to x = S\,y$ without using the tactic *injection*. Then compare the proof term obtained with *injective* and with the proof term obtained with your proof.

Exercise 10.1.4 Prove $(O, true) \neq (O, false)$ with discriminate and without discriminate. Pairing is defined inductively as in §9.5.

Exercise 10.1.5 Prove $\forall x\,y.\ (x, y) = (y, O) \to x = O$ with and without injection.

10.2 Rules for Matches on Nat

Here is the typing rule for matches on *nat*.

$$\frac{s : nat \qquad x : nat \Rightarrow t : \mathsf{U}_n \qquad u : t^x_O \qquad y : nat \Rightarrow v : t^x_{S\,y}}{\mathsf{match}\ s\ \mathsf{as}\ x\ \mathsf{return}\ t\ \mathsf{with}\ O \Rightarrow u \mid S\,y \Rightarrow v\ \mathsf{end} : t^x_s}$$

Matches on *nat* can be provided with the following function.

```
Definition nat_match (f : nat -> Type) (u : f O) (v : forall y : nat, f (S y)) (x : nat) : f x
:= match x as x' return f x' with O => u | S y => v y end.
```

The predecessor function can be obtained with *nat_match* as follows:

```
Definition pred' : nat -> nat := nat_match (fun _ => nat) O (fun y => y).
```

Here are the reduction rules for matches on *nat*:

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

10.3 Structural Recursion

Inductive types are accompanied by a **recursion operator fix** that provides for the formulation of recursive functions. Here is a term that describes a recursive function $nat \to nat$ that doubles its argument.

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

You can think of a term $(fix\ f\,(x : s) : t := u)$ as a variant of the term $(fun\ x : s \Rightarrow u)$ where the additional local variable $f : s \to t$ names the recursive function being described. To name a recursive function outside its description, one can use an ordinary definition.

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

This definition can be written more conveniently as

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

Use the print command to see that *double* and *double'* are equated to the same term (up to alpha renaming). The keyword *Fixpoint* pays tribute to a theory where recursive functions appear as fixed points of non-recursive functions.

Recursive functions reduce as one would expect.

```
Eval cbv in double (S (S O)).
```
S (S (S (S O)))

```
Eval cbv in (fix f x : nat := match x with O => O | S x' => f x' end) (S O).
```
O

```
Eval cbv in fun n => (fix f x : nat := match x with O => O | S x' => f x' end) (S (S n)).
```
fun n ⇒ (fix f x : nat := match x with O ⇒ O | S x' ⇒ f x' end) n

Coq accepts a fix term only if it can determine that the recursion described by the fix term terminates. For instance, given an argument $S(S(S\,O))$, recursive applications are possible for the smaller values $S(S\,O)$, $S\,O$, and O. This form of recursion is known as **structural recursion**. Non-terminating recursions would destroy the consistency of the calculus of constructions (i.e., a proof of *False* may appear whose reduction does not terminate).

Here are recursive definitions of addition and multiplication.

```
Fixpoint add (x y : nat) : nat := match x with O => y | S x' => S (add x' y) end.
Fixpoint mul (x y : nat) : nat := match x with O => O | S x' => add (mul x' y) y end.
Eval cbv in mul (S (S O)) (S (S (S O))).
```
S (S (S (S (S (S O)))))

Both recursions decrease the first argument. If you print *add* and *mul*, you will see that fix can take more than one argument. Here is a subtraction function:

```
Fixpoint sub (x y : nat) : nat := match x, y with
| O, _ => O
| S x', S y' => sub x' y'
| S x', O => S x'
end.
Lemma sub_pred (x : nat) : pred x = sub x (S O).
destruct x ; simpl. reflexivity. destruct x ; simpl. reflexivity. reflexivity. Qed.
```

Here are the typing rule and the reduction rule for fix with one argument.

$$\frac{s : \mathsf{U}_n \qquad t : \mathsf{U}_n \qquad f : s \to t,\ x : s \Rightarrow u : t}{\mathsf{fix}\ f\,(x : s) : t\ :=\ u\ :\ s \to t}$$

$$(\mathsf{fix}\ f\ x := s)\ t \quad \rightsquigarrow \quad \left(s\,{}^{f}_{\mathsf{fix}\ f\ x := s}\right)^{x}_{t}$$

The reduction rule only applies if t starts with a constructor. This restriction is needed to obtain termination since the reduction rules (and hence the rule for fix) apply below abstractions and matches. The reduction of matches and fixes is known as **iota reduction**.

Exercise 10.3.1 Write the definition of *sub* in Coq's kernel syntax. Use the print command to check your answer.

Exercise 10.3.2 Define functions that yield powers m^n and factorials $n!$.

Exercise 10.3.3 Give a term that is accepted by Coq whose reduction would not terminate if the application constraint of the reduction rule for fix was dropped.

10.4 Inductive Proofs

An inductive proof is a recursive proof. Since *nat* has infinitely many elements, a proof of a universally quantified claim $\forall x : nat.\, s$ may require recursion. A typical example of a proposition that requires an inductive proof is the **induction principle for** *nat*:

$$\forall p : nat \to Prop.\ \ p\,O \to (\forall x : nat.\ p\,x \to p\,(S\,x)) \to \forall x : nat.\ p\,x$$

It states that we can obtain a proof of $\forall x : nat.\ p\,x$ from a proof of $p\,O$ (the **base case**) and a proof of $\forall x : nat.\ p\,x \to p\,(S\,x)$ (the **induction step**). Here is a proof of the induction principle:

```
Check
fun (p : nat -> Prop) (base : p O) (step : forall x, p x -> p (S x)) =>
    fix IHx (x : nat) : p x :=
    match x as n return p n with O => base | S x' => step x' (IHx x') end.
```

```
forall p : nat → Prop,  p O → (forall x : nat, p x → p (S x)) → forall x : nat, p x
```

Make sure that you understand the proof. The local variable *IHx* represents the recursive function and takes the rôle of the **inductive hypothesis**. Its type is $\forall x : nat.\ p\,x$.

There is a special tactic *fix* that synthesizes fix terms. Here is a script that synthesizes the proof of the induction principle.

```
Lemma nat_induction (p : nat -> Prop) :
p O -> (forall x, p x -> p (S x)) -> forall x, p x.
intros p base step. fix IHx 1. destruct x.
apply base.
apply step. apply IHx. Qed.
```

Step through the script to see what happens. The command *fix IHx 1* starts the synthesis of the fix term. It adds the assumption $IHx : \forall x : nat.\ p\,x$. It is possible to replace the command *fix IHx 1* with the command *refine* (*fix IHx* $(x : nat) := \_$). Check it out.

When we construct a recursive proof, Coq delays the termination check until the final Qed. If the termination check fails, Coq prints an error message and leaves the name of the lemma undefined.

After you have applied the inductive hypothesis (i.e., synthesized a recursive application), you may use the command *Guarded* to check whether the termination condition is satisfied. The command *Guarded* and the fact that a fix can have more than one argument seem to be the reason that one has to specify a number with the fix tactic. The number says which argument of the fix must be decreased by recursive applications.

Now that we synthesize matches and fixes, it is sometimes helpful to see the partial proof term synthesized so far. You can display this partial term by entering the command *Show Proof*.

It's time for more examples. Here is an inductive proof of a straightforward lemma.

```
Lemma S_irreflexive (x : nat) : x <> S x.
fix IHx 1. destruct x.
discriminate.
injection. apply IHx. Qed.
```

Look at the proof term and make sure that you understand how it is synthesized by the script. It is also possible to prove the lemma with the lemma *nat_induction*.

```
Lemma S_irreflexive' (x : nat) : x <> S x.
apply nat_induction.
discriminate.
intros x IHx. injection. apply IHx. Qed.
```

This proof is more high-level than the proof with *fix* in that we don't have to worry about termination. It turns out that the induction principle *nat_induction* suffices for most proofs that recurse over *nat*. Note that the induction principle relates to inductive proofs in the same way primitive recursion relates to recursive functions. Both functions provide high-level recursion schemes that ensure termination. The induction principle for natural numbers was formulated by Peano around 1900 as one of his axioms for the natural numbers.

Coq provides a tactic *induction* that knows about inductive proofs:

```
Lemma S_irreflexive'' (x : nat) : x <> S x.
induction x.
discriminate.
injection. apply IHx. Qed.
```

The tactic induction works for all inductive types and relies on induction principles automatically generated by Coq. For *nat* and *bool* Coq's induction principles look as follows:

```
Check nat_ind.
forall p : nat → Prop,  p O → (forall n, p n → p (S n)) → forall n, p n
Check bool_ind.
forall p : bool → Prop,  p true → p false → forall b, p b
```

Of course, Coq generates proofs for the induction principles it uses. Use the print command to look behind the curtain.

Here is a further example of an inductive proof.

```
Lemma sub_add (x y z : nat) : sub x (add y z) = sub (sub x y) z.
induction x ; intros y z. reflexivity.
destruct y. reflexivity. simpl. apply IHx. Qed.
```

Exercise 10.4.1 Prove $\forall x : nat.\ S(Sx) \neq Sx$. Give four proofs: An inductive proof with *fix*, an inductive proof with *nat_induction*, an inductive proof with the induction tactic, and a non-inductive proof with *S_injective* and *S_irreflexive*.

Exercise 10.4.2 (Zigzag Induction) Prove the following lemma.

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

Exercise 10.4.3 Here is a function that tests whether a number is even.

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

a) Prove $\forall x.\ even\ x = even(S(S\ x))$.

b) Prove $\forall x.\ even\ x = negb(even(S\ x))$. The proof is interesting since it is easy with *fix* and hard with the induction principle (a lemma is required). The reason is that *fix* admits recursions that take the predecessor more than once.

c) Write a function that computes the same results as *even* but uses *prim_rec* rather that fix and match. Prove the correctness of your function.

Exercise 10.4.4 The recursion operator fix is available for all inductive types. Here is an extreme example.

```
Inductive crazy : Prop :=
| Crazy : crazy -> crazy.
Lemma crazy_False : crazy -> False.
fix IH 1. intros [c']. apply (IH c'). Qed.
```

Explain why the lemma does not suffice to construct a proof of *False* without assumptions.

10.5 Basic Laws for Addition and Multiplication

We prove that the addition function *add* defined in §10.3 is commutative. This requires two lemmas.

```
Lemma add_S (x y : nat) : add x (S y) = S (add x y).
induction x ; intros y ; simpl.
reflexivity.
rewrite IHx. reflexivity. Qed.
```

```
Lemma add_O (x : nat) : add x O = x.
induction x ; simpl.
reflexivity.
rewrite IHx. reflexivity. Qed.
```

```
Lemma add_com (x y : nat) : add x y = add y x.
induction x ; intros y ; simpl.
rewrite add_O. reflexivity.
rewrite IHx. rewrite add_S.  reflexivity. Qed.
```

Exercise 10.5.1 Prove the following lemmas.

```
Lemma add_asso (x y z: nat) : add (add x y) z = add x (add y z).
```

```
Lemma mul_O  (x : nat) : mul x O = O.
```

```
Lemma mul_S (x y : nat) : mul x (S y) = add (mul x y) x.
```

```
Lemma mul_com (x y : nat) : mul x y = mul y x.
```

```
Lemma mul_dist (x y z: nat) : mul (add x y) z = add (mul x z) (mul y z).
```

```
Lemma mul_asso (x y z: nat) : mul (mul x y) z = mul x (mul y z).
```

Exercise 10.5.2 Prove $injective\,(fun\,x \Rightarrow add\,x\,x)$.

Exercise 10.5.3 Prove *injective add*. Hint: Use *add_O*.

Exercise 10.5.4 Try to prove $\forall x.\ add\ (add\ x\ x)\ x = add\ x (add\ x\ x)$. A proof with *add_asso* is straightforward but a direct proof by induction seems impossible.

10.6 Generalized Induction

Suppose you want to prove that a predicate p holds for all x. Then it suffices to show px for an arbitrary x under the assumption that p holds for all values that are smaller than x. We will formulate this generalized induction principle in Coq and prove its correctness. We start with the definition of the canonical order of the natural numbers.

```
Fixpoint less (x y : nat) : Prop := match x, y with
| O, S _ => True
| S x', S y' => less x' y'
| _, _ => False
end.

Notation "s < t" := (less s t).
```

Next we define the relaxation provided by the general induction principle.

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

The general induction principle can now be formulated as follows:

```
forall X p f, relax X p f -> forall x, p x.
```

We need a few facts about the order *less*.

```
Lemma lessO n  : ~ n < O.
destruct n ; tauto. Qed.

Lemma lessS n : n < S n.
induction n ; simpl ; tauto. Qed.

Lemma lessStrans {k m n} : k < m -> m < S n -> k < n.
fix IH 1. intros k m n ; destruct m ; destruct n ; simpl.
intros A. destruct (lessO _ A).
intros A. destruct (lessO _ A).
intros _ A. destruct (lessO _ A).
destruct k ; simpl. tauto.
apply IH. Qed.
```

We reformulate the principle it so that induction on *nat* becomes applicable. This leads to the following lemma.

```
Lemma size_induction' (X : Type) (p : X -> Prop) (f : X -> nat) :
relax X p f -> forall n x, f x < n  -> p x.
intros X p f R. induction n ; intros x A.
destruct (lessO _ A).
apply R. intros y B. apply IHn. apply (lessStrans B A). Qed.
```

The formulation of this lemma is the key insight of the proof of the generalized induction principle. The rest is straightforward.

```
Theorem size_induction (X : Type) (p : X -> Prop) (f : X -> nat) :
relax X p f -> forall x, p x.
intros X p f R x.  apply (size_induction' X p f R (S (f x)) x). apply lessS. Qed.
```

We obtain the principle of complete induction as a corollary.

```
Corollary complete_induction (p: nat->Prop) :
(forall n, (forall m, m < n -> p m)  -> p n) -> forall n, p n.
intros p. apply (size_induction nat p (fun n => n)). Qed.
```

We use the occasion to establish an important theorem about the canonical order of nat. It is remarkable that the theorem holds without assuming excluded middle. This is the case since *nat* is an inductively defined type.

```
Theorem trichotomy (x y : nat) :  x < y \/ x=y \/ y < x.
induction x ; destruct y ; simpl ; try tauto.
destruct (IHx y) as [H|[[]|H]] ; tauto. Qed.
```

The derived tactic *try tauto* leaves the goal unchanged and succeeds in case *tauto* fails. This way we can solve 3 of 4 subgoals with *tauto* and prove the remaining subgoal by hand. Also note that the pattern [] in the nested pattern eliminates the equation $x = y$.

10.7 Primitive Recursion

The following function formulates a recursion scheme known as **primitive recursion**.

```
Definition prim_rec (X : Type) (base : X) (step : nat -> X -> X) : nat -> X
:= fix f n  := match n with O => base | S n' => step n' (f n') end.
```

With *prim_rec* we can define many recursive functions without using recursion operator fix. This has the advantage that we need not worry about termination. The following equations caption the essence of primitive recursion.

$$prim\_rec\ X\ x\ f\ O = x$$
$$prim\_rec\ X\ x\ f\ (S\,n) = f\ n\ (prim\_rec\ X\ x\ f\ n)$$

Here are definitions of functions that use primitive recursion to compute the results of double and add.

```
Definition double'' : nat -> nat := prim_rec nat O (fun _ x => S (S x)).
Definition add' (x y : nat) : nat := prim_rec nat y (fun _ => S) x.
```

Here is another function that adds two numbers with primitive recursion.

```
Definition add'' : nat -> nat -> nat := prim_rec (nat -> nat) (fun y => y) (fun x' f y => S (f y)).
```

A correctness proof is straightforward.

```
Lemma add_add'' (x y : nat) : add x y = add'' x y.
induction x ; intros y ; simpl ; try tauto. rewrite IHx. reflexivity. Qed.
```

Exercise 10.7.1 Write functions that compute differences, products, powers, and factorials with primitive recursion. Prove the correctness of your functions.

Exercise 10.7.2 Express the predecessor function with *prim_rec*. Do not use *match*. Prove the correctness of your function.

Exercise 10.7.3 Below is a subtraction function that uses neither match nor fix. Prove the correctness of this function.

```
Definition sub' : nat -> nat -> nat :=
prim_rec (nat -> nat) (fun _ => O)
(fun x' f => prim_rec nat (S x') (fun y' _ => f y')).
Lemma sub'_correct (x y : nat) : sub' x y = sub x y.
```

10.8 Ackermann's Function

Recursive functions appear already in Dedekind's 1888 paper "Was sind und was sollen Zahlen?". The following predicate specifies a function designed by Ackermann in 1928.

```
Definition ackermann (f : nat -> nat -> nat) : Prop := forall m n,
f O n = S n   /\
f (S m) O = f m (S O)   /\
f (S m) (S n) = f m (f (S m) n).
```

The specification of Ackermann's function poses two questions:

1. Can we define with structural recursion a function that satisfies the predicate *ackermann*?
2. Can we prove that there is at most one function that satisfies the predicate *ackermann*?

We will answer both questions positively. In this section we address the existence question. The uniqueness question will be settle with the inductive proof technique introduced in a later section.

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 application of the equations terminates. The equations terminate since either the first argument is decreased, or the first argument stays the same and the second argument is decreased.

Existence

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 decreased by every recursive application. The problem can be resolved by formulating Ackermann's function with two nested recursions.

```
Definition ack : nat -> nat -> nat := fix f m := match m with
| O => S
| S m' => fix g n := match n with
            | O => f m' (S O)
            | S n' => f m' (g n')
end end.
```

Note that *ack* is defined as a recursive function that returns a recursive function. Check that each of the two recursions decreases its single argument. If you have not seen a transformation like this before, this may look like magic to you. Here is a proof that *ack* satisfies its specification. It is straightforward since each of the three equations follows by reduction.

```
Lemma ack_correctness : ackermann ack.
intros m n. split. reflexivity. split ; reflexivity. Qed.
```

Here is a formulation of Ackermann's function that uses primitive recursion.

```
Definition ack' := prim_rec (nat -> nat) S
(fun _ f => prim_rec nat (f (S O)) (fun _ => f)).
```

Correctness once more follows by reduction.

```
Lemma ack'_correctness : ackermann ack'.
intros m n. split. reflexivity. split ; reflexivity. Qed.
```

Ackermann proved that his function cannot be obtained with primitive recursion at *nat* (i.e., *prim_rec nat*). In fact, our formulation *ack′* uses primitive recursion both at $nat \to nat$ (outer recursion) and at *nat* (inner recursion). One says that Ackermann's function cannot be expressed with first-order primitive recursion.

Canonicity

We now prove that there is at most one function that satisfies our specification of Ackermann's function. The proof follows our definition *ack* of Ackermann's function in that there is an outer induction for the first argument and an inner induction for the second argument. The lemmas *P1* and *P2* are from §9.4.[1]

[1] Recall that we have switched to Coq's predefined logical operations. Thus our old lemmas must be reproven in the new setting.

```
Lemma ackermann_canonical (f g : nat -> nat -> nat) :
ackermann f -> ackermann g -> forall m n, f m n = g m n.
intros f g af ag. induction m.
intros n. rewrite (P1 af O n). rewrite (P1 ag O n). reflexivity.
induction n.
  rewrite (P1 (P2 af) m O). rewrite (P1 (P2 ag) m O). apply IHm.
  rewrite (P2 (P2 af) m n). rewrite (P2 (P2 ag) m n). rewrite IHn. apply IHm. Qed.
```

10.9 Reflection

One can define a function *nat_reflect* that subsumes *nat_match*, *nat_induction*, and *prim_rec*. With *nat_reflect* one can write all matches and many recursive functions and inductive proofs without using match and fix. Since fix is not used, there is no need to think about termination. We define *nat_reflect* with a script.

```
Definition nat_reflect : forall f : nat -> Type, f O -> (forall x, f x -> f (S x)) -> forall x, f x.
intros f base step. fix IHx 1. destruct x. apply base. apply step. apply IHx. Defined.
```

This is the first time we synthesize a term with a script whose type is not a proposition. Since we want to compute with the function *nat_reflect*, it is essential that we finish the definition with *Defined* rather than *Qed*. The following term is a proof of the induction principle for *nat*.

```
Check (fun p : nat -> Prop => nat_reflect p).
forall p : nat → Prop,  p O → (forall x : nat, p x → p (S x)) → forall x : nat, p x
```

And the next term gives us primitive recursion:

```
Check (fun (X : Type) => nat_reflect (fun _ => X)).
forall X : Type,  X → (nat → X → X) → nat → X
```

Here is a proof of this statement.

```
Lemma nat_rec_prim_rec (X : Type) x f n :
prim_rec X x f n = nat_reflect (fun _ => X) x f n.
intros X x f. induction n ; simpl. reflexivity. rewrite IHn. reflexivity. Qed.
```

Exercise 10.9.1 Write a function *nat_match′* with *nat_reflect* that has the same type as *nat_match* and computes the same result. Prove the correctness of your function.

10.10 Impredicative Definitions

It turns out that recursion can be simulated to some extend with the technique of **impredicative definition**. We demonstrate this technique with the recursive function *double* defined in §10.3. We define a predicate *doublep* not using recursion (i.e., *fix*) and prove $\forall x\, y.\ doublep\, x\, y \leftrightarrow double\, x = y$.

```
Definition doublep (x y : nat) : Prop := forall r : nat -> nat -> Prop,
r O O ->
(forall x y, r x y -> r (S x) (S (S y))) ->
r x y.
```

Speaking mathematically, *doublep* is defined as the least relation that satisfies the two equations that yield the recursive definition of the function *double*.

$$\begin{aligned} double\, O &= O \\ double\, (S\, x) &= S(S(double\, x))) \end{aligned}$$

Here is the correctness proof.

```
Lemma doublep_correct (x y : nat) : doublep x y <-> double x = y.
intros x y. split.
intros A. apply (A (fun x y => double x = y)).
  reflexivity.
  intros [|m] n [] ; reflexivity.
revert y. induction x ; simpl ; intros y [] r A B.
  apply A.
  apply B. apply IHx. reflexivity. apply A. apply B.
Qed.
```

Exercise 10.10.1 Here is an impredicative definition of the order on *nat*.

```
Definition less_imp (x y : nat) : Prop :=
forall p : nat -> nat -> Prop,
(forall y, p O (S y)) ->
(forall x y, p x y -> p (S x) (S y)) ->
p x y.
```

Prove $\forall x\, y.\ less\_imp\, x\, y \leftrightarrow x < y$.

10.11 Peano Axioms

At the end of the 19th century mathematicians were able to explain the numbers starting from first principles know as **Peano axioms**. The two main players in this development were Dedekind and Peano. We can formulate the Peano axioms as follows.

```
Section Peano_Axioms.
Variable nat : Type.
Variable O : nat.
Variable S : nat -> nat.
Variable S_neq_O : forall x, S x <> O.
Variable S_injective : injective S.
Variable induction : forall p : nat -> Prop, p O -> (forall x, p x -> p (S x)) -> forall x, p x.
End Peano_Axioms.
```

Starting from the Peano axioms one can obtain addition and multiplication using impredicative definitions and Skolem functions. One can then prove all basic properties of the natural numbers and continue with the construction of the real and complex numbers. A careful mathematical construction of the numbers starting from the Peano axioms is carried out in Edmund Landau's famous book "Grundlagen der Analysis" from 1930.

Our inductive definition of *nat* establishes a model of the Peano axioms. Thus there is no need to assume the Peano Axioms in Coq. We can say that Coq obtains the natural numbers from more general principles than the Peano axioms.

10.12 Finiteness

We want to define a predicate on types that holds on a type if the type has only finitely many elements. We observe that a type is infinite if and only if it is inhabited and there is a total and loop-free binary relation on the type. We use the following definitions.

```
Definition total {X : Type} (r : X -> X -> Prop) : Prop := forall x, exists y, r x y.
Definition transitive {X : Type} (r : X -> X -> Prop) : Prop := forall x y z, r x y -> r y z -> r x z.
Definition irreflexive {X : Type} (r : X -> X -> Prop) : Prop := forall x, ~ r x x.
Definition finite (X : Type) : Prop :=
inhab X -> forall r : X -> X -> Prop, total r ->  transitive r -> exists x, r x x.
Definition infinite (X : Type) : Prop :=
inhab X /\ exists r : X -> X -> Prop, total r /\ transitive r /\ irreflexive r.
```

We work with transitive relations so that loop freeness coincides with irreflexivity. First we prove that a type cannot be both finite and infinite.

```
Lemma fin_inf (X : Type) : finite X -> infinite X -> False.
ntros X fin. intros [inh [r [tot [trans irr]]]].
destruct (fin inh r tot trans) as [x H].
revert H. apply irr. Qed.
```

Given a surjective function from X to Y, X is infinite if Y is infinite.

```
Lemma sur_inf (X Y : Type) (f : X -> Y) : surjective f -> infinite Y -> infinite X.
intros X Y f surj. intros [inh [r [tot [trans irr]]]]. split.
(* inhab X *)  destruct inh as [y _]. destruct (surj y) as [x _].  exists x. apply I.
(* exists p *)  pose (p := fun x x' => r (f x) (f x')). exists p. split.
(* total p *)  intros x. destruct (tot (f x)) as [y A]. destruct (surj y) as [x' B].
  exists x'. red. rewrite B. apply A. split.
(* transitive p *)  intros x x' x''. apply trans.
(* irreflexive p *)  intros x. apply irr. Qed.
```

Next we prove that *bool* is finite.

```
Lemma bool_finite : finite bool.
intros inh r tot trans.
destruct (tot true) as [x A]. destruct x.
exists true. apply A.
destruct (tot false) as [y B]. destruct y.
  exists true. revert A B. apply trans.
  exists false. apply B. Qed.
```

To show that *nat* is infinite, we use the order *less*, which is total, transitive, and irreflexive.

```
Lemma less_total : total less.
intros x. exists (S x). induction x ; simpl. apply I. apply IHx. Qed.
```

Note the use of the tactic *revert*, which generalizes a given assumption and then clears it. One can see *revert* as inverse to *intros*. With the combination of *intros* and *reverse* one can rearrange a claim so that one gets a sufficiently strong inductive hypothesis.

```
Lemma less_transitive : transitive less.
intros x. induction x ; destruct y; destruct z ; simpl ; try tauto.
intros []. apply IHx. Qed.

Lemma less_irreflexive : irreflexive less.
intros x. induction x ; simpl ; tauto. Qed.

Lemma nat_infinite : infinite nat.
split. exists O. apply I. exists less. split.
apply less_total. split.
apply less_transitive.
apply less_irreflexive. Qed.
```

Exercise 10.12.1 Prove *finite False*.

Exercise 10.12.2 Prove $FE \to$ *finite* (*False* $\to$ *Prop*).

Exercise 10.12.3 Prove $PCA \to$ *finite Prop*.

Exercise 10.12.4 Prove $\forall X : Type.\ (\forall x\ y : X.\ x = y) \to$ *finite* X.

10.13 Transitive Closure with Nat

There is an intuitively appealing definition of the transitive closure operation that exploits the inductive structure of the natural numbers. We formalize this definition as follows:

```
Fixpoint iter {X : Type} (r : X -> X -> Prop) (n : nat) (x y : X) : Prop
:= match n with O => r x y | S n' => exists x', r x x' /\ iter r n' x' y end.

Definition ntc {X : Type} (r : X -> X -> Prop) (x x' : X) : Prop
:= exists n, iter r n x x'.
```

We prove that *ntc* yields a transitive relation.

```
Lemma ntc_left_transitive  {X : Type} {r : X -> X -> Prop} {x y z : X} :
r x y -> ntc r y z -> ntc r x z.
intros X r x y z A. intros [n B]. exists (S n). exists y. tauto. Qed.

Lemma ntc_transitive {X : Type} (r : X -> X -> Prop) : transitive (ntc r).
intros X r x y z. intros [n A]. revert x y z A.
induction n ; intros x y z ; simpl.
apply ntc_left_transitive.
intros [x' [A B]] C. apply (ntc_left_transitive A). revert B C. apply IHn. Qed.
```

We now show that *ntc* satisfies the specification *tc* of transitive closure we introduced in §8.4.

```
Definition implies {X : Type} (p q : X -> X -> Prop) : Prop :=
forall x y, p x y -> q x y.

Definition strong {X : Type} (r p : X -> X -> Prop) : Prop :=
forall q : X -> X -> Prop,  transitive q -> implies r q -> implies p q.

Definition tc {X : Type} (r p : X -> X -> Prop) : Prop :=
transitive p /\ implies r p /\ strong r p.

Lemma tc_ntc (X : Type) (r : X -> X -> Prop): tc r (ntc r).
intros X r. split.
(* trans ntc *)  apply ntc_transitive. split.
(* implies *)  intros x y A. exists O. apply A.
(* strong *)  intros q tra imp x y. intros [n A]. revert x y A. induction n.
apply imp.
intros x y [x' [A B]]. fold @iter in B. apply (tra x x').
  apply imp. apply A.
  apply IHn. apply B. Qed.
```

Note the command *fold @iter in B*. It undoes an unnecessary delta reduction of *iter* and this way makes the proof more readable. The proof still goes through if this command is deleted.

10.14 Coq's Predefined Natural Numbers

Coq predefines the natural numbers as an inductive data type *nat* with the constructors O and S. There are notational conveniences, for instance $S(S(SO))$ can be written as 3. There is also a powerful tactic *omega* that can solve many arithmetic goals. The tactic must first be loaded. Here is an example that runs in an empty Coq shell.

```
Require Import Omega.
Goal forall k m n : nat,  k < m -> m+1 < n+2 -> k < n.
intros k m n. omega. Qed.
```

11 Models and First-Order Logic

In this chapter we will study models. Models give us a way to define truth independent of provability. We will distinguish between validity (truth in all models), satisfiability (truth in some model) and unsatisfiability (truth in no models). For simplicity, we will restrict our attention to a first-order language with a single binary relation (FOL). A model of such a language is simply a (directed) graph. For this reason, we begin by considering graphs and properties of graphs. We can use Coq to prove properties of graphs in general and to prove properties of particular graphs.

In spite of its weakness, FOL is rich enough that validity is already undecidable. We will give a natural deduction system for FOL that corresponds to validity. This natural deduction system demonstrates that while validity is not decidable, it is at least recursively enumerable (RE).

11.1 Graphs

A **graph** G is given as a pair (V, E) where V is a nonempty set of vertices and E is a binary relation on V (i.e., $E \subseteq V \times V$). We give a few example graphs in Figure 11.1 and give them the names G_1, $G_1^=$, G_2, $G_2^=$, G_3 and G_∞.

We consider several properties a graph (equivalently, a binary relation) may have. We can represent these properties in Coq and prove relationships between them. Later we will show that the properties can also be expressed in a weaker language called first-order logic (FOL).

We first consider three simple properties.

- **full:** Every vertex is related to every other vertex. $\forall xy.(x, y) \in E$
- **reflexive:** Every vertex is related to itself. $\forall x.(x, x) \in E$
- **irreflexive:** No vertex is related to itself. $\forall x.(x, x) \notin E$

Table 11.1 indicates which of the graphs in Figure 11.1 satisfy which of these properties.

Clearly, any full graph must be reflexive. We prove this simple fact in Coq. To represent a general graph in Coq, we open a section, assume a type V for vertices, and a binary predicate E for edges. Since we only consider graphs with a nonempty set of vertices, we assume V is inhabited. Furthermore, we assume

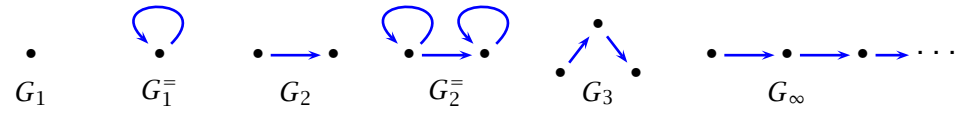


Figure 11.1: Six Example Graphs

	G_1	$G_1^=$	G_2	$G_2^=$	G_3	G_∞
full	No	Yes	No	No	No	No
reflexive	No	Yes	No	Yes	No	No
irreflexive	Yes	No	Yes	No	Yes	Yes

Table 11.1: Properties of the Six Example Graphs

XM to reflect that we are in a classical setting.

```
Section Graph.
  Variable V : Type.
  Variable E : V -> V -> Prop.
  Variable inh : inhab V.
  Variable xm : XM.
```

Next we include definitions corresponding to the three properties defined above.

```
  Definition full : Prop := forall x y:V, E x y.
  Definition reflexive : Prop := forall x:V, E x x.
  Definition irreflexive : Prop := forall x:V, ~E x x.
```

After the section is closed, these definitions will be generalized to account for the fact that the local variables are no longer available. Each of these definitions only depends on the variables V and E (but not *inh* or *xm*). Consequently, after the section is closed, the definitions will be as follows:

$$\begin{aligned} \mathit{full} &: \forall V : \mathit{Type}.(V \to V \to \mathit{Prop}) \to \mathit{Prop} \\ &:= \lambda V : \mathit{Type}.\lambda E : V \to V \to \mathit{Prop}.\forall x\,y : V.Exy \\ \mathit{reflexive} &: \forall V : \mathit{Type}.(V \to V \to \mathit{Prop}) \to \mathit{Prop} \\ &:= \lambda V : \mathit{Type}.\lambda E : V \to V \to \mathit{Prop}.\forall x : V.Exx \\ \mathit{irreflexive} &: \forall V : \mathit{Type}.(V \to V \to \mathit{Prop}) \to \mathit{Prop} \\ &:= \lambda V : \mathit{Type}.\lambda E : V \to V \to \mathit{Prop}.\forall x : V.\neg Exx \end{aligned}$$

For now, we continue to work within the section.

We now prove that a full graph must be reflexive.

```
Theorem full_ref : full -> reflexive.
intros a x.  apply a.
Qed.
```

Within the section, the proof is simply $\lambda a : \mathit{full}.\lambda x : V.axx$. After the section is closed, the theorem and proof will be

$$\forall V : \mathit{Type}.\forall E : V \to V \to \mathit{Prop}.\mathit{full}\, V\, E \to \mathit{reflexive}\, V\, E$$

and $\lambda V : \mathit{Type}.\forall E : V \to V \to \mathit{Prop}.\lambda a : \mathit{full}\, V\, E.\lambda x : V.axx$. This reflects the fact that our extra assumptions (inhabitation and excluded middle) were not used in the proof.

We next prove that if a graph is reflexive, then it is not irreflexive. In this case, we must use the fact that V is inhabited.

```
Theorem ref_not_irref : reflexive -> ~irreflexive.
intros a b.  destruct inh as [x _].  apply (b x (a x)).
Qed.
```

Since inhabitation was used, but excluded middle was not, the type of the theorem outside the scope of the section will be

$$\forall V : \mathit{Type}.\forall E : V \to V \to \mathit{Prop}.\mathit{inhab}\, V \to \mathit{reflexive}\, V\, E \to \neg \mathit{irreflexive}\, V\, E$$

Next we consider three more properties.

- **symmetric:** If there is an edge from x to y, then there is an edge from y to x. $\forall xy.(x,y) \in E \to (y,x) \in E$. A symmetric graph is essentially an undirected graph.
- **transitive:** For all nodes $x, y, z \in V$, if there are edges from x to y and from y to z, then there is an edge from x to z.

$$\forall xyz.(x,y) \in E \wedge (y,z) \in E \to (x,z) \in E$$

- **total:** For every node $x \in V$, there is a node $y \in V$ with an edge from x to y.

$$\forall x.\exists y.(x,y) \in E$$

In Coq, the definitions are given as follows:

```
Definition symmetric : Prop := forall x y:V, E x y -> E y x.
Definition transitive : Prop := forall x y z:V, E x y -> E y z -> E x z.
Definition total : Prop := forall x:V, exists y:V, E x y.
```

Again we use a table (Table 11.2) to indicate which of the graphs in Figure 11.1 satisfy which of these properties.

We prove that every symmetric, transitive, total graph is reflexive.

	G_1	$G_1^=$	G_2	$G_2^=$	G_3	G_∞
symmetric	Yes	Yes	No	No	No	No
transitive	Yes	Yes	Yes	Yes	No	No
total	No	Yes	No	Yes	No	Yes

Table 11.2: More Properties of the Six Example Graphs

```
Theorem stt_ref : symmetric -> transitive -> total -> reflexive.
intros a b c x.  destruct (c x) as [y Hxy].  apply (b _ y).  apply Hxy. apply a.  apply Hxy.
Qed.
```

Note that we used neither inhabitation nor XM.

Now, let us consider the following property:

$$\forall xy : V.(Exy \to Eyx) \vee (Eyx \to Exy)$$

This property clearly holds for any symmetric graph (e.g., G_1 and $G_1^=$). The property also holds in G_2. To check this, one would need to consider four cases where x, y range over the two vertices. For each case, either the left or right disjunct must be true. After careful consideration, the reader should be convinced that the property holds in all the graphs in Figure 11.1. In fact, the property holds for all graphs, as we now prove.

```
Theorem gd : forall x y, (E x y -> E y x) \/ (E y x -> E x y).
intros x y.  destruct (xm (E x y)) as [a|a].
right.  tauto.
left.  intros b.  destruct (a b).
Qed.
```

In this case, we did use our classical assumption XM. Consequently, when the section is closed, the theorem will have type

$$\forall V : Type.\forall E : V \to V \to Prop.XM \to \forall xy : V.(Exy \to Eyx) \vee (Eyx \to Exy).$$

Exercise 11.1.1 Prove the following:

```
Theorem total_total2 : total -> forall x:V, exists y:V, exists z:V, E x y /\ E y z.
```

Exercise 11.1.2 Consider the **diamond** property defined as follows:

```
Definition diamond : Prop := forall x y z, E x y -> E x z -> exists w, E y w /\ E z w.
```

Which of the graphs in Figure 11.1 satisfy this property?

Exercise 11.1.3 Let the following definitions be given.

```
Definition EqV (x y : V) : Prop := E x y /\ E y x.
Definition maximal (x : V) : Prop := forall y, E x y -> E y x.
```

Prove the following.

```
Theorem EqVRef : reflexive -> forall x, EqV x x.

Theorem EqVSym : forall x y, EqV x y -> EqV y x.

Theorem EqVTra : transitive -> forall x y z, EqV x y -> EqV y z -> EqV x z.

Theorem unf :  transitive -> reflexive -> diamond ->
   forall x y z, E x y -> E x z -> maximal y -> maximal z -> EqV y z.
```

We now close this section by closing the corresponding section in Coq.

```
End Graph.
```

11.2 Working with particular graphs in Coq

One can also use Coq to verify properties of particular graphs. As an example, we consider the graph G_2. We would like to reuse the names V and E in Coq for the vertices and edge relation for this particular graph. We can reuse these names with no problem because we have closed the Graph section in which V and E were local variables. On the other hand, if we define V and E for this particular graph globally, then we will not be able to redefine V and E for other examples in the same Coq file. For this reason, we will open a section for each example graph and define V and E locally within the section. To declare a definition that will not persist beyond a section, we use `Let` instead of `Definition`.

First, we must decide how to define the type of vertices of G_2. Let us call the vertices v_1 and v_2 according to the picture

We can use inductive types to obtain such a type of vertices.

```
Inductive V2 : Type :=
| v1 : V2
| v2 : V2.
```

We will reuse this type $V2$ for any graph with two vertices.[1] Now we open a section $G2$ and define the type V to be $V2$ and the edge relation E.

[1] We define $V2$ globally since it is not possible to make a local inductive definition in Coq.

```
Section G2.
 Let V := V2.
 Let E (x y : V) := match x,y with v1,v2 => True | _,_ => False end.
```

The reader should carefully consider the definition of E and why the definition corresponds to the graph G_2.

We can easily verify that G_2 is neither full nor reflexive. The idea in both cases is to assume the property, and then apply the property to counterexample vertices to conclude False.

```
 Theorem not_full_G2 : ~full V E.
 intros H.  apply (H v1 v1).
 Qed.

 Theorem not_reflexive_G2 : ~reflexive V E.
 intros H.  apply (H v1).
 Qed.
```

We can also easily verify that G_2 is irreflexive by case analysis.

```
 Theorem irreflexive_G2 : irreflexive V E.
 intros x H.  destruct x; apply H.
 Qed.
```

Exercise 11.2.1 Use Coq to verify that G_2 is transitive but is neither symmetric nor total.

Exercise 11.2.2 Consider the graph with two vertices $V = \{v_1, v_2\}$ and two edges $E = \{(v_1, v_2), (v_2, v_2)\}$.

Determine which of the properties (full, reflexive, irreflexive, symmetric, transitive, total) this graph has. Verify your answer with a proof in Coq.

11.3 First-Order Logic

Let $\mathcal{V}$ be a set of variables and let x, y range over these variables. **First-order formulas** are obtained with the grammar

$$s, t \ ::= \ \mathsf{E}xy \mid \bot \mid s \to t \mid \forall x.s$$

Let *Form* be the set of all formulas. We assume the same notational conventions as before. For example, $s \to t \to u$ means $s \to (t \to u)$, and $\forall xy.s \to t$ means $\forall x \forall y.(s \to t)$. We will also use $\neg s$ to denote $s \to \bot$.

In Coq we can obtain first-order formulas with an inductive definition. We use the natural numbers to represent variables.

```
Definition Var := nat.

Inductive Form : Type :=
| Edge : Var -> Var -> Form
| Fal : Form
| Imp : Form -> Form -> Form
| All : Var -> Form -> Form.
```

Let us introduce a right associative infix notation for implication.

```
Infix "-->" := Imp (at level 35, right associativity).
```

Sometimes it is helpful to have an equality function that maps to *bool* instead of *Prop*. We can define such a function for equality of variables recursively (using the fact that *Var* is *nat*).

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

We define a function FV : $Form \to \wp(\mathrm{Var})$ taking each formula s to the set $\mathrm{FV}(s)$ of its free variables by a simple recursion.

- $\mathrm{FV}(\mathsf{E}\,y\,z) = \{y, z\}$
- $\mathrm{FV}(\bot) = \emptyset$
- $\mathrm{FV}(s \to t) = \mathrm{FV}(s) \cup \mathrm{FV}(t)$
- $\mathrm{FV}(\forall y.s) = \mathrm{FV}(s) \setminus \{y\}$

In Coq, we can define FV as follows (using *bool* instead of *Prop*):

```
Fixpoint FV (s : Form) (x : Var) : bool :=
match s with
| (Edge y z) => if (EqVar x y) then true else (EqVar x z)
| Fal => false
| Imp s t => if (FV s x) then true else FV t x
| All y s => if (EqVar x y) then false else FV s x
end.
```

From now on we will simply call a first-order formula s a **formula**. A **sentence** is a formula with no free variables.

We similarly define a function $\mathrm{BV} : \mathit{Form} \to \wp(\mathrm{Var})$ taking each formula s to the set $\mathrm{BV}(s)$ of its bound variables by a simple recursion.

- $\mathrm{BV}(\mathsf{E}\,y\,z) = \emptyset$
- $\mathrm{BV}(\bot) = \emptyset$
- $\mathrm{BV}(s \to t) = \mathrm{BV}(s) \cup \mathrm{BV}(t)$
- $\mathrm{BV}(\forall y.s) = \mathrm{BV}(s) \cup \{y\}$

Again, we give the implementation in Coq.

```
Fixpoint BV (s : Form) (x : Var) : bool :=
match s with
| Imp s t => if (BV s x) then true else BV t x
| All y s => if (EqVar x y) then true else BV s x
| _ => false
end.
```

For $x, y, z \in \mathrm{Var}$, we write z^x_y to mean y when z is x and to mean z if z is not x. We can easily define a naive form of substitution s^x_y on formulas as follows:

- $(\mathsf{E}\,w\,z)^x_y := \mathsf{E}\; w^x_y\, z^x_y$
- $\bot^x_y := \bot$
- $(s \to t)^x_y := s^x_y \to t^x_y$
- $(\forall x.s)^x_y := \forall x.s$
- $(\forall z.s)^x_y := \forall z.(s^x_y)$ if $z \neq x$.

Note that this substitution allows capturing. For example, $(\forall y.\mathsf{E}\,x\,y)^x_y$ will be $\forall y.\mathsf{E}\,y\,y$. The intention is to only use s^x_y when we know $y \notin \mathrm{BV}s$. Here is an implementation in Coq.

```
Definition SubstVar (z x y:Var) : Var :=
if (EqVar z x) then y else z.

Fixpoint Subst (s : Form) (x y : Var) : Form :=
match s with
| Edge z w => Edge (SubstVar z x y) (SubstVar w x y)
| Fal => Fal
| Imp s t => Imp (Subst s x y) (Subst t x y)
| All z s => if (EqVar z x) then (All z s)
          else (All z (Subst s x y))
             (*** This assumes y is not z ***)
end.
```

Defining equivalence up to alpha renaming is a bit more challenging. We can define the binary relation $s \sim_\alpha t$ as a special case of a more general relation

$s \overset{f,g}{\sim} t$ where $f, g : \mathrm{Var} \to \mathrm{Var}$. The more general relation is recursively defined as follows:

1. $\mathsf{E}xy \overset{f,g}{\sim} \mathsf{E}wz$ if $fx = w$, $fy = z$, $x = gw$ and $y = gz$.
2. $\bot \overset{f,g}{\sim} \bot$.
3. $s_1 \to s_2 \overset{f,g}{\sim} t_1 \to t_2$ if $s_1 \overset{f,g}{\sim} t_1$ and $s_2 \overset{f,g}{\sim} t_2$.
4. $\forall x.s \overset{f,g}{\sim} \forall y.t$ if $s \overset{f^x_y, g^y_x}{\sim} t$

We use the notation f^x_y to denote the function such that $f^x_y x = y$ and $f^x_y z = fz$ if $z \neq x$. We define $s \sim_\alpha t$ to hold when $s \overset{\iota,\iota}{\sim} t$ where $\iota : \mathrm{Var} \to \mathrm{Var}$ is the identity function. We give the following definition in Coq using an auxiliary function to represent $s \overset{f,g}{\sim} t$.

```
Fixpoint AlphaEqAux (s t : Form) (f g : Var -> Var) :=
match s,t with
| Edge x y, Edge w z =>
  (match (EqVar (f x) w),(EqVar (f y) z),(EqVar x (g w)),(EqVar y (g z)) with
    true,true,true,true => true
  | _,_,_,_ => false end)
| Fal,Fal => true
| Imp s1 s2,Imp t1 t2 => if (AlphaEqAux s1 t1 f g) then (AlphaEqAux s2 t2 f g) else false
| All x s,All y t =>
    AlphaEqAux s t
      (fun z => if (EqVar z x) then y else (f z))
      (fun z => if (EqVar z y) then x else (g z))
| _,_ => false
end.

Definition AlphaEq (s t : Form) := AlphaEqAux s t (fun x => x) (fun x => x).
```

11.4 Satisfaction

Relative to a graph G, each sentence s is either true or false. Consider the sentence $\forall x.\mathsf{E}xx$. This is true if the graph is reflexive. The sentence is true in $G_1^=$ and $G_2^=$, but is false in the other graphs in Figure 11.1. Likewise, the sentence $\forall xy.\mathsf{E}xy$ is true exactly when the graph is full.

We have relied on the intuition of the reader above to check whether a sentence is true or false in a graph. Now we give precise definitions and prove some important results. We will define a binary relation $G \models s$ between graphs G and sentences s. We say **G is a model of s** or **s is true in G** precisely when $G \models s$ holds. In order to define $G \models s$, we will generalize to the case where s is a formula (i.e., s may have free variables). To account for free variables, we use assignments.

Let $G = (V, E)$ be a graph. An **assignment** into G is a mapping $\varphi : \mathcal{V} \to V$ from variables to vertices in G. Given an assignment φ into G, a variable $x \in \mathcal{V}$ and a vertex v, we use φ^x_v to denote the assignment such that $\varphi^x_v(x) = v$ and $\varphi^x_v(y) = \varphi y$ for $y \in \mathcal{V} \setminus \{x\}$. We define a 3-ary relation $G \vDash_\varphi s$ for graphs G, assignments φ into G, and formulas s. When $G \vDash_\varphi s$ holds, we say *G* **satisfies** *s* **with** *φ* or *φ* **is a satisfying assignment for** *s* **in** *G*. The relation $G \vDash_\varphi s$ is defined by recursion on formulas:

- $G \vDash_\varphi \mathsf{E}xy$ iff $(\varphi x, \varphi y) \in E$.
- $G \nvDash_\varphi \bot$
- $G \vDash_\varphi (s \to t)$ iff $G \vDash_\varphi s$ implies $G \vDash_\varphi t$.
- $G \vDash_\varphi \forall x.s$ iff $G \vDash_{\varphi^x_v} s$ for every $v \in V$.

We can also define the satisfaction relation in Coq. To do so, we first need to define assignments and a way to update an assignment φ to be a new assignment φ^x_v. In order to use the if-then-else notation at type *bool*, we make use of the *bool*-valued equality function *EqVar* on variables.

```
Definition Assignment (V:Type) : Type := Var -> V.

Definition Update {V:Type} (phi:Assignment V) (x:Var) (v:V) : Assignment V :=
 fun y => if (EqVar y x) then v else (phi y).

Fixpoint satisfies {V:Type} (E:V -> V -> Prop) (phi : Assignment V) (s : Form) : Prop :=
match s with
| Edge x y => E (phi x) (phi y)
| Fal => False
| Imp s t => (satisfies E phi s) -> (satisfies E phi t)
| All x s => forall v:V, satisfies E (Update phi x v) s
end.
```

We say a formula is **valid** if it is satisfied by every G and φ. We say a formula is **satisfiable** if it is satisfied by some G and φ. A formula is **unsatisfiable** if it is not satisfiable. Note that for any formula s, s is valid iff $\neg s$ is unsatisfiable. Likewise, $\neg s$ is valid iff s is unsatisfiable.

Exercise 11.4.1 Let G be a graph and φ be an assignment into G. Formulate and prove the following properties in Coq, assuming the classical assumption *XM*.

a) $G \vDash_\varphi \neg s$ iff $G \nvDash_\varphi s$.

b) $G \vDash_\varphi s$ iff $G \nvDash_\varphi \neg s$.

c) $G \vDash_\varphi s \to t$ iff either $G \nvDash_\varphi s$ or $G \vDash_\varphi t$

d) $G \nvDash_\varphi s \to t$ iff $G \vDash_\varphi s$ and $G \nvDash_\varphi t$.

e) $G \nvDash_\varphi \forall x.s$ iff there is some $v \in V$ such that $G \nvDash_{\varphi^x_v} s$.

$$\begin{array}{lcl} s \vee t & := & \neg s \to t \\ s \wedge t & := & \neg(s \to \neg t) \\ \exists x.s & := & \neg \forall x.\neg s \end{array}$$

Table 11.3: Notation for other logical operators

f) $G \vDash_\varphi \neg s \to t$ iff $G \vDash_\varphi s$ or $G \vDash_\varphi t$.

g) $G \vDash_\varphi \neg(s \to \neg t)$ iff $G \vDash_\varphi s$ and $G \vDash_\varphi t$.

h) $G \vDash_\varphi \neg \forall x.\neg s$ iff there is some $v \in V$ such that $G \vDash_{\varphi^x_v} s$.

Table 11.3 introduces notation for $\vee$, $\wedge$ and $\exists$. Exercise 11.4.1 justifies each of these. In Coq we can define these logical operators (and negation) as follows:

```
Definition Neg (s : Form) : Form := (s --> Fal).
Definition Or (s t : Form) := ((Neg s) --> t).
Definition And (s t : Form) := (Neg (s --> (Neg t))).
Definition Ex (x : Var) (s : Form) : Form := (Neg (All x (Neg s))).
```

Exercise 11.4.2 Prove the relationship between validity and unsatisfiability.

a) s is valid iff $\neg s$ is unsatisfiable.

b) $\neg s$ is valid iff s is unsatisfiable.

The following result is easily proven by recursion on formulas.

Proposition 11.4.3 Let G be a graph, s be a formula and φ, ψ be assignments such that $\varphi x = \psi x$ for every x free in s. Then $G \vDash_\varphi s$ iff $G \vDash_\psi s$.

In particular, Proposition 11.4.3 implies that $G \vDash_\varphi s$ does not depend on the assignment φ when s is a sentence. That is, for any graph G and sentence s, either $G \vDash_\varphi s$ for every assignment φ into G or for no assignment φ into G. We define $G \vDash s$ to hold if $G \vDash_\varphi s$ for every assignment φ into G and in this case we say *G* **is a model of** *s* or *s* **is true in** *G*.

Exercise 11.4.4 Find a sentence s such that a graph G is total iff $G \vDash s$.

Exercise 11.4.5 Let x and y be distinct variables. For each $n \geq 1$, give a formula s_n such that for every graph G and assignment φ, $G \vDash_\varphi s_n$ iff there is no path of length n in G from φx to φy.

Exercise 11.4.6 Which of the following formulas are valid? Which are satisfiable?

a) $((\mathsf{E}xy \to \mathsf{E}yx) \to \mathsf{E}xy) \to \mathsf{E}xy$

b) $((\forall x.\mathsf{E}xx) \to (\forall x.\mathsf{E}xx \to \bot) \to \bot$

Finally, we state two more results. These show that satisfaction respects alpha equivalence and substitution.

Proposition 11.4.7 If $s \sim_\alpha t$, then for every graph G and assignment φ, $G \vDash_\varphi s$ iff $G \vDash_\varphi t$.

Proposition 11.4.8 Let x and y be variables and s be a formula such that y is not bound in s. For any graph G and assignment φ, we have $G \vDash_\varphi s^x_y$ iff $G \vDash_{\varphi^x_{\varphi y}} s$.

11.5 Computational Properties

We can now ask a number of computational questions. We will also give the answers to these questions, but will spend some time elaborating in the rest of this chapter and the next.

1. Is there an algorithm that, given a formula s, determines whether or not s is valid? The answer is no. The set of valid formulas is not decidable.
2. Is there an algorithm that, given a formula s, determines whether or not s is satisfiable? No. This follows from the relationship between validity and satisfiability (see Exercise 11.4.2). Hence the set of satisfiable formulas is also not decidable.
3. Is the collection of valid formulas recursively enumerable (RE)? The answer to this question is yes. We will justify this by giving a proof system for which provability corresponds to validity.
4. Is the collection of unsatisfiable formulas RE? Yes. This also follows from the relationship between validity and satisfiability (see Exercise 11.4.2).
5. Is the collection of satisfiable formulas RE? No. Since the set of unsatisfiable formulas is RE, the set of satisfiable formulas is co-RE. If the set of satisfiable formulas were RE, then it would be decidable.
6. Is the collection of all satisfiable formulas in which no $\forall$ quantifiers appear decidable? Yes. This is essentially the problem of propositional satisfiability, which is NP-complete.
7. Is the collection of all satisfiable formulas in which $\forall$ quantifiers only appear in a prefix decidable? Yes. This is the Bernays-Schönfinkel fragment which we will prove decidable.

Recall that to show a set is RE, it is enough to give an algorithm which takes an input and terminates if the input is in the set and does not terminate if the

input is not in the set. Thus we can show a set is RE by giving a proof system that generates members of the set.

12 First-Order Natural Deduction

We now give a natural deduction proof system for first-order logic. Provability in this system will correspond to validity. Hence we can use this proof system to demonstrate that validity is RE.

12.1 Sequents and Sets of Assumptions

In natural deduction one proves formulas s relatives to a set of assumptions A. To make this technically precise, we introduce the notion of a sequent. A **sequent** is a pair $A \Rightarrow s$ where A is a finite set of formulas and s is a formula. We write $\Rightarrow s$ for $\emptyset \Rightarrow s$.

We can extend our notions of satisfaction and satisfiability to sets of formulas in an obvious way. Let A be a set of formulas. We say G **satisfies** A **with assignment** φ (and write $G \models_\varphi A$) if $G \models_\varphi t$ for every $t \in A$. We say A is **satisfiable** if there is some G and φ such that $G \models_\varphi A$, otherwise we say A is **unsatisfiable**.

We say a sequent $A \Rightarrow s$ is **valid** if $G \models_\varphi s$ for any G and φ for which $G \models_\varphi A$. In other words, A is valid if $A \cup \{\neg s\}$ is unsatisfiable. We can also extend the $\models$ notation to sequents and write $G \models_\varphi A \Rightarrow s$ (and say G **satisfies** $A \Rightarrow s$ **with assignment** φ) to mean $G \models_\varphi A$ implies $G \models_\varphi s$. In other words, $G \models_\varphi A \Rightarrow s$ means either $G \not\models_\varphi A$ or $G \models_\varphi s$. Note that $A \Rightarrow s$ is valid iff $G \models_\varphi A \Rightarrow s$ for all G and φ.

We can represent sets of assumptions by an inductive type in Coq (giving lists of formulas, not sets). Sequents can also be represented as an inductive type in Coq (essentially pairs).

```
Inductive Assumptions : Type :=
| Nil : Assumptions
| Add : Assumptions -> Form -> Assumptions.

Infix ",," := Add (at level 40, left associativity).

Inductive Sequent : Type :=
| Seq : Assumptions -> Form -> Sequent.

Infix "==>" := Seq (at level 42, left associativity).
```

Note that we have defined infix notation allowing us to read and write sequents in a more natural format. We also need a function to check for membership of a formula in a list. We define a simple function Eqform to check equality of formulas.

```
Fixpoint EqForm (s t : Form) : bool :=
match s,t with
| Edge x y,Edge w z => if (EqVar x w) then (EqVar y z) else false
| Fal,Fal => true
| Imp s1 s2,Imp t1 t2 => if (EqForm s1 t1) then (EqForm s2 t2) else false
| All x s,All y t => if (EqVar x y) then (EqForm s t) else false
| _,_ => false
end.
```

We can use this to define a function Assum to check if a formula is in a set of assumptions.

```
Fixpoint Assum (A : Assumptions) (s : Form) : bool :=
match A with
| Nil => false
| A,, t => if (EqForm s t) then true else Assum A s
end.
```

Recall from the last chapter that we have functions

```
FV : Form -> Var -> bool
BV : Form -> Var -> bool
AlphaEq : Form -> Form -> bool
Subst : Form -> Var -> Var -> Form
```

for free variables, bound variables, alpha equivalence, and substitution. We extend FV to apply to sets of formulas in the obvious way, defining $\mathrm{FV}(A)$ as $\bigcup_{t\in A}\mathrm{FV}(t)$. We can easily define a function to check if a variable occurs free in some assumption.

```
Fixpoint FVA (A : Assumptions) (x : Var) : bool :=
match A with
| Nil => false
| A,, t => if (FV t x) then true else FVA A x
end.
```

12.2 Natural Deduction

Figure 12.1 gives rules defining a first-order natural deduction (ND) proof system. We write $A \vdash s$ and say $A \Rightarrow s$ **is provable** when $A \Rightarrow s$ can be derived using these rules. We write $\vdash s$ and say s **is provable** if $\emptyset \Rightarrow s$ can be derived.

$$\textbf{Assu}\ \frac{}{A, s \Rightarrow s} \qquad \alpha\ \frac{A \Rightarrow s \qquad s \sim_\alpha t}{A \Rightarrow t} \qquad \rightarrow\textbf{I}\ \frac{A, s \Rightarrow t}{A \Rightarrow s \rightarrow t}$$

$$\rightarrow\textbf{E}\ \frac{A \Rightarrow s \rightarrow t \qquad A \Rightarrow s}{A \Rightarrow t} \qquad \forall\textbf{I}\ \frac{A \Rightarrow t}{A \Rightarrow \forall x.\, t}\ x \text{ fresh}$$

$$\forall\textbf{E}\ \frac{A \Rightarrow \forall x.\, t}{A \Rightarrow t^x_y}\ y \text{ not bound in } t \qquad \textbf{Contra}\ \frac{A, s \rightarrow \bot \Rightarrow \bot}{A \Rightarrow s}$$

x fresh means x is not free in any formula in A

Figure 12.1: First-order natural deduction proof rules

We can represent provability in this proof system in Coq by an inductive predicate Prov on sequents.

```
Inductive Prov : Sequent -> Prop :=
| Assu : forall A s, Assum A s = true -> Prov (A ==> s)
| Alpha : forall A s t, AlphaEq s t = true -> Prov (A ==> s) -> Prov (A ==> t)
| ImpI : forall A s t, Prov (A,, s ==> t) -> Prov (A ==> s --> t)
| ImpE : forall A s t, Prov (A  ==> s --> t) -> Prov (A ==> s) -> Prov (A ==> t)
| AllI : forall A x t, FVA A x = false -> Prov (A ==> t) -> Prov (A ==> All x t)
| AllE : forall A x t y, BV t y = false -> Prov (A ==> All x t) -> Prov (A ==> Subst t x y)
| Contra : forall A s, Prov (A,, s --> Fal ==> Fal) -> Prov (A ==> s).

Notation "A |- s" := (Prov (A ==> s)) (at level 42).
```

The correctness of this proof system can be factored into two properties: soundness and completeness.

- **Soundness**: If $A \vdash s$, then $A \Rightarrow s$ is valid.
- **Completeness**: If $A \Rightarrow s$ is valid, then $A \vdash s$.

Soundness can be proven by an easy recursion on the derivation of $A \vdash s$. We will show completeness later.

The idea of the soundness proof is that each rule preserves validity. For all the rules except the $\forall\mathbf{I}$ rule, we can show a stronger property. Namely, for any G and φ, if G satisfies every premise with assignment φ, then G satisfies the conclusion with the same assignment φ.

Exercise 12.2.1 Check the soundness of three of the rules by checking the following facts. Let G be a graph and φ be an assignment into G.

a) If $G \vDash_{\varphi} A, s \to \bot \Rightarrow \bot$, then $G \vDash_{\varphi} A \Rightarrow s$.

b) If $G \vDash_{\varphi} A, s \Rightarrow t$, then $G \vDash_{\varphi} A \Rightarrow s \to t$.

c) If $G \vDash_{\varphi} A \Rightarrow s \to t$ and $G \vDash_{\varphi} A \Rightarrow s$, then $G \vDash_{\varphi} A \Rightarrow t$.

We often say we want to prove $A \vdash s$ to mean we want to prove that $A \Rightarrow s$ is derivable. We may do this under the assumption that other sequents are derivable. Suppose, for example, a sequent $A, s \Rightarrow s \to t$ is derivable (i.e., $A \vdash s \to t$). By **Assu**, we know $A, s \vdash s$. By $\to$**E** we can conclude that $A, s \Rightarrow t$ is derivable (i.e., $A, s \vdash t$).

Let us repeat this argument. Assume $A, s \vdash s \to t$. We wish to prove $A, s \vdash t$. We can use $\to$**E** to reduce this to proving $A, s \vdash s \to t$ and $A, s \vdash s$. We know $A, s \vdash s \to t$ by assumption. We know $A, s \vdash s$ by the rule **Assu**.

A more complex example is given by an important property called **Weakening**. If A and A' are finite sets of formulas and $A \subseteq A'$, then it can be proven that if $A \Rightarrow s$ is derivable, then $A' \Rightarrow s$ is derivable. In other words, it can be proven that if $A \vdash s$, then $A' \vdash s$. Weakening can be proven by recursion on derivations, with some care taken to preserve freshness in the $\forall$**I** rule.

Proposition 12.2.2 [Weakening] Suppose $A \vdash s$ and A' is a finite set of formulas such that $A \subseteq A'$. Then $A' \vdash s$.

Instead of proving Weakening in Coq, we will simply take it as an axiom. This allows us to freely use Weakening to demonstrate provability of sequents.

```
Axiom Weakening : forall (A A' : Assumptions) (s : Form),
    A |- s
 -> (forall s, Assum A s = true -> Assum A' s = true)
 ->  A' |- s.
```

In order to demonstrate the mechanics of first-order natural deduction proofs, we will prove

$$\forall y.\mathsf{E}\,x\,y \vdash \forall x.\exists z.\mathsf{E}\,z\,x$$

where x, y and z are distinct variables. Recall that $\exists z.\mathsf{E}\,z\,x$ means $\neg\forall z.\neg\mathsf{E}\,z\,x$. Since x is free in the set of assumptions, we cannot apply the $\forall$**I** rule. Instead we first use the α rule to rename x to y. Hence it is enough to prove

$$\forall y.\mathsf{E}\,x\,y \vdash \forall y.\exists z.\mathsf{E}\,z\,y$$

Applying the $\forall$**I** rule, we reduce the goal to proving

$$\forall y.\mathsf{E}\,x\,y \vdash \exists z.\mathsf{E}\,z\,y$$

Since $\exists z.\mathsf{E}\,z\,y$ is $(\forall z.\neg\mathsf{E}\,z\,y) \to \bot$, we can apply the $\to$**I** rule to reduce the goal to

$$\forall y.\mathsf{E}\,x\,y, \forall z.\neg\mathsf{E}\,z\,y \vdash \bot$$

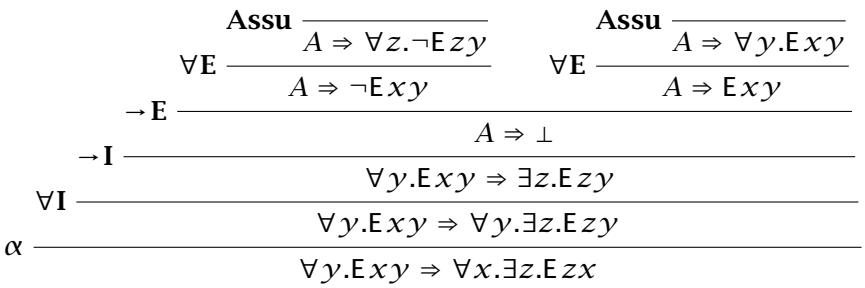


Figure 12.2: Example natural deduction derivation

Using **Assu** and $\forall$**E** (with $\forall z.\neg \mathsf{E}\, z y$ and x), we know

$$\forall y.\mathsf{E}\, x y, \forall z.\neg \mathsf{E}\, z y \Rightarrow \neg \mathsf{E}\, x y$$

Since $\neg \mathsf{E}\, x y$ is $\mathsf{E}\, x y \rightarrow \bot$, we can applying $\rightarrow$**E** to reduce the goal to proving

$$\forall y.\mathsf{E}\, x y, \forall z.\neg \mathsf{E}\, z y \vdash \mathsf{E}\, x y$$

This goal can be solved using **Assu** and $\forall$**E** (with $\forall y.\mathsf{E}\, x y$ and y). We show the full derivation in Figure 12.2. In this figure, A is the set $\{\forall y.\mathsf{E}\, x y, \forall z.\neg \mathsf{E}\, z y\}$.

12.3 Simulating ND in Coq

One can use Coq to simulate ND proof rules to justify $A \vdash s$. Figure 12.3 lists how to use Coq tactics to simulate ND proof rules. In addition to the basic rules, one can also use Weakening as follows: If the claim of the goal is $A' \vdash s$, then we can use

```
apply (Weakening A)
```

to reduce to two subgoals: (1) showing $A \vdash s$ and (2) showing every formula in A is also in A'.

As a demonstration of this, we simulate the proof in Figure 12.2. To be definite about variables, we take x, y and z to be 0, 1 and 2, respectively. (Recall that we have implemented first-order variables as natural numbers in Coq.)

```
Section Examples.
Let x : Var := O.
Let y : Var := (S O).
```

Assu: If the claim of the goal is $A \vdash s$ and $s \in A$, then use:

```
apply Assu.
```

This will leave a subgoal of proving $s \in A$. Often such a subgoal can be proven by computation using `reflexivity`.

α: If the claim of the goal is $A \vdash t$ and we want to use the α rule to reduce to a subgoal $A \vdash s$ where $s \sim_\alpha t$, then use

```
apply (Alpha _ s).
```

This leaves a subgoal of proving $s \sim_\alpha t$ (which may be provable by computation) and a subgoal $A \vdash t$.

$\to$I: If the claim of the goal is $A \vdash s \to t$, then to simulate $\to$**I** use

```
apply ImpI.
```

This leaves a subgoal $A, s \vdash t$.

$\to$E: If the claim of the goal is $A \vdash t$, then to simulate $\to$**E** use

```
apply (ImpE _ s).
```

This introduces two subgoals $A \vdash s \to t$ and $A \vdash s$. Note that we must explicitly give the formula s.

$\forall$I: If the claim of the goal is $A \vdash \forall x.t$, then to simulate $\forall$**I** use

```
apply AllI.
```

This leaves a subgoal of proving x is not free in A and a subgoal $A \vdash t$.

$\forall$E: Suppose the claim of the goal is $A \vdash s$ and we want to apply $\forall$**E**. Then s must be of the form t^x_y for some variables x and y and a formula t such that $y \notin \mathrm{BV}(t)$. To simulate $\forall$**E**, we must explicitly give x, t and y as follows:

```
apply (AllE _ x t y).
```

This leaves the subgoals of proving $y \notin \mathrm{BV}(t)$ and $A \vdash \forall x.t$. Coq will check that t^x_y computes to s. (If this is not the case, we may need to rewrite the current claim until it has this form.)

Contra: If the claim of the goal is $A \vdash s$, then to simulate **Contra** use

```
apply Contra.
```

This leaves the subgoal $A, \neg s \vdash \bot$.

Figure 12.3: Coq tactics for simulating natural deduction rules

```
Let z : Var := (S (S O)).

Example FOEx1 : Nil,, All y (Edge x y) |- All x (Ex z (Edge z x)).
apply (Alpha _ (All y (Ex z (Edge z y)))). reflexivity.
apply AllI. reflexivity.
apply ImpI.
apply (ImpE _ (Edge x y)).
apply (AllE _ z (Neg (Edge z y)) x). reflexivity.
apply Assu. reflexivity.
apply (AllE _ y (Edge x y) y). reflexivity.
apply Assu. reflexivity.
Qed.
End Examples.
```

Exercise 12.3.1 Prove the following:

```
Let x : Var := O.
Example refl_not_irrefl_FO : Nil,, All x (Edge x x),, All x (Neg (Edge x x)) |- Fal.

Example XM1_FO : Nil |- (Or (Edge x x) (Neg (Edge x x))).

Example XM2_FO : Nil |- (Or (Neg (Edge x x)) (Edge x x)).
```

Exercise 12.3.2 Suppose $A \vdash \bot$. Use weakening to argue that $A \vdash s$ for any formula s. (This is why our natural deduction system does not contain a $\bot$-elimination rule.)

12.4 Useful Facts about Provability

We now prove a number of facts about provability. These facts will be used to relate provability to refutability and will ultimately be used to prove completeness.

Proposition 12.4.1

1. If $s \in A$ and $\neg t \in A$ and $s \sim_\alpha t$, then $A \vdash \bot$.
2. If $s \to t \in A$ and $A, \neg s \vdash \bot$ and $A, t \vdash \bot$, then $A \vdash \bot$.
3. If $\neg(s \to t) \in A$ and $A, s, \neg t \vdash \bot$, then $A \vdash \bot$.
4. If $\forall x.s \in A$, $y \notin \mathrm{BV}(s)$ and $A, s^x_y \vdash \bot$, then $A \vdash \bot$.
5. If $\neg\forall x.s \in A$, y is not free in A, $y \notin \mathrm{BV}(s)$ and $A, \neg s^x_y \vdash \bot$, then $A \vdash \bot$.

Proof

1. Exercise.
2. Exercise.

3. Assume $\neg(s \to t) \in A$ and $A, s, \neg t \vdash \bot$. We need to prove $A \vdash \bot$. (That is, we must show that $A \Rightarrow \bot$ is provable.) By $\to$**E** and **Assu**, it is enough to prove $A \vdash s \to t$. By $\to$**I** it is enough to prove $A, s \vdash t$. We have this by **Contra** and the assumption that $A, s, \neg t \vdash \bot$.
4. Exercise.
5. This part is the most technically difficult because it involves α-renaming. Assume $\neg\forall x.s \in A$, y is not free in A, $y \notin \mathrm{BV}(s)$ and $A, \neg s^x_y \vdash \bot$. By $\to$**E** and **Assu**, it is enough to prove $A \vdash \forall x.s$. Under our assumptions, one can show $(\forall x.s) \sim_\alpha (\forall y.s^x_y)$. By α, it is enough to prove $A \vdash \forall y.s^x_y$. Since y is fresh, we can use $\forall$**I** to reduce our task to proving $A \vdash s^x_y$. By **Contra** and the assumption that $A, \neg s^x_y \vdash \bot$, we are done. ∎

The proof of the third part of Proposition 12.4.1 can be simulated in Coq as follows.

```
Lemma ImpN_Prov : forall A s t, Assum A (Neg (s --> t)) = true -> A,, s,, (Neg t) |- Fal
  -> A |- Fal.
intros A s t a b.
apply (ImpE _ (s --> t)).
apply Assu. apply a.
apply ImpI.
apply Contra.
apply b.
Qed.
```

The proof of the last part of Proposition 12.4.1 can also be simulated in Coq, assuming we have enough facts about $\sim_\alpha$.

Exercise 12.4.2 Prove the remaining parts of Proposition 12.4.1.

12.5 Conclusion

We have seen a natural deduction system for sequents. It is easy to see that the set of provable sequents $A \Rightarrow s$ can be enumerated using the proof system. Hence the set of provable formulas can be enumerated. By soundness and completeness, a formula is valid iff it is provable. Thus the set of valid formulas can be enumerated. (It also follows that the set of valid sequents can also be enumerated.)

13 First-Order Tableaux

Recall that a formula s is valid iff $\neg s$ is unsatisfiable. Likewise, sequent $A \Rightarrow s$ is valid iff the set $A, \neg s$ is unsatisfiable. Hence we can reduce validity to unsatisfiability. In the previous chapter we considered a natural deduction system for sequents such that provability of $A \Rightarrow s$ corresponds to validity of $A \Rightarrow s$. In this chapter we consider a tableau refutation system for sets of assumptions such that A is refutable iff A is unsatisfiable. Completeness of the tableau system will imply completeness of the natural deduction system.

13.1 Tableau System

A **branch** is a finite set of first-order formulas. That is, a branch is a set of assumptions. As with sets of assumptions, we use A, s as notation for $A \cup \{s\}$.

In general, a **tableau rule** (or **rule**) is a tuple $\langle A, A_1, \ldots, A_n \rangle$ of branches with $n \geq 0$ such that $A \subseteq A_i$ for each $i \in \{1, \ldots, n\}$. We can also write this tuple in the form

$$\frac{A}{A_1 \mid \cdots \mid A_n}$$

We refer to A as the **head** of this tableau rule and refer to each A_i as an **alternative** of the rule. If $n \geq 2$ we say the rule is **branching**.

We usually indicate a certain set of tableau rules by giving a rule schema. For example $\mathcal{T}_\to$ in Figure 13.1 is the set of rules $\langle A, A_1, A_2 \rangle$ where for some $s, t : o$ we have $s \to t \in A$, $A_1 = A \cup \{\neg s\}$ and $A_2 = A \cup \{t\}$. We say a rule **applies to** A if A is the head of the rule.

From an operational point of view, the tableau rule $\mathcal{T}_\to$ can be applied to A whenever $s \to t$ is in A. Applying $\mathcal{T}_\to$ in such a situation yields two branches $A, \neg s$ and A, t.

While schemas like $\mathcal{T}_\to$ and $\mathcal{T}_{\neg\to}$ are technically sets of rules, we will often refer to them simply as rules.

Our specific tableau system is defined by the rules in Figure 13.1.

We say a branch A is **closed** if one of the rules $\mathcal{T}_\bot$ or $\mathcal{T}_\neg$ applies to A. In other words, A is closed if either $\bot \in A$ or $\{s, \neg s\} \subseteq A$ for some s. If a branch is not closed, we say it is **open**.

We define the set of **refutable** branches inductively as follows:

$$\mathcal{T}_\bot\ \frac{\bot}{-} \qquad \mathcal{T}_\neg\ \frac{\neg s, s}{-} \qquad \mathcal{T}_\to\ \frac{s \to t}{\neg s \mid t} \qquad \mathcal{T}_{\neg\to}\ \frac{\neg(s \to t)}{s, \neg t} \qquad \mathcal{T}_\forall\ \frac{\forall x.s}{s^x_y}\ y \notin \mathrm{BV}s$$

$$\mathcal{T}_{\neg\forall}\ \frac{\neg\forall x.s}{\neg s^x_y}\ y \text{ fresh}, y \notin \mathrm{BV}s \qquad\qquad \mathcal{T}_\alpha\ \frac{s}{t}\ s \sim_\alpha t$$

y fresh means y is not free in the branch.

Figure 13.1: First-order tableau rules

- If $\langle A, A_1, \ldots, A_n\rangle$ is a rule and A_i is refutable for all $i \in \{1, \ldots, n\}$, then A is refutable. (In the special case when $n = 0$, this means that every closed branch is refutable.)

The tableau calculus is sound and complete:

- **Refutation Soundness:** Every refutable branch is unsatisfiable.
- **Completeness:** Every unsatisfiable branch is refutable.

We will prove both of these properties soon.

13.2 Relationship to Natural Deduction

In this section we relate the notion of (tableau) refutability to that of (natural deduction) provability.

Proposition 13.2.1 If A is refutable, then $A \vdash \bot$.

Proof This is easy to prove by induction on derivations. For each rule of the form $\langle A, A_1, \ldots, A_n\rangle$, the inductive hypothesis implies we know $A_1 \vdash \bot$, ..., $A_n \vdash \bot$. We must justify $A \vdash \bot$. For the rules $\mathcal{T}_\neg$, $\mathcal{T}_\to$, $\mathcal{T}_{\neg\to}$, $\mathcal{T}_\forall$ and $\mathcal{T}_{\neg\forall}$ we can use Proposition 12.4.1. For $\mathcal{T}_\bot$, if $\bot \in A$, then we know $A \vdash \bot$ by **Assu**. It only remains to consider the $\mathcal{T}_\alpha$ case. Assume $A, s, t \vdash \bot$. We must show $A, s \vdash \bot$. By $\to$**I** we know $A, s \vdash t \to \bot$. By **Assu** we know $A, s \vdash s$ and so $A, s \vdash t$ by α. By $\to$**E** we know $A, s \vdash \bot$ as desired. ∎

We can now infer refutation soundness from soundness of the natural deduction system.

Theorem 13.2.2 (Refutation Soundness) If A is refutable, then A is unsatisfiable.

Proof Suppose A is refutable. By Proposition 13.2.1 $A \vdash \bot$. By soundness of the natural deduction system, $A \Rightarrow \bot$ is valid. That is, A is unsatisfiable. ∎

Similarly, once we know completeness of the tableau system, then we will know the natural deduction system is complete.

Proposition 13.2.3 Assume the tableau system is complete. The natural deduction is also complete. That is, if $A \Rightarrow s$ is valid, then $A \vdash s$.

Proof Assume $A \Rightarrow s$ is valid. Then $A, \neg s$ is unsatisfiable. Completeness of the tableau system implies $A, \neg s$ is refutable. By Proposition 13.2.1 $A, \neg s \vdash \bot$. By **Contra** we have $A \vdash s$ as desired. ∎

13.3 Examples

We now consider a few example refutations. In each case we use a tableau refutation to show a formula s is valid by showing its negation $\neg s$ is unsatisfiable.

Example 13.3.1 For our first example, we will prove $\forall x.\bot \to \mathsf{E}xx$. We will refute the branch $\{\neg\forall x.\bot \to \mathsf{E}xx\}$. The branch can be refuted simply by applying the rules $\mathcal{T}_{\neg\forall}$ (with x), $\mathcal{T}_{\neg\to}$ and $\mathcal{T}_{\bot}$. This refutation can be shown in the following format:

$$\begin{array}{c} \neg\forall x.\bot \to \mathsf{E}xx \\ \neg(\bot \to \mathsf{E}xx) \\ \bot \\ \neg\mathsf{E}xx \end{array}$$

The first formula is the negation of the formula we wish to prove valid. The remaining formulas are added by applications of tableau rules. The final branch is closed by $\mathcal{T}_{\bot}$. □

Example 13.3.2 We now use tableau to refute the negation of the Drinker's Lemma (see 6.3): $\exists x.\forall y.\mathsf{E}xx \to \mathsf{E}yy$. Here we are using $\mathsf{E}xx$ in place of a unary predicate dx. Also, note that $\exists x.s$ as used here is notation for $\neg\forall x.\neg s$. We negate the formula and begin the refutation

$$\neg\neg\forall x.\neg\forall y.\mathsf{E}xx \to \mathsf{E}yy$$

How can we remove the double negation? Since $\neg\neg s$ is the same as $\neg(s \to \bot)$ we can apply $\mathcal{T}_{\neg\to}$ to obtain

$$\begin{array}{c} \neg\neg\forall x.\neg\forall y.\mathsf{E}xx \to \mathsf{E}yy \\ \forall x.\neg\forall y.\mathsf{E}xx \to \mathsf{E}yy \\ \neg\bot \end{array}$$

The formula $\neg\bot$ will not contribute to finding a refutation, so we take the liberty to omit writing it from now on. Applying $\mathcal{T}_\forall$ with the variable x we add

$$\neg\forall y.\mathsf{E}xx \to \mathsf{E}yy$$

to the branch. Applying $\mathcal{T}_{\neg\forall}$ with the variable y we add

$$\neg(\mathsf{E}xx \to \mathsf{E}yy)$$

to the branch. Applying $\mathcal{T}_{\neg\forall}$ with the variable y we add to the branch. Now we apply $\mathcal{T}_{\neg\to}$ to add $\mathsf{E}xx$ and $\neg\mathsf{E}yy$ to the branch. We now reconsider the formula

$$\forall x.\neg\forall y.\mathsf{E}xx \to \mathsf{E}yy$$

We wish to apply $\mathcal{T}_\forall$ with y. Unfortunately, y occurs bound in this formula. We resolve this difficulty by applying $\mathcal{T}_\alpha$ to add the alpha equivalent formula

$$\forall y.\neg\forall z.\mathsf{E}yy \to \mathsf{E}zz$$

to the branch. We now apply $\mathcal{T}_\forall$ to this formula with y, followed by $\mathcal{T}_{\neg\forall}$ with the variable z (assumed to be different from x and y) and finally $\mathcal{T}_{\neg\to}$. This yields the branch

$$\begin{array}{c}
\neg\neg\forall x.\neg\forall y.\mathsf{E}xx \to \mathsf{E}yy\\
\forall x.\neg\forall y.\mathsf{E}xx \to \mathsf{E}yy\\
\neg\forall y.\mathsf{E}xx \to \mathsf{E}yy\\
\neg(\mathsf{E}xx \to \mathsf{E}yy)\\
\mathsf{E}xx, \neg\mathsf{E}yy\\
\forall y.\neg\forall z.\mathsf{E}yy \to \mathsf{E}zz\\
\neg\forall z.\mathsf{E}yy \to \mathsf{E}zz\\
\neg(\mathsf{E}yy \to \mathsf{E}zz)\\
\mathsf{E}yy, \neg\mathsf{E}zz
\end{array}$$

Note that this branch is closed by $\mathcal{T}_\neg$ since it contains both $\mathsf{E}yy$ and $\neg\mathsf{E}yy$. □

Our first two examples have not made use of the only branching rule: $\mathcal{T}_\to$. Consequently, our final tableau refutation only had one branch. In general, the refutation will contain several branches. Branching shows up in our next example: Russell's Law.

Example 13.3.3 Recall that Russell's law states $\neg\exists x.\forall y.\mathsf{E}xy \leftrightarrow \neg\mathsf{E}yy$. Here $s \leftrightarrow t$ means $(s \to t) \wedge (t \to s)$ and $u \wedge v$ means $\neg(u \to \neg v)$. Consequently, $s \leftrightarrow t$ means $\neg((s \to t) \to \neg(t \to s))$. We begin by negating the law.

$$\neg\neg\exists x.\forall y.\mathsf{E}xy \leftrightarrow \neg\mathsf{E}yy$$

We apply $\mathcal{T}_{\neg\to}$ to get rid of the double negation, $\mathcal{T}_{\neg\forall}$ with x and $\mathcal{T}_{\neg\to}$ again. After applying these rules we have the following branch (omitting $\neg\bot$):

$$\begin{array}{c}\neg\neg\exists x.\forall y.\mathsf{E}xy \leftrightarrow \neg\mathsf{E}yy\\ \neg\forall x.\neg\forall y.\mathsf{E}xy \leftrightarrow \neg\mathsf{E}yy\\ \neg\neg\forall y.\mathsf{E}xy \leftrightarrow \neg\mathsf{E}yy\\ \forall y.\mathsf{E}xy \leftrightarrow \neg\mathsf{E}yy\end{array}$$

Next we apply $\mathcal{T}_{\forall}$ with x to add

$$\mathsf{E}xx \leftrightarrow \neg\mathsf{E}xx$$

to the branch. Since equivalence is simply notation, we can apply $\mathcal{T}_{\neg\to}$ to add $\mathsf{E}xx \to \neg\mathsf{E}xx$ and $\neg\neg(\neg\mathsf{E}xx \to \mathsf{E}xx)$ to the branch. We apply $\mathcal{T}_{\neg\to}$ to get rid of the double negation. At this point we have the following branch:

$$\begin{array}{c}\neg\neg\exists x.\forall y.\mathsf{E}xy \leftrightarrow \neg\mathsf{E}yy\\ \neg\forall x.\neg\forall y.\mathsf{E}xy \leftrightarrow \neg\mathsf{E}yy\\ \neg\neg\forall y.\mathsf{E}xy \leftrightarrow \neg\mathsf{E}yy\\ \forall y.\mathsf{E}xy \leftrightarrow \neg\mathsf{E}yy\\ \mathsf{E}xx \leftrightarrow \neg\mathsf{E}xx\\ \mathsf{E}xx \to \neg\mathsf{E}xx\\ \neg\neg(\neg\mathsf{E}xx \to \mathsf{E}xx)\\ \neg\mathsf{E}xx \to \mathsf{E}xx\end{array}$$

We proceed by applying $\mathcal{T}_{\to}$ to $\neg\mathsf{E}xx \to \mathsf{E}xx$. Thus we obtain two branches, one extended with $\neg\neg\mathsf{E}xx$ and the other extended with $\mathsf{E}xx$. We visualize the new state as a tree with two branches.

$$\begin{array}{c}\neg\neg\exists x.\forall y.\mathsf{E}xy \leftrightarrow \neg\mathsf{E}yy\\ \neg\forall x.\neg\forall y.\mathsf{E}xy \leftrightarrow \neg\mathsf{E}yy\\ \neg\neg\forall y.\mathsf{E}xy \leftrightarrow \neg\mathsf{E}yy\\ \forall y.\mathsf{E}xy \leftrightarrow \neg\mathsf{E}yy\\ \mathsf{E}xx \leftrightarrow \neg\mathsf{E}xx\\ \mathsf{E}xx \to \neg\mathsf{E}xx\\ \neg\neg(\neg\mathsf{E}xx \to \mathsf{E}xx)\\ \neg\mathsf{E}xx \to \mathsf{E}xx\\ \neg\neg\mathsf{E}xx \;\big|\; \mathsf{E}xx\end{array}$$

On each of these branches we can apply $\mathcal{T}_{\to}$ on $\mathsf{E}xx \to \neg\mathsf{E}xx$. The result is a tree with four branches, each extended with $\neg\mathsf{E}xx$. It is easy to see that all four

branches are closed via $\mathcal{T}_\neg$.

$$\neg\neg\exists x.\forall y.\mathsf{E}xy \leftrightarrow \neg\mathsf{E}yy$$
$$\neg\forall x.\neg\forall y.\mathsf{E}xy \leftrightarrow \neg\mathsf{E}yy$$
$$\neg\neg\forall y.\mathsf{E}xy \leftrightarrow \neg\mathsf{E}yy$$
$$\forall y.\mathsf{E}xy \leftrightarrow \neg\mathsf{E}yy$$
$$\mathsf{E}xx \leftrightarrow \neg\mathsf{E}xx$$
$$\mathsf{E}xx \rightarrow \neg\mathsf{E}xx$$
$$\neg\neg(\neg\mathsf{E}xx \rightarrow \mathsf{E}xx)$$
$$\neg\mathsf{E}xx \rightarrow \mathsf{E}xx$$

$$\begin{array}{ccc} \neg\neg\mathsf{E}xx & & \mathsf{E}xx \\ \neg\mathsf{E}xx \mid \neg\mathsf{E}xx & \mid & \neg\mathsf{E}xx \mid \neg\mathsf{E}xx \end{array}$$

13.4 Simulating Tableau in Coq

We now give a way to simulate this tableau refutation system in Coq. We assume we are in a section with an assumed type V and an edge relation E. We further assume V is inhabited and that excluded middle holds.

```
Section Tableau.
Variable V : Type.
Variable E : V -> V -> Prop.
Variable inh : inhab V.
Variable xm : XM.
```

We can embed first-order formulas into this context as follows:

- $\overline{\bot} :=$ `False`
- $\overline{\mathsf{E}xy} :=$ `E x y`
- $\overline{s \rightarrow t} := \overline{s} -> \overline{t}$
- $\overline{\forall x.s} :=$ `forall x,` $\overline{s}$

In other words, we use Coq's version of $\forall$, $\rightarrow$ and $\bot$. For convenience, we define notation for some other logical constants.

```
Definition existsF (p : V -> Prop) : Prop := ~forall x, ~p x.
Notation "'existsF' x , s" := (existsF (fun x => s)) (at level 42).
Definition andF (s t : Prop) : Prop := ~(s -> ~t).
Definition orF (s t : Prop) : Prop := (~s -> t).
Definition equivF (s t : Prop) : Prop := andF (s -> t) (t -> s).
```

We will now define a set of tactics to simulate our tableau system.

If our goal is to prove s by a tableau refutation, we always start by assuming $\neg s$. This can be justifying by a double negation lemma, which is proven using excluded middle.

```
Lemma DN (X : Prop) : (~X -> False) -> X.
intros X H. destruct (xm X) as [a|b].
apply a.
destruct (H b).
Qed.
```

We define a tactic `tab_start` to apply DN.

```
Ltac tab_start H := (apply DN ; intros H).
```

We will start all of our tableau-style proofs with

```
tab_start H1.
```

After using this tactic, claim of our goal is

```
False
```

under one assumption

H1: $\sim s$

This tactic is only used once in the proof. After this tactic is used, the claim of all our goals will remain `False`.

Next consider the closing rules $\mathcal{T}_\bot$ and $\mathcal{T}_\neg$. We can apply $\mathcal{T}_\bot$ if we have an assumption `False`. Since our claim is also `False`, we can complete our proof of the goal using the `exact` tactic. To be specific, if `H` is the label of the assumption `False`, `exact H` will complete the proof of the goal. We define a simple tactic `tab_bot` to do this.

```
Ltac tab_bot H := (exact H).
```

In a similar way, if $\mathcal{T}_\neg$ applies, then there must be an formula s such that s and $\neg s$ are both assumptions. Suppose `H` is the label of s and `H'` is the label of $\neg s$. `exact (H' H)` will complete the proof of the goal. We define a simple tactic `tab_conflict` to do this.

```
Ltac tab_conflict H H' := (exact (H' H)).
```

All the remaining tactics have the effect of extending our set of assumptions.

We can apply $\mathcal{T}_\rightarrow$ if $s \rightarrow t$ is an assumption, say with label `H`. After applying the rule, we will have two new subgoals, each with the claim `False` and each with one new assumption. In one subgoal, the new assumption will be $\neg s$. In the other subgoal, the new assumption will be t. We will use the same label `H'` for the new assumption in both subgoals. The rule $\mathcal{T}_\rightarrow$ is simulated by the tactic `tab_imp` applied to `H` and the new label `H'`. `tab_imp` is defined (using a lemma) as follows:

```
Lemma TImp {X Y:Prop} : (X -> Y) -> (~X -> False) -> (Y -> False) -> False.
intros X Y H R1 R2.
apply R1. intros x. apply R2. apply (H x).
Qed.

Ltac tab_imp H H' := (apply (TImp H) ; intros H').
```

We can apply $\mathcal{T}_{\neg\to}$ if $\neg(s \to t)$ is an assumption, say with label H. This will add two new assumptions, s and $\neg t$. We simulate this rule by `tab_imp'` applied to the label of the assumption $\neg(s \to t)$ and two new labels for the two new assumptions.

```
Lemma TImp' {X Y:Prop} : ~(X -> Y) -> (X -> ~Y -> False) -> False.
intros X Y H R.
apply H. intros x. destruct R.
apply x.
intros y. apply H. intros _. apply y.
Qed.

Ltac tab_imp' H H' H'' := (apply (TImp' H) ; intros H' H'').
```

As we have already seen in examples, we sometimes would like to apply $\mathcal{T}_{\neg\to}$ but ignore one of the new assumptions. This is particularly true if we apply the rule to a double negation and so one of the second new assumption is $\neg\bot$. We define two tactics that allow us to only label one of the new assumptions.

```
Ltac tab_imp1' H H' := (apply (TImp' H) ; intros H' _).
Ltac tab_imp2' H H' := (apply (TImp' H) ; intros _ H').
```

Since Coq considers alpha equivalent terms equivalent, we do not need to simulate the $\mathcal{T}_\alpha$ rule. For the same reason, we can drop the side condition that $y \notin \mathrm{BV}(s)$ in the $\mathcal{T}_\forall$ and $\mathcal{T}_{\neg\forall}$ rules. This will mean there are slight differences between the proof using the tableau system and its simulation in Coq. Make sure to notice where we must use $\mathcal{T}_\alpha$ in the refutation system, even though this is left implicit in Coq.

The $\mathcal{T}_\forall$ rule can be simulated with a tactic `tab_all` which must be given the label of an assumption of the form $\forall x.s$, a variable y of type V and a fresh label for the new assumption. This is defined using a lemma as follows:

```
Lemma TAll {p:V -> Prop} : (forall x:V, p x) -> forall y:V, (p y -> False) -> False.
intros p H y R.
apply R. apply H.
Qed.

Ltac tab_all H y H' := (apply (TAll H y) ; intros H').
```

There is one minor problem to work around. In the tableau system, we could use any y in the $\mathcal{T}_\forall$ rule (so long as $y \notin \mathrm{BV}(s)$). Coq will only allow us to use a y where $y : V$ is an assumption of our goal. Since V is assumed to be inhabited, we add a tactic to introduce the assumption $y : V$ whenever we wish to use y as an instantiation. In particular, we will need this tactic to simulate the Drinker's Lemma (Example 13.3.2).

```
Ltac tab_inh y := (destruct inh as [y _]).
```

Finally, the $\mathcal{T}_{\neg\forall}$ can be simulated by a tactic `tab_all'` defined as follows.

```
Lemma TAll' {p:V -> Prop} : ~(forall x:V, p x) -> (forall y:V, ~p y -> False) -> False.
intros p H R.
apply H. intros x. destruct (xm (p x)) as [a|a].
apply a.
destruct (R x a).
Qed.

Ltac tab_all' H y H' := (apply (TAll' H) ; intros y H').
```

A tableau refutation in Coq of a first-order formula is simply a proof that only uses the tactics defined above:

```
tab_start H.        (*** Only used once, at the start. ***)
tab_bot H.
tab_conflict H H'.
tab_imp H H'.
tab_imp' H H' H''.
tab_imp1' H H'.
tab_imp2' H H'.
tab_all H y H'.
tab_inh y.
tab_all' H y H'.
```

Here is a quick summary of how each tactic is used.

- `tab_start H.` - This is only used once, at the beginning of the proof. H must be a fresh name and will correspond to the negation of the formula we wish to prove.
- `tab_bot H.` - This presupposes $H : \bot$ is an assumption and finishes the goal.
- `tab_conflict H H'.` - This presupposes $H : s$ and $H' : \neg s$ are assumptions and finishes the goal.
- `tab_imp H H'.` - This presupposes $H : s \to t$ is an assumption and creates two new subgoals. The first subgoal has the additional assumptiom $H' : \neg s$ and the second subgoal has the additional assumption $H' : t$. H' must be a fresh name.

- `tab_imp' H H' H''.` - This presupposes $H : \neg(s \to t)$ is an assumption and adds the two new assumptions $H' : s$ and $H'' : \neg t$. H' and H'' must be fresh.
- `tab_imp1' H H'.` - This presupposes $H : \neg(s \to t)$ is an assumption and adds the new assumption $H' : s$. H' must be fresh.
- `tab_imp2' H H'.` - This presupposes $H : \neg(s \to t)$ is an assumption and adds the new assumption $H' : \neg t$. H' must be fresh.
- `tab_all H y H'.` - This presupposes $H : \forall x.s$ is an assumption and adds the new assumption $H' : s^x_y$. (Coq's substitution will avoid capture.) H' must be a fresh name.
- `tab_inh y.` - This adds the assumption $y : V$, where y must be a fresh variable.
- `tab_all' H y H'.` - This presupposes $H : \neg\forall x.s$ is an assumption and adds the new assumption $H' : \neg s^x_y$. (Coq's substitution will avoid capture.) H' must be a fresh name. y and H' must be fresh.

13.4.1 Simulating the Examples in Coq

We now reconsider the tableau refutations from Section 13.3 and show how to simulate the refutations in Coq.

Example 13.4.1 (Coq Simulation of Example 13.3.1) In Example 13.3.1 we constructed a tableau refutation using $\mathcal{T}_{\neg\forall}$ (with x), $\mathcal{T}_{\neg\neg}$ and $\mathcal{T}_\bot$. The Coq simulation is easy:

```
Example FalseElim_Tab : (forall x, False -> E x x).
tab_start H1.
tab_all' H1 x H2.
tab_imp1' H2 H3.
tab_bot H3.
Qed.
```

We can now show the tableau we constructed in Example 13.3.1 with the labels from the Coq simulation:

$$\begin{array}{c} \mathsf{H1} : \neg\forall x.\bot \to \mathsf{E}\,x\,x \\ \mathsf{H2} : \neg(\bot \to \mathsf{E}\,x\,x) \\ \mathsf{H3} : \bot \end{array}$$

Note that we have used `tab_imp1'` instead of `tab_imp'` to suppress the unused assumption $\neg\mathsf{E}\,x\,x$. □

Example 13.4.2 (Coq Simulation of Example 13.3.2) We next simulate the tableau refutation proving the Drinker's Lemma. Note the use of `tab_inh x` to put the variable x into the assumptions of the goal so that we can use x in the $\mathcal{T}_\forall$ rule.

```
Example Drinker_Tab : existsF x, forall y, E x x -> E y y.
tab_start H1.
tab_imp1' H1 H2.
tab_inh x.
tab_all H2 x H3.
tab_all' H3 y H4.
tab_imp' H4 H5 H6.
tab_all H2 y H7.
tab_all' H7 z H8.
tab_imp' H8 H9 H10.
tab_conflict H9 H6.
Qed.
```

We now show the tableau refutation with the corresponding Coq labels for the assumptions. Note that the refutation requires a use of $\mathcal{T}_\alpha$. This is reflected by giving two different, but alpha equivalent, formulas the label H2.

$$\begin{gathered}
\mathsf{H1} : \neg\neg\forall x.\neg\forall y.\mathsf{E}\,x\,x \to \mathsf{E}\,y\,y \\
\mathsf{H2} : \forall x.\neg\forall y.\mathsf{E}\,x\,x \to \mathsf{E}\,y\,y \\
\mathsf{H3} : \neg\forall y.\mathsf{E}\,x\,x \to \mathsf{E}\,y\,y \\
\mathsf{H4} : \neg(\mathsf{E}\,x\,x \to \mathsf{E}\,y\,y) \\
\mathsf{H5} : \mathsf{E}\,x\,x \\
\mathsf{H6} : \neg\mathsf{E}\,y\,y \\
\mathsf{H2} : \forall y.\neg\forall z.\mathsf{E}\,y\,y \to \mathsf{E}\,z\,z \\
\mathsf{H7} : \neg\forall z.\mathsf{E}\,y\,y \to \mathsf{E}\,z\,z \\
\mathsf{H8} : \neg(\mathsf{E}\,y\,y \to \mathsf{E}\,z\,z) \\
\mathsf{H9} : \mathsf{E}\,y\,y \\
\mathsf{H10} : \neg\mathsf{E}\,z\,z
\end{gathered}$$

The formulas labeled H5 and H10 are unused. We can simplify the presentation of the refutation by using the tactics `tab_imp2' H4 H6` and `tab_imp1' H8 H9` and omitting these two assumptions.

```
Example Drinker_Tab : existsF x, forall y, E x x -> E y y.
tab_start H1.
tab_imp1' H1 H2.
tab_inh x.
tab_all H2 x H3.
tab_all' H3 y H4.
tab_imp2' H4 H6. (*** instead of tab_imp' H4 H5 H6. ***)
tab_all H2 y H7.
tab_all' H7 z H8.
tab_imp1' H8 H9.  (*** instead of tab_imp' H8 H9 H10. ***)
tab_conflict H9 H6.
Qed.
```

Example 13.4.3 (Coq simulation of Example 13.3.3) The refutation proving Russell's law can be simulated as follows:

```
Example Russell_Tab : ~existsF x, forall y,  equivF (E x y) (~E y y).
tab_start H1.
tab_imp1' H1 H2.
tab_all' H2 x H3.
tab_imp1' H3 H4.
tab_all H4 x H5.
tab_imp' H5 H6 H7.
tab_imp1' H7 H8.
tab_imp H8 H9.
(*** Branch 1 ***)
tab_imp H6 H10.
(*** Branch 1.1 ***)
tab_conflict H10 H9.
(*** Branch 1.2 ***)
tab_conflict H10 H9.
(*** Branch 2 ***)
tab_imp H6 H10.
(*** Branch 2.1 ***)
tab_conflict H9 H10.
(*** Branch 2.2 ***)
tab_conflict H9 H10.
Qed.
```

The tableau refutation with labels is given below.

$$\mathsf{H1} : \neg\neg\exists x.\forall y.\mathsf{E}\,x\,y \leftrightarrow \neg\mathsf{E}\,y\,y$$
$$\mathsf{H2} : \neg\forall x.\neg\forall y.\mathsf{E}\,x\,y \leftrightarrow \neg\mathsf{E}\,y\,y$$
$$\mathsf{H3} : \neg\neg\forall y.\mathsf{E}\,x\,y \leftrightarrow \neg\mathsf{E}\,y\,y$$
$$\mathsf{H4} : \forall y.\mathsf{E}\,x\,y \leftrightarrow \neg\mathsf{E}\,y\,y$$
$$\mathsf{H5} : \mathsf{E}\,x\,x \leftrightarrow \neg\mathsf{E}\,x\,x$$
$$\mathsf{H6} : \mathsf{E}\,x\,x \rightarrow \neg\mathsf{E}\,x\,x$$
$$\mathsf{H7} : \neg\neg(\neg\mathsf{E}\,x\,x \rightarrow \mathsf{E}\,x\,x)$$
$$\mathsf{H8} : \neg\mathsf{E}\,x\,x \rightarrow \mathsf{E}\,x\,x$$

$\mathsf{H9} : \neg\neg\mathsf{E}\,x\,x$		$\mathsf{H9} : \mathsf{E}\,x\,x$	
$\mathsf{H10} : \neg\mathsf{E}\,x\,x$	$\mathsf{H10} : \neg\mathsf{E}\,x\,x$	$\mathsf{H10} : \neg\mathsf{E}\,x\,x$	$\mathsf{H10} : \neg\mathsf{E}\,x\,x$

Exercise 13.4.4 Use tableau to prove the following formulas by refuting the negation. Simulate the refutation in Coq and draw the tableau refutation with labels corresponding to the Coq assumptions.

a) $(\forall x\,y.\mathsf{E}\,x\,y) \rightarrow \forall y.\mathsf{E}\,y\,y$

```
Lemma full_ref_Tab : (forall x y, E x y) -> forall y, E y y.
```

b) $(\forall x.\exists y.\mathsf{E}\,x\,y) \to \forall x.\exists y.\exists z.\mathsf{E}\,x\,y \wedge \mathsf{E}\,y\,z$

```
Lemma total_total2_Tab : (forall x, existsF y, E x y)
      -> (forall x, existsF y, existsF z, andF (E x y) (E y z)).
```

c) $(\forall x.\mathsf{E}\,x\,x) \to \neg(\forall x.\neg\mathsf{E}\,x\,x)$

```
Lemma refl_not_irrefl_Tab : ((forall x, (E x x)) -> ~(forall x, ~(E x x))).
```

d) $\forall x\,y.((\mathsf{E}\,x\,y \to \mathsf{E}\,y\,x) \to \mathsf{E}\,x\,y) \to \mathsf{E}\,x\,y$

```
Lemma Peirce_Tab : (forall x y, ((E x y -> E y x) -> E x y) -> E x y).
```

13.4.2 A Similar Simulation of ND in Coq

We can also simulate natural deduction using tactics within the same section in Coq. We briefly describe the tactics, define the tactics and show two examples. Like the tableau case, we omit the α rule in Coq and use the fact that alpha equivalence is handled by Coq. For the same reason, we can ignore the side condition that $y \notin \mathrm{BV}(s)$ in the $\forall\mathbf{E}$ rule and liberalize the $\forall\mathbf{I}$ rule to allow any fresh variable to be used instead of only using the particular variable bound by the quantifier. We define a tactic `nd_inh` which is the same as `tab_inh` for the same purpose, to introduce a new variable $y : V$ into the assumptions so that it is available for the $\forall\mathbf{E}$ rule. The tactic `nd_Contra` applies the double negation lemma, and is defined the same way as `tab_start`. Unlike `tab_start`, `nd_Contra` may be used at any point in the proof and may be used more than once. The introduction tactics `nd_ImpI` and `nd_AllI` simply do `intros` with the given name. To ensure that `nd_AllI` is only used when the claim of the goal is a universal quantifier (over V) the tactic first checks if the goal has the right form. Similarly, `nd_ImpI` checks that the claim of the goal is *not* a universal quantifier (over V) before doing the introduction. The elimination tactics `nd_AllE` and `nd_ImpE` apply simple lemmas `NDImpE` and `NDAllE`. With the `nd_ImpE` tactic we must explicitly give the formula on the left of the implication. With the `nd_AllE` tactic we must explicitly give the body of the quantifier (as $\lambda x : V.s$) and the variable used for the instantiation.

```
Lemma NDImpE {Y : Prop} (X : Prop) : (X -> Y) -> X -> Y.
intros Y X a x. apply a. apply x.
Qed.

Lemma NDAllE (p : V -> Prop) (y : V) : (forall x:V, p x) -> p y.
intros p y a. apply a.
Qed.

Ltac nd_Assu H := (exact H).

Ltac nd_Contra H := (apply DN; intros H).
```

```
Ltac nd_AllI y :=
 (match goal with |- (forall x:V, _) => intros y
           | _ => fail 1 "Claim is not a forall" end).

Ltac nd_ImpI H :=
 (match goal with |- (forall x:V, _) => fail 1 "Claim is not an implication"
           | _ => intros H end).

Ltac nd_ImpE s := (apply (NDImpE s)).
Ltac nd_AllE p y := (apply (NDAllE p y)).

Ltac nd_inh y := (destruct inh as [y _]).
```

A simulation of a first-order ND proof in Coq is a proof that only uses the seven tactics

```
nd_Assu H.
nd_Contra H.
nd_ImpI H.
nd_AllI y.
nd_ImpE s.
nd_AllE p y.
nd_inh y.
```

Here is a quick summary of how each tactic is used.

- `nd_Assu H.` - This finishes the goal where the claim is s and $H : s$ is an assumption.
- `nd_Contra H.` - If s is the claim, then a new assumption $H : \neg s$ is added and the new claim is $\bot$.
- `nd_ImpI H.` - This presupposes the claim is $s \to t$ and H is fresh. A new assumption $H : s$ is added and the new claim is t.
- `nd_AllI y.` - This presupposes the claim is $\forall x.s$ and y is fresh. A new assumption $y : V$ is added and the new claim is s^x_y. (Coq's substitution avoids capture.)
- `nd_ImpE s.` - If t is the claim, then two new subgoals are created, one with claim $s \to t$ and the other with claim s.
- `nd_AllE p y.` - Here p has type $V \to \mathit{Prop}$ and $y : V$. The claim must be (py) up to conversion. The new claim is $\forall x.px$ for some x (of Coq's choice) where x is not free in p.
- `nd_inh y.` - This adds $y : V$ as an assumption.

We consider two simple examples.

Example 13.4.5 Consider the valid formula $\forall x.\bot \to \mathsf{E}xx$. We give an ordinary Coq proof script. A simpler Coq script would use `destruct` on the assumption `H1`, but we will make use of `apply DN` instead. The reason is to make the proof closer to the first-order ND system.

```
Example FalseElim_1 : (forall x, False -> E x x).
intros x H1.
apply DN.
intros H2.
apply H1.
Qed.
```

Next we give a natural deduction derivation justifying $\vdash \forall x.\bot \to \mathsf{E}xx$. We add labels to the assumptions corresponding to the labels in the Coq simulation.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\mathtt{H1}:\bot, \mathtt{H2}:\neg\mathsf{E}xx \Rightarrow \bot}\,\mathbf{Assu}}{\mathtt{H1}:\bot \Rightarrow \mathsf{E}xx}\,\mathbf{Contra}}{\Rightarrow \bot \to \mathsf{E}xx}\,{\to}\mathbf{I}}{\Rightarrow \forall x.\bot \to \mathsf{E}xx}\,\forall\mathbf{I}$$

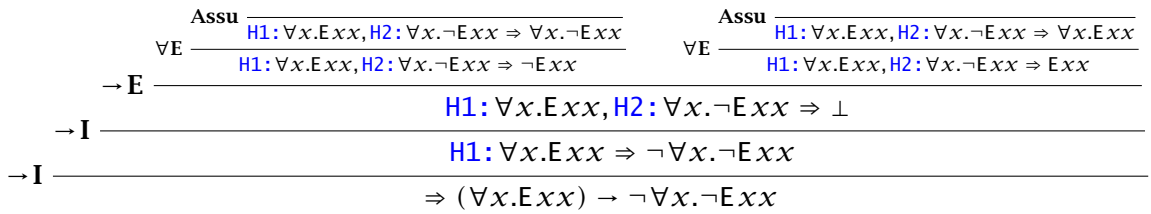
Finally we give a Coq simulation of the ND derivation.

```
Example FalseElim_ND : (forall x, False -> E x x).
nd_AllI x.
nd_ImpI H1.
nd_Contra H2.
nd_Assu H1.
Qed.
```

Example 13.4.6 Consider the valid formula $(\forall x.\mathsf{E}xx) \to \neg(\forall x.\neg\mathsf{E}xx)$. We first give an ordinary Coq proof script.

```
Lemma refl_not_irrefl_1 : (forall x, E x x) -> ~(forall x, ~E x x).
intros H1 H2.
destruct inh as [x _].
apply (H2 x).
apply H1.
Qed.
```

We now give the ND derivation. We add labels to the assumptions corresponding to the labels in Coq.



Now we give Coq simulation of the ND derivation.

```
Lemma refl_not_irrefl_ND : (forall x, E x x) -> ~(forall x, ~E x x).
nd_ImpI H1.
nd_ImpI H2.
nd_inh x.
nd_ImpE (E x x).
(*** Subgoal 1 ***)
nd_AllE (fun x => ~E x x) x.
nd_Assu H2.
(*** Subgoal 2 ***)
nd_AllE (fun x => E x x) x.
nd_Assu H1.
Qed.
```

Exercise 13.4.7 Justify $\vdash \forall x.\neg\neg \mathsf{E}xx \to \mathsf{E}xx$ with a natural deduction derivation and with a simulation of ND in Coq.

```
Lemma DN_ND : (forall x, ~~E x x -> E x x).
```

Exercise 13.4.8 Justify $\vdash \forall x.(\forall y.\neg\neg \mathsf{E}xy) \to \mathsf{E}xx$ with a natural deduction derivation and with a simulation of ND in Coq.

```
Lemma DN2_ND : forall x, (forall y, ~~E x y) -> E x x.
```

14 First-Order Completeness

In this chapter we prove completeness of the first-order tableau system. Completeness of the first-order natural deduction system follows (see Proposition 13.2.3). Along the way, we will note decidability of satisfiability for a certain fragment of first-order logic.

14.1 Evident Sets and Herbrand Models

Fix a variable $x_0 \in \mathrm{Var}$. For a set A of formulas, we define the **Herbrand universe** $\mathcal{H}(A)$ of A as follows:

$$\mathcal{H}(A) := \begin{cases} \mathrm{FV}(A) & \text{if } \mathrm{FV}(A) \neq \emptyset \\ \{x_0\} & \text{otherwise.} \end{cases}$$

The Herbrand universe of a branch will provide us with a sufficient set of variables to use in the $\mathcal{T}_\forall$ rule.

We now define **evidence conditions** that a set A of formulas may satisfy.

$\mathcal{E}_\bot$ $\bot \notin A$.

$\mathcal{E}_\neg$ If $\neg s \in A$, then $s \notin A$.

$\mathcal{E}_\to$ If $s \to t \in A$, then $\neg s \in A$ or $t \in A$.

$\mathcal{E}_{\neg\to}$ If $\neg(s \to t) \in A$, then $s \in A$ and $\neg t \in A$.

$\mathcal{E}_\forall$ If $(\forall x.s) \in A$, then for all $y \in \mathcal{H}(A)$, there is some t such that $s \sim_\alpha t$, $y \notin \mathrm{BV}(t)$ and $t^x_y \in A$.

$\mathcal{E}_{\neg\forall}$ If $(\neg\forall x.s) \in A$, then $\neg s^x_y \in A$ for some $y \notin \mathrm{BV}(s)$.

We say A is **evident** if it satisfies these evidence conditions.

The major result we prove is that every evident set A is satisfiable. Let A be an evident set. The **Herbrand model of A** is the graph $G = (V, E)$ defined by taking V to be the variables in the Herbrand universe and E to be the edges determined by the set A. That is, define $V := \mathcal{H}(G)$. We define E such that $(x, y) \in E$ iff $\mathsf{E}xy \in A$.

Lemma 14.1.1 Let A be an evident set and $G = (V, E)$ be the Herbrand model of A. Let φ be an assignment into G such that $\varphi x = x$ for each $x \in \mathcal{H}(A)$. For each formula u the following two properties hold:

(1^u) If $u \in A$, then $G \vDash_\varphi u$.

(2^u) If $\neg u \in A$, then $G \nvDash_\varphi u$.

Proof We prove this by induction on the size of the formula u. The size $|s|$ of a formula s is defined recursively as

$$\begin{aligned} |\bot| &:= 0 \\ |\mathsf{E}xy| &:= 0 \\ |s \to t| &:= 1 + |s| + |t| \\ |\forall x.s| &:= 1 + |s| \end{aligned}$$

Note that $|s^x_y| = |s|$ and that $|s| = |t|$ whenever $s \sim_\alpha t$.

If u is $\bot$, we have $(1^\bot)$ and $(2^\bot)$ by $\mathcal{E}_\bot$ and $G \nvDash_\varphi \bot$.

Assume u is $\mathsf{E}xy$ and $u \in A$. Clearly $x, y \in \mathcal{H}(A)$ and $(x, y) \in E$. Hence $G \vDash_\varphi u$.

Assume u is $\mathsf{E}xy$ and $\neg u \in A$. Again, $x, y \in \mathcal{H}(A)$. By $\mathcal{E}_\neg$, $\mathsf{E}xy \notin A$. Hence $(x, y) \notin E$ and so $G \nvDash_\varphi u$.

Assume u is $s \to t$ and $u \in A$. By $\mathcal{E}_\to$ we know either $\neg s \in A$ or $t \in A$. First, assume $\neg s \in A$. By inductive hypothesis we know (2^s) and so $G \nvDash_\varphi s$ and $G \vDash_\varphi u$. Next, assume $t \in A$. By inductive hypothesis we know (1^t) and so $G \vDash_\varphi t$ and $G \vDash_\varphi u$.

Assume u is $s \to t$ and $\neg s \in A$. By $\mathcal{E}_{\neg\to}$ we know $s \in A$ and $\neg t \in A$. By inductive hypothesis we know (1^s) and (2^t). Hence $G \vDash_\varphi s$ and $G \nvDash_\varphi t$. Thus $G \nvDash_\varphi u$.

Assume u is $\forall x.s$ and $u \in A$. We must prove $G \vDash_\varphi \forall x.s$. Let $y \in V$ (i.e., $y \in \mathcal{H}(A)$) be given. We will prove $G \vDash_{\varphi^x_y} s$. By $\mathcal{E}_\forall$ there is some t such that $s \sim_\alpha t$, $y \notin \mathrm{BV}(t)$ and $t^x_y \in A$. Since $|t^x_y| = |s|$ we can apply the inductive hypothesis $(1^{t^x_y})$ to obtain $G \vDash_\varphi t^x_y$. By Proposition 11.4.8 and the fact that $\varphi y = y$ we have $G \vDash_{\varphi^x_y} t$. By Proposition 11.4.7 we know $G \vDash_{\varphi^x_y} s$, as desired.

Assume u is $\neg\forall x.s$ and $u \in A$. By $\mathcal{E}_{\neg\forall}$ we know $\neg s^x_y \in A$ for some $y \notin \mathrm{BV}(s)$. By the inductive hypothesis $(2^{s^x_y})$ we know $G \nvDash_\varphi s^x_y$. By Proposition 11.4.8 we have $G \nvDash_{\varphi^x_{\varphi y}} s$. Hence $G \nvDash \forall x.s$. ∎

Theorem 14.1.2 (First-Order Model Existence) Let A be an evident set of (first-order) formulas. Then A is satisfiable. That is, there is a graph G and an assignment φ into G such that $G \vDash_\varphi A$.

Proof Let G be the Herbrand model of A and take φ such that $\varphi x := x$ for each $x \in \mathcal{H}(A)$. For variables $z \notin \mathcal{H}(A)$, we take φz to be some element in $\mathcal{H}(A)$. By Lemma 14.1.1, we know $G \vDash_\varphi A$. ∎

Exercise 14.1.3 Let x, y, z, w be distinct variables. For each of the following sets, determine whether or not the set is evident. If the set is evident, compute the Herbrand model of the set. If the set is not evident, list all the conditions that fail.

a) $\emptyset$

b) $\{\bot\}$

c) $\{\neg\bot\}$

d) $\{\mathsf{E}\,x\,y, \mathsf{E}\,y\,y\}$

e) $\{\forall x.\mathsf{E}\,x\,y, \mathsf{E}\,y\,y\}$

f) $\{\forall y.\mathsf{E}\,x\,y, \mathsf{E}\,y\,y\}$

g) $\{\forall x.\mathsf{E}\,x\,x, \forall x\,y.\mathsf{E}\,x\,y \to \mathsf{E}\,y\,x, \mathsf{E}\,z\,z\}$

h) $\{\neg\forall x.\mathsf{E}\,x\,x, \forall x.\exists y.\mathsf{E}\,x\,y, \exists y.\mathsf{E}\,z\,y, \exists y.\mathsf{E}\,w\,y, \neg\mathsf{E}\,z\,z, \neg\neg\mathsf{E}\,z\,w, \neg\neg\mathsf{E}\,w\,z, \mathsf{E}\,z\,w, \mathsf{E}\,w\,z\}$

Exercise 14.1.4 Let G be the Herbrand model of the following evident set.

$$\{\mathsf{E}\,x\,y, \mathsf{E}\,y\,y, \mathsf{E}\,z\,z, \mathsf{E}\,z\,y\}$$

Determine which of the following sentences is true in the model.

a) $\forall x.\mathsf{E}\,x\,x$

b) $\forall x\,y.\mathsf{E}\,x\,y \to \mathsf{E}\,y\,x$

c) $\forall x\,y\,z.\mathsf{E}\,x\,y \to \mathsf{E}\,y\,z \to \mathsf{E}\,x\,z$

d) $\forall x.\exists y.\mathsf{E}\,x\,y$

Exercise 14.1.5

a) Give an example of an evident set A such that the Herbrand model of A has infinitely many vertices and infinitely many edges.

b) Give an example of an evident set A such that the Herbrand model of A has infinitely many vertices, but only finitely many edges.

14.2 A Decidable Fragment

In this section we will consider classes of formulas which have a special form. It will turn out that if all the formulas in a branch A have one of these special forms, then we can decide satisfiability of A. In particular, we will be able to use the tableau system to decide satisfiability of A.

A formula is **quantifier-free** if it has no occurrence of $\forall$. We say a formula s is a $\forall^*$**-formula** if it has the form

$$\forall y_1 \cdots \forall y_m.t$$

where t is quantifier-free and $m \geq 0$. We say a formula s is a **$\exists^*$-formula** if it has the form

$$\exists y_1 \cdots \exists y_m . t$$

where t is quantifier-free and $m \geq 0$. We say a formula s is a **$\exists^*\forall^*$-formula** if it has the form

$$\exists x_1 \cdots \exists x_n \forall y_1 \cdots \forall y_m . t$$

where t is quantifier-free and $n, m \geq 0$. We say a formula s is a **$\forall^*\exists^*$-formula** if it has the form

$$\forall x_1 \cdots \forall x_n \exists y_1 \cdots \exists y_m . t$$

where t is quantifier-free and $n, m \geq 0$.

Note the following:

- If s is quantifier-free, then s is also a $\forall^*$-formula and an $\exists^*$-formula.
- If s is a $\forall^*$-formula, then s is also an $\exists^*\forall^*$-formula and a $\forall^*\exists^*$-formula.
- If s is a $\exists^*$-formula, then s is also an $\exists^*\forall^*$-formula and a $\forall^*\exists^*$-formula.

We say a branch A is a **Bernays-Schönfinkel- (BS-) branch** if every formula in A is an $\exists^*\forall^*$-formula. We will prove that satisfiability is decidable for BS-branches. In particular, this means satisfiability of $\exists^*\forall^*$-formulas is decidable. This also means that validity if $\forall^*\exists^*$-formulas is decidable.

We say a branch A is a **$\forall^*$-branch** if every formula in A is a $\forall^*$-formulas. We say a branch A is **pure** if no variable occurs both free and bound in A and $x_0 \notin \mathrm{BV}(A)$. We prove satisfiability of pure $\forall^*$-branches is decidable. The other decidability results will follow from this.

The following lemma will be helpful.

Lemma 14.2.1 Let A be a pure $\forall^*$-branch. If A is neither closed nor evident, then at least one of the following hold:

1. There is some $s \to t \in A$ such that $\neg s \notin A$ and $t \notin A$.
2. There is some $\neg(s \to t) \in A$ such that $\{s, \neg t\} \notin A$.
3. There is some $\forall x . s \in A$ and $y \in \mathcal{H}(A)$ such that $s^x_y \notin A$.

Proof Assume A is not closed, but none of the three options hold. We prove A is evident.

$\mathcal{E}_\bot$ $\bot \notin A$ since A is not closed.

$\mathcal{E}_\neg$ If $\neg s \in A$, then $s \notin A$ since A is not closed.

$\mathcal{E}_\to$ If $s \to t \in A$, then $\neg s \in A$ or $t \in A$ since otherwise the first option holds.

$\mathcal{E}_{\neg\to}$ If $\neg(s \to t) \in A$, then $s \in A$ and $\neg t \in A$ since otherwise the second option holds.

$\mathcal{E}_\forall$ Suppose $(\forall x.s) \in A$ and $y \in \mathcal{H}(A)$. Since A is pure, $y \notin \mathrm{BV}(s)$. We must have $s^x_y \in A$ since otherwise the third option holds.

$\mathcal{E}_{\neg\forall}$ We never have $(\neg\forall x.s) \in A$ since A is a $\forall^*$-branch. ∎

We can use a restricted version of the tableau procedure to give a nondeterministic algorithm findEvident that takes a pure $\forall^*$-branch and either returns refutable or returns an **evident extension** A' (i.e., a branch A' such that $A \subseteq A'$). The idea of the procedure is to apply the tableau rules with the following restrictions:

1. Only apply a rule if every alternative is a strict superset of the head.
2. Only apply $\mathcal{T}_\forall$ with y in the Herbrand universe of the branch.
3. Never apply $\mathcal{T}_\alpha$ or $\mathcal{T}_{\neg\forall}$.

If findEvident(A) returns refutable, then A will be refutable and hence unsatisfiable. If findEvident(A) returns A', then we know A' is satisfiable by Theorem 14.1.2 and so A is satisfiable.

We first define the closure $\mathsf{cl}(A)$ of a branch to be the least set such that

- If $s \in A$, then $s \in \mathsf{cl}(A)$.
- If $s \to t \in \mathsf{cl}(A)$, then $\neg s \in \mathsf{cl}(A)$ and $t \in \mathsf{cl}(A)$.
- If $\neg(s \to t) \in \mathsf{cl}(A)$, then $s \in \mathsf{cl}(A)$ and $\neg t \in \mathsf{cl}(A)$.
- If $\forall x.s \in \mathsf{cl}(A)$ and $y \in \mathcal{H}A$, then $s^x_y \in \mathsf{cl}(A)$.

It is not difficult to see that the set $\mathsf{cl}(A)$ is finite.

The algorithm findEvident(A) is defined as follows:

- If A is closed (i.e., $\bot \in A$ or $\{s, \neg s\} \subseteq A$), then return refutable
- If A is evident, then return A.
- Otherwise, Lemma 14.2.1 applies. Nondeterministically choose one of the following options.
  1. Choose some $s \to t \in A$ such that $\neg s \notin A$ and $t \notin A$. Do either of the following:
     a) If findEvident$(A, \neg s)$ returns an evident set A', then return A'. Otherwise, return findEvident(A, t).
     b) If findEvident(A, t) returns an evident set A', then return A'. Otherwise, return findEvident$(A, \neg s)$.
  2. Choose some $\neg(s \to t) \in A$ such that $\{\neg s, t\} \subseteq A$. Return findEvident$(A, \neg s, t)$.
  3. Choose some $\forall x.s \in A$ and $y \in \mathcal{H}(A)$ such that $s^x_y \notin A$. Return findEvident(A, s^x_y).

Consider a possible recursive call findEvident(A') when computing

$\mathsf{findEvident}(A)$. It is easy to check that $\mathcal{H}(A) = \mathcal{H}(A')$ and $\mathsf{cl}(A) = \mathsf{cl}(A')$. Since $\mathsf{cl}(A)$ is finite and $A \subsetneq A' \subseteq \mathsf{cl}(A)$, we know the algorithm terminates whichever nondeterministic choices are made.

Proposition 14.2.2 Satisfiability of pure $\forall^*$-branches is decidable.

Proof It is clear that if $\mathsf{findEvident}(A)$ returns $\mathsf{refutable}$, then A is, in fact, refutable. (Each nondeterministic option corresponds to applying tableau rule.) Otherwise, $\mathsf{findEvident}(A)$ returns an evident extension A'. We know A' is satisfiable by Theorem 14.1.2. Since $A \subseteq A'$ we conclude A is satisfiable. ∎

It is now easy to see that satisfiability of $\forall^*$-branches is decidable.

Proposition 14.2.3 Satisfiability of $\forall^*$-branches is decidable.

Proof Given a $\forall^*$-branch A, we can replace each $s \in A$ with a formula $t \in A$ such that $s \sim_\alpha t$ and such that $\mathrm{BV}(t) \cap \mathrm{FV}(A) = \emptyset$ and $x_0 \notin \mathrm{BV}(t)$. Let B be the branch consisting of these formulas t. Clearly B is a pure $\forall^*$-branch and satisfiability of A is equivalent to satisfiability of B by Proposition 11.4.7. Decidability follows from Proposition 14.2.2. ∎

Finally we prove decidability of satisfiability for BS is noting that we can get rid of the leading $\exists$-quantifiers. This is justified by the following proposition.

Proposition 14.2.4 Let A be a branch, s be formula and x and y be variables. Assume $y \notin \mathrm{FV}(A) \cup \mathrm{FV}(s) \cup \mathrm{BV}(s)$. Then $A \cup \{\exists x.s\}$ is satisfiable iff $A \cup \{s^x_y\}$ is satisfiable.

Proof Suppose $G \vDash_\phi A \cup \{\exists x.s\}$ where $G = (V, E)$. Since $G \vDash_\phi \exists x.s$ there is some $a \in V$ such that $G \vDash_{\phi^x_a} s$ (by Exercise 11.4.1). Since $y \notin \mathrm{FV}(s)$ we know $G \vDash_{(\phi^y_a)^x_a} s$ by Proposition 11.4.3. Since $\phi^y_a(y) = a$ we know $G \vDash_{\phi^y_a} s^x_y$ by Proposition 11.4.8. Since $y \notin \mathrm{FV}(A)$ we know $G \vDash_{\phi^y_a} A$ by Proposition 11.4.3. Hence $G \vDash_{\phi^y_a} A \cup \{s^x_y\}$ as desired.

The other direction is easier. Suppose $G \vDash_\phi A \cup \{s^x_y\}$. By Proposition 11.4.8 we know $G \vDash_{\phi^x_{\phi y}} s$. Hence $G \vDash_\phi \exists x.s$ (by Exercise 11.4.1) and so $G \vDash_\phi A \cup \{\exists x.s\}$ as desired. ∎

Theorem 14.2.5 Satisfiability of BS-branches is decidable.

Proof We can preprocess BS-branches A to obtain $\forall^*$-branches as follows. If A is not already a $\forall^*$-branch, choose some $\exists x.s$ in A and choose some variable $y \notin \mathrm{FV}(A) \cup \mathrm{FV}(s) \cup \mathrm{BV}(s)$. By Proposition 14.2.4 A is satisfiable iff $(A \setminus \{\exists x.s\}) \cup \{s^x_y\}$ is satisfiable. Note that $(A \setminus \{\exists x.s\}) \cup \{s^x_y\}$ has one fewer $\exists$-quantifiers. This process clearly terminates with an equi-satisfiable $\forall^*$-branch. We finish the proof with an appeal to Proposition 14.2.3. ∎

Exercise 14.2.6 Let x, y, z, w, w' be distinct variables. Determine whether or not the following pure $\forall^*$-branches are satisfiable. If the branch is satisfiable, give an evident extension. If the branch is unsatisfiable, give a tableau refutation.

a) $\{\forall x.\mathsf{E}xx, \forall xy.\mathsf{E}xy \to \mathsf{E}yx\}$

b) $\{\forall x.\neg\mathsf{E}xx, \forall xy.\mathsf{E}xy \to \mathsf{E}yx\}$

c) $\{\mathsf{E}wz, \neg\mathsf{E}zz, \forall xy.\mathsf{E}xy \to \mathsf{E}xx\}$

d) $\{\mathsf{E}wz, \neg\mathsf{E}ww, \forall xy.\mathsf{E}xy \to \mathsf{E}xx\}$

e) $\{\mathsf{E}wz, \forall xy.\mathsf{E}xy \to \neg\mathsf{E}yx\}$

f) $\{\mathsf{E}wz, \forall xy.\neg\mathsf{E}xy \to \mathsf{E}yx\}$

g) $\{\mathsf{E}ww', \neg\mathsf{E}ww, \forall xy.\mathsf{E}xy \to \mathsf{E}yx, \forall xyz.\mathsf{E}xy \to \mathsf{E}yz \to \mathsf{E}xz\}$

14.3 Completeness

We now prove completeness. That is, we prove that every unsatisfiable branch is refutable. In fact, we will prove the contrapositive. We say a branch A is **consistent** if there is a branch A' such that $A \subseteq A'$ and A' is not refutable. We will assume A is a branch that is not refutable and prove A is satisfiable. We prove this in two steps. We first prove that every consistent branch can be extended to an evident set.

Lemma 14.3.1 Let A be an consistent branch. There is an evident set B such that $A \subseteq B$.

Proof Let $u_0, u_1, u_2, \ldots$ be an enumeration of all formulas. We define an infinite chain of consistent branches

$$A_0 \subseteq A_1 \subseteq A_2 \subseteq \cdots$$

as follows. We take A_0 to be A (assumed to be consistent). Assume A_n is defined. We define A_{n+1} as follows. If $A_n \cup \{u_n\}$ is not consistent, then let A_{n+1} be A_n. Otherwise, assume $A_n \cup \{u_n\}$ is consistent. First, assume u_n is of the form $\neg\forall x.t$. Choose some $A' \supseteq A_n \cup \{u_n\}$ such that A' is not refutable. Choose some $y \notin \mathrm{FV}(A') \cup \mathrm{BV}(t)$. Let A_{n+1} be $A_n \cup \{u_n, \neg t^x_y\}$. If $A', \neg t^x_y$ were refutable, then A' would be refutable by $\mathcal{T}_{\neg\forall}$. Hence $A', \neg t^x_y$ witnesses that the branch A_{n+1} is consistent. Next, assume u_n is not of the form $\neg\forall x.t$. In this case we define A_{n+1} to be $A_n \cup \{u_n\}$.

Let B be $\bigcup_n A_n$. Clearly $A \subseteq B$. We check B is evident.

$\mathcal{E}_\perp$ If $\perp \in B$, then $\perp \in A_n$ for some n, contradicting consistency of A_n.

$\mathcal{E}_\neg$ If $\neg s \in B$ and $s \in B$, then $\{\neg s, s\} \subseteq A_n$ for some n, contradicting consistency of A_n.

$\mathcal{E}_{\rightarrow}$ Assume $s \rightarrow t \in B$. Let n and m be such that u_n is $\neg s$ and u_m is t. Let $r \geq \max(n, m)$ such that $s \rightarrow t \in A_r$. Choose $A' \supseteq A_r$ such that A' is not refutable. If $A' \cup \{\neg s\}$ and $A' \cup \{t\}$ were both refutable, then A' would be refutable by $\mathcal{T}_{\rightarrow}$. Thus either $A' \cup \{\neg s\}$ or $A' \cup \{t\}$ is not refutable. First assume $A' \cup \{\neg s\}$ is not refutable. Since $n \leq r$ we know $A_n \cup \{\neg s\} \subseteq A' \cup \{\neg s\}$. Hence $A_n \cup \{\neg s\}$ is consistent and we must have $u_n \in A_{n+1}$ by construction. That is, $\neg s \in A_{n+1}$ and so $\neg s \in B$ as desired. Next assume $A' \cup \{t\}$ is consistent. In this case, $t \in A_{m+1}$ since $A_m \cup \{t\} \subseteq A' \cup \{t\}$.

$\mathcal{E}_{\neg\rightarrow}$ Assume $\neg(s \rightarrow t) \in B$. Let n and m be such that u_n is s and u_m is $\neg t$. Let $r \geq \max(n, m)$ such that $\neg(s \rightarrow t) \in A_r$ Choose $A' \supseteq A_r$ such that A' is not refutable. We know $A' \cup \{s, \neg t\}$ is not refutable (via $\mathcal{T}_{\neg\rightarrow}$). Hence $s \in A_{n+1}$ and $\neg t \in A_{m+1}$.

$\mathcal{E}_{\forall}$ Assume $(\forall x.s) \in B$ and $y \in \mathcal{H}(B)$. Let t be a formula such that $s \sim_\alpha t$ and $y \notin \mathrm{BV}(t)$. Let n be such that u_n is t^x_y. Let $r \geq n$ be such that $\forall x.s \in A_r$. Choose $A' \supseteq A_r$ such that A' is not refutable. We know $A' \cup \{\forall x.t, t^x_y\}$ is not refutable since otherwise A' would be refutable by $\mathcal{T}_\alpha$ and $\mathcal{T}_\forall$. Hence $t^x_y \in A_{n+1}$.

$\mathcal{E}_{\neg\forall}$ Assume $(\neg\forall x.s) \in B$. Let n be such that u_n is $\neg\forall x.s$. By construction there is some $y \notin \mathrm{BV}(s)$ such that $\neg s^x_y$ is in A_{n+1}. ∎

Theorem 14.3.2 (Completeness) If a branch A is unsatisfiable, then A is refutable.

Proof Assume A is not refutable. Clearly, A witnesses that A is consistent. By Lemma 14.3.1 there is an evident set B such that $A \subseteq B$. We know B is satisfiable by Theorem 14.1.2 and so A is satisfiable. ∎

Coq Summary

This appendix lists and classifies the Coq commands and tactics used in this book. It also lists some predefined variables. Consult the Coq reference manual for more information.

A.1 Commands

The Coq shell processes commands. Commands start with a keyword and end with a dot.

Definitions

- *Definition* $x : s := t$ Defines variable x.
- *Inductive* $x : s := x_1 : s_1 \mid \cdots \mid x_n : s_n$ Defines variables x and $x_1, \ldots, x_n$.
- *Fixpoint* Abbreviates *Definition* $x : s :=$ *fix* ... .
- *Notation* ... Defines a notation
- *Implicit Arguments* $x\ [x_1 \cdots x_n]$ Defines implicit argument notation for defined variable x.
- *Ltac* $x\ x_1 \cdots x_n := t$ Defines a tactic x taking arguments $x_1, \ldots, x_n$.

Proof Synthesis

- *Definition* $x : s$ Starts synthesis of a term of type s.
- *Defined* Ends synthesis and defines variable x.
- *Qed* Like *Defined* but defined variables becomes irreducible.
- *Show Proof* Shows current proof state.
- *Goal* s Like *Definition* $x : s$ where x is generated by Coq.
- *Theorem* $x : s$ Like *Definition* $x : s$.
- *Example* Synonym for Definition.
- *Lemma* Synonym for Theorem.
- *Corollary* Synonym for Theorem.

Requests

- *Check* t Elaborates and type checks t.
- *Eval* t *in* s Applies tactic t to s.

- *Print x* Prints information about defined variable x.
- *About x* Says in which library x is defined.

Sections

- *Section x* Opens a section x.
- *Variable x : t* Declares a local parameter x.
- *End x* Ends section x.

A.2 Tactics

Tactics are commands that provide for type-directed synthesis of terms. They can only be used in proof synthesis mode.

Basic Tactics

- *refine t* Refines the underlines in t such that t proves the claim. Introduces subgoals for underlines it cannot synthesize.
- *exact t* Like refine but fails if an underline cannot be synthesized.
- *apply t* Smart version of refine. Tries to refine t. If this fails, it tries to refine t _, then t _ _, and so on. Uses *pattern* to synthesize functions.
- *intros x* Synthesizes lambdas and creates subgoal for remaining body. Synthesizes matches for patterns.

Let Tactics

Synthesize let terms.

- *assert (x := t)* Creates assumption $x : u$ where $t : u$.
- *assert (x : t)* Creates subgoal for t and assumption $x : t$ in current subgoal.
- *pose (x := t)* Behaves like a local definition, x is delta reducible.

Conversion Tactics

Act on the claim. Can be applied to an assumption x by adding "*in x*".

- *cbv* Reduces to normal form.
- *cbv beta* Performs beta reduction.
- *pattern t* Patterns out subterm t by creating a beta redex st.
- *pattern t at n* Patterns out n-th occurrence of subterm t.
- *unfold x* Delta reduces x, then beta reduces.
- *fold x* Undoes unfold if there was no beta reduction.
- *red* Performs beta and delta reduction, delta only at top level.

- *simpl* Performs beta and iota reduction, plus delta reductions that exhibit iota redexes.

Logical Tactics

- *generalize t* Weakens claim with premise u where $t : u$.
- *clear x* Deletes assumption x.
- *revert x* Equivalent to *generalize x*; *clear x*.
- *absurd t* Replaces claim with subgoals for $\neg t$ and t.
- *split* Splits conjunctive claim.
- *left* Strengthens disjunctive claim to left side.
- *right* Strengthens disjunctive claim to right side.
- *exists t* Strengthens existential claim to witness t.

Equational Tactics

- *reflexivty* Proves claim with introduction rule for equations.
- *rewrite t* Rewrites subterm u of claim to v provided $t : u{=}v$.
- *rewrite<- t* Rewrites subterm v of claim to u provided $t : u{=}v$.
- *subst* Eliminates variables for which an assumption $x = t$ exists.
- *congruence* Tries to solve the goal by rewriting with equations that appear as assumptions.

Inductive Tactics

- *destruct t* Proves claim with a match on t. Creates subgoals for clauses.
- *fix x n* Proves claim with a recursive function x where recusion decreases n-th argument.
- *discriminate t* Proves claim if equation proved by t is contradictory because of constructor disjointness.
- *injection t* Weakens claim by equational premises that follow by constructor injectivity from the equation proved by t.
- *induction t* Applies induction principle of the inductive type of t.

Automation Tactics

- *tauto* Solves every goal that they can be solved with intros, reflexivity, and the introduction and elimination rules for implication, conjunction, and disjunction. Fails if it cannot solve a goal.
- *auto* Both stronger and weaker than *tauto*. Never fails, leaves goal unchanged if it cannot solve it.

- *omega* Can prove claims that involve *nat*. Must be loaded with the command *Require Import Omega*.

Tacticals

Compose tactics into more powerful tactics.

- *s*;*t* Applies tactic s, then applies tactic *t* to every subgoal created by *s*.
- *s* || *t* Applies tactic s. If application of *s* fails, tactic *t* is applied.
- *repeat t* Applies tactic *t* until it either fails or leaves goal unchanged.
- *try t* Applies tactic *t*. If *t* fails, *try t* leaves goal unchanged and succeeds.

A.3 Predefined Variables

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

Inductive False : Prop := .

Definition not (A : Prop) := A -> False.
Notation "~ x" := (not x) : type_scope.

Inductive and (A B : Prop) : Prop := conj : A -> B -> and A B.
Notation "A /\ B" := (and A B) : type_scope.
Implicit Arguments conj [A B].

Inductive or (A B : Prop) : Prop := or_introl : A -> or A B  | or_intror : B -> or A B.
Notation "A \/ B" := (or A B) : type_scope.
Implicit Arguments or_introl [A].
Implicit Arguments or_intror [B].

Definition iff (A B : Prop) := (A -> B) /\ (B -> A).
Notation "A <-> B" := (iff A B) : type_scope.

Inductive ex (A:Type) (P:A -> Prop) : Prop :=  ex_intro : forall x:A, P x -> ex A P.
Notation "'exists' x , p" := (ex (fun x => p))
  (at level 200, x ident, right associativity) : type_scope.
Notation "'exists' x : t , p" := (ex (fun x:t => p))
  (at level 200, x ident, right associativity) : type_scope.
Implicit Arguments ex [A].
Implicit Arguments ex_intro [A].

Definition all {A:Type} (P:A -> Prop) := forall x:A, P x.

Inductive eq (A:Type) (x:A) : A -> Prop := refl_equal : eq A x x.
Notation "x = y" := (@eq _ x y) : type_scope.
Notation "x <> y" := (~ x = y) : type_scope.
Implicit Arguments eq [A].
Implicit Arguments refl_equal [A].

Lemma sym_eq {A : Type} {x y : A} :  x = y -> y = x.
Lemma f_equal {A B : Type} (f : A -> B) {x y : A} :  x =y -> f x = f y.
```