Introduction to Computational Logic

Lecture Notes SS 2014

July 16, 2014

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

Copyright © 2014 by Gert Smolka and Chad E. Brown, All Rights Reserved

Contents

Introduction 1

1 Types and Functions 3

- 1.1 Booleans 3
- 1.2 Cascaded Functions 6
- 1.3 Natural Numbers 8
- 1.4 Structural Induction and Rewriting 10
- 1.5 More on Rewriting 12
- 1.6 Recursive Abstractions 14
- 1.7 Defined Notations 15
- 1.8 Standard Library 16
- 1.9 Pairs and Implicit Arguments 18
- 1.10 Lists 21
- 1.11 Quantified Inductive Hypotheses 24
- 1.12 Iteration as Polymorphic Higher-Order Function 25
- 1.13 Options and Finite Types 27
- 1.14 More about Functions 29
- 1.15 Discussion and Remarks 31

2 Propositions and Proofs 33

- 2.1 Logical Connectives and Quantifiers 33
- 2.2 Implication and Universal Quantification 34
- 2.3 Predicates 35
- 2.4 The Apply Tactic 36
- 2.5 Leibniz Characterization of Equality 37
- 2.6 Propositions are Types 37
- 2.7 Falsity and Negation 38
- 2.8 Conjunction and Disjunction 40
- 2.9 Equivalence and Rewriting 41
- 2.10 Automation Tactics 44
- 2.11 Existential Quantification 44
- 2.12 Basic Proof Rules 46
- 2.13 Proof Rules as Lemmas 48

2.14 Inductive Propositions . . . 49
2.15 An Observation . . . 51
2.16 Excluded Middle . . . 52
2.17 Discussion and Remarks . . . 54
2.18 Tactics Summary . . . 55

3 Definitional Equality and Propositional Equality 57
3.1 Conversion Principle . . . 57
3.2 Disjointness and Injectivity of Constructors . . . 61
3.3 Leibniz Equality . . . 63
3.4 By Name Specification of Implicit Arguments . . . 66
3.5 Local Definitions . . . 66
3.6 Proof of nat ≠ bool . . . 67
3.7 Cantor's Theorem . . . 68
3.8 Kaminski's Equation . . . 70
3.9 Boolean Equality Tests . . . 71

4 Induction and Recursion 73
4.1 Induction Lemmas . . . 73
4.2 Primitive Recursion . . . 75
4.3 Size Induction . . . 77
4.4 Equational Specification of Functions . . . 78

5 Truth Value Semantics and Elim Restriction 81
5.1 Truth Value Semantics . . . 81
5.2 Elim Restriction . . . 83
5.3 Propositional Extensionality Entails Proof Irrelevance . . . 84
5.4 A Simpler Proof . . . 86

6 Sum and Sigma Types 87
6.1 Boolean Sums and Certifying Tests . . . 87
6.2 Inhabitation and Decidability . . . 89
6.3 Writing Certifying Tests . . . 90
6.4 Definitions and Lemmas . . . 94
6.5 Decidable Predicates . . . 95
6.6 Sigma Types . . . 96
6.7 Strong Truth Value Semantics . . . 98

7 Inductive Predicates 101
7.1 Nonparametric Arguments and Linearization . . . 101
7.2 Even . . . 103
7.3 Less or Equal . . . 106

7.4 Equality . . . 108
7.5 Exceptions to the Elim Restriction . . . 109
7.6 Safe and Nonuniform Parameters . . . 110
7.7 Constructive Choice for Nat . . . 112
7.8 Technical Summary . . . 114
7.9 Induction Lemmas . . . 115

8 Lists **119**
8.1 Constructors and Notations . . . 120
8.2 Recursion and Induction . . . 121
8.3 Membership . . . 124
8.4 Inclusion and Equivalence . . . 125
8.5 Disjointness . . . 128
8.6 Decidability . . . 128
8.7 Filtering . . . 131
8.8 Element Removal . . . 133
8.9 Cardinality . . . 133
8.10 Duplicate-Freeness . . . 135
8.11 Power Lists . . . 137

9 Syntactic Unification **139**
9.1 Terms, Substitutions, and Unifiers . . . 139
9.2 Solved Equation Lists . . . 142
9.3 Unification Rules . . . 144
9.4 Presolved Equation Lists . . . 146
9.5 Unification Algorithm . . . 146
9.6 Alternative Representations . . . 149
9.7 Notes . . . 151

10 Propositional Entailment **153**
10.1 Propositional Formulas . . . 153
10.2 Structural Properties of Entailment Relations . . . 154
10.3 Logical Properties of Entailment Relations . . . 156
10.4 Variables and Substitutions . . . 157
10.5 Natural Deduction System . . . 158
10.6 Classical Natural Deduction . . . 163
10.7 Glivenko's Theorem . . . 165
10.8 Hilbert System . . . 168
10.9 Intermediate Logics . . . 170
10.10 Remarks . . . 171

11 Classical Tableau Method 173
11.1 Boolean Evaluation and Satisfiability 173
11.2 Validity and Boolean Entailment 175
11.3 Signed Formulas and Clauses 175
11.4 Solved Clauses 176
11.5 Tableau Method 177
11.6 DNF Procedure 179
11.7 Recursion Trees 181
11.8 Assisted Decider for Satisfiability 182
11.9 Main Results 182
11.10 Refutation Lemma 183

12 Intuitionistic Gentzen System 185
12.1 Gentzen System GS 185
12.2 Completeness of GS 188
12.3 Decidability 190
12.4 Finite Closure Iteration 192
12.5 Realization in Coq 193
12.6 Notes 195

Bibliography 197

Introduction

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

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

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

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

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

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

Dedication

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

Contents

1 Types and Functions

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

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

1.1 Booleans

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

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

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

We define a negation function *negb*.

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

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

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

We can evaluate terms with the command *Compute*.

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

We are now ready for our first proof with Coq.

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

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

Our second proof shows that double negation is identity.

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

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

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

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

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

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

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

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

We define a boolean conjunction function *andb*.

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

We prove that boolean conjunction is commutative.

```
Lemma andb_com x y :
  andb x y = andb y x.
Proof.
  destruct x.
  - destruct y ; reflexivity.
  - destruct y ; reflexivity.
Qed.
```

The proof can be written more succinctly as

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

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

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

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

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

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

b) Prove that disjunction is commutative.

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

1.2 Cascaded Functions

When we look at the type of *andb*

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

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

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

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

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

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

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

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

Given the above explanations, the term

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

1.3 Natural Numbers

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

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

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

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

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

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

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

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

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

$$\begin{aligned} O + y &= y \\ Sx + y &= S(x + y) \end{aligned}$$

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

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

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

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

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

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

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

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

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

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

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

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

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

$$\begin{aligned} O \cdot y &= O \\ Sx \cdot y &= y + x \cdot y \end{aligned}$$

and use the addition function *plus*.

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

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

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

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

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

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

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

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

1.4 Structural Induction and Rewriting

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

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

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

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

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

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

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

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

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

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

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

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

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

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

$\dfrac{x : nat}{plus\ x\ O = x}$	$\dfrac{}{O = O}$	$\dfrac{x : nat \quad IHx : plus\ x\ O = x}{S(plus\ x\ O) = Sx}$	$\dfrac{x : nat \quad IHx : plus\ x\ O = x}{Sx = Sx}$
induction x ; simpl	reflexivity	rewrite IHx	reflexivity

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

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

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

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

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

Note that the lemmas are applied with the rewrite tactic.

Next we prove that addition is associative.

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

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

1.5 More on Rewriting

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

```
Require Import Omega.

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

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

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

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

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

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

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

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

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

Exercise 1.5.2 Prove the following lemmas.

```
Lemma mult_O (x : nat) :
  mult x O = O.
Lemma mult_S (x y : nat) :
  mult x (S y) = plus (mult x y) x.
Lemma mult_com (x y : nat) :
  mult x y = mult y x.
Lemma mult_dist (x y z: nat) :
  mult (plus x y) z = plus (mult x z) (mult y z).
Lemma mult_asso (x y z: nat) :
  mult (mult x y) z = mult x (mult y z).
```

1.6 Recursive Abstractions

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

```
Print plus.

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

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

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

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

```
Compute plus O.
% fun y : nat ⇒ y

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

Compute fun x ⇒ plus (S x).

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

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

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

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

1.7 Defined Notations

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

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

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

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

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

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

```
Set Printing All.
```

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

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

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

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

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

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

1.8 Standard Library

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

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

```
Set Printing All.
```

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

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

```
Locate "*".
```

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

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

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

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

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

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

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

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

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

1.9 Pairs and Implicit Arguments

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

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

which fixes two constructors

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

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

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

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

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

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

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

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

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

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

```
Set Printing All.

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

Unset Printing All.
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Here is a function swapping the components of a pair:

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

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

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

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

will succeed with the type $nat * nat$.

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

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

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

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

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

1.10 Lists

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

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

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

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

The definition provides three constructors:

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

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

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

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

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

We define an infix notation for *cons*.

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

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

We also define the bracket notation for lists.

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

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

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

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

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

$$
\begin{aligned}
|nil| &= 0 \\
|x :: A| &= 1 + |A|
\end{aligned}
$$

$$
\begin{aligned}
nil + B &= B \\
x :: A + B &= x :: (A + B)
\end{aligned}
$$

$$
\begin{aligned}
rev\ nil &= nil \\
rev\ (x :: A) &= rev\ A + [x]
\end{aligned}
$$

The Coq definitions are now straightforward.

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

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

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

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

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

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

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

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

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

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

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

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

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

Exercise 1.10.1 Prove the following lemmas.

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

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

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

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

1.11 Quantified Inductive Hypotheses

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

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

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

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

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

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

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

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

Exercise 1.11.1 Prove the following lemma.

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

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

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

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

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

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

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

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

1.12 Iteration as Polymorphic Higher-Order Function

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

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

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

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

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

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

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

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

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

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

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

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

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

Exercise 1.12.1 Prove the following lemma:

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

Exercise 1.12.2 Prove the following lemma:

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

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

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

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

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

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

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

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

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

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

1.13 Options and Finite Types

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

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

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

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

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

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

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

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

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

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

We can now define a function *fin* : *nat* → *Type* such that *fin n* is a type with *n* elements.

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

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

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

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

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

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

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

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

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

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

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

1.14 More about Functions

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

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

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

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

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

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

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

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

their types:

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

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

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

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

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

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

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

1.15 Discussion and Remarks

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

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

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

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

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

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

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

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

Coq Summary

Type and Value Constructors from the Standard Library

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

Defined Functions from the Standard Library

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

Term Variants

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

Definitional Commands

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

Tactics

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

Notational Commands

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

Module Commands

Require Import, *Import*.

Query Commands

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

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

2 Propositions and Proofs

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

2.1 Logical Connectives and Quantifiers

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

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

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

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

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

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

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

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

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

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

2.2 Implication and Universal Quantification

Example: Symmetry of Equality

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

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

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

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

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

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

Exercise 2.2.1 Prove the following goal.

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

Example: Modus Ponens

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

```
Goal ∀ X Y : Prop, X → (X → Y) → Y.
Proof. intros X Y x A. exact (A x). Qed.
```

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

$$\begin{array}{l} X : Prop \\ Y : Prop \\ x : X \\ A : X \to Y \\ \hline \quad Y \end{array}$$

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

Example: Transitivity of Implication

```
Goal ∀ X Y Z : Prop, (X → Y) → (Y → Z) → X → Z.
Proof. intros X Y Z A B x. exact (B (A x)). Qed.
```

Exercise 2.2.2 Prove that equality is transitive.

2.3 Predicates

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

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

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

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

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

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

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

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

2.4 The Apply Tactic

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

```
Goal ∀ X Y Z : Prop, (X → Y) → (Y → Z) → X → Z.
Proof. intros X Y Z A B x. apply B. apply A. exact x. Qed.
```

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

```
Goal ∀ p q : nat → Prop, p 7 → (∀ x, p x → q x) → q 7.
Proof. intros p q A B. apply B. exact A. Qed.
```

Step through the proofs with Coq to understand.

Exercise 2.4.1 Prove the following goals.

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

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

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

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

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

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

Hint: Use case analysis and induction.

2.5 Leibniz Characterization of Equality

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

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

```
Proof. intros X x y A. apply (A (fun z ⇒ x=z)). reflexivity. Qed.
```

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

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

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

Exercise 2.5.1 Prove the following goals.

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

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

2.6 Propositions are Types

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

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

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

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

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

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

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

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

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

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

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

2.7 Falsity and Negation

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

```
Goal ⊥ → 2=3.
```

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

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

```
Goal ∀ X : Prop, X → ¬¬ X.
```

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

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

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

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

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

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

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

Exercise 2.7.1 Prove the following goals.

```
Goal ∀ X : Prop, ¬¬¬ X → ¬ X.
```

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

Exercise 2.7.2 Prove the following goals.

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

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

Exercise 2.7.3 Prove the following proposition in Coq using only the tactic *exact*.

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

2.8 Conjunction and Disjunction

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

```
Goal ∀ X Y : Prop, X ∧ Y → Y ∧ X.
Proof.
  intros X Y A. destruct A as [x y]. split.
  - exact y.
  - exact x.
Qed.
```

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

```
Goal ∀ X Y : Prop, X ∨ Y → Y ∨ X.
Proof.
  intros X Y A. destruct A as [x|y].
  - right. exact x.
  - left. exact y.
Qed.
```

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

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

```
Goal ∀ X Y : Prop, X ∧ Y → Y ∧ X.
Proof.
  intros X Y [x y]. split.
  - exact y.
  - exact x.
Qed.
```

```
Goal ∀ X Y : Prop, X ∨ Y → Y ∨ X.
Proof.
  intros X Y [x|y].
  - right. exact x.
  - left. exact y.
Qed.
```

Nesting of destructuring patterns is possible:

```
Goal ∀ X Y Z : Prop,
X ∨ (Y ∧ Z) → (X ∨ Y) ∧ (X ∨ Z).
Proof.
  intros X Y Z [x|[y z]].
  - split; left; exact x.
```

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

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

Exercise 2.8.1 Prove the following goals.

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

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

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

Exercise 2.8.2 Prove the following goals.

```
Goal ∀ X : Prop, ¬¬ (X ∨ ¬ X).

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

2.9 Equivalence and Rewriting

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

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

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

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

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

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

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

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

We leave the proof of this goal as an exercise.

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

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

Proof.
 intros X Y Z.
 setoid_rewrite deMorgan.
 apply and_com.
Qed.

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

Proof.
 intros X p q A.
 setoid_rewrite ← deMorgan.
 exact A.
Qed.
```

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

```
Goal ∀ X : Prop, X ↔ X.

Proof. reflexivity. Qed.
```

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

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

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

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

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

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

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

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

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

Exercise 2.9.4 Prove the following goals.

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

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

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

2.10 Automation Tactics

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

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

```
Goal ∀ X Y : Prop, X ∧ Y → Y ∧ X.
Proof. intros X Y [x y].  split  ;  assumption. Qed.
```

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

```
Goal ∀ (X : Type) (p : list X → Prop) (xs : list X),
p nil → (∀ x xs, p xs → p (cons x xs)) → p xs.
Proof. induction xs ; auto. Qed.
```

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

```
Goal ∀ X : Prop, ¬ (X ↔ ¬ X).
Proof. τto. Qed.
```

2.11 Existential Quantification

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

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

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

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

Run the proof scripts with Coq to understand.

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

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

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

The proof of the diagonal law is not difficult.

```
Lemma circuit (X : Prop) : ¬ (X ↔ ¬ X).

Proof. τto. Qed.

Goal diagonal.

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

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

```
Goal diagonal.

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

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

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

Proof.
  split.
  - intros A. ∃ (fun z ⇒ x = z). auto.
  - intros [p [A B]] C. apply B. rewrite ← C. apply A.
Qed.
```

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

Exercise 2.11.1 Prove the De Morgan law for existential quantification.

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

Exercise 2.11.2 Prove the exchange rule for existential quantifications.

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

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

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

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

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

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

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

Exercise 2.11.5 Prove the following goal.

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

Exercise 2.11.6

a) Prove the following goal.

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

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

2.12 Basic Proof Rules

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

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

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

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

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

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

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

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

Figure 2.1: Basic proof rules

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

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

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

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

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

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

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

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

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

$$\neg s := s \rightarrow \bot$$

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

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

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

2.13 Proof Rules as Lemmas

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

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

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

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

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

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

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

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

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

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

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

2.14 Inductive Propositions

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

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

```
Inductive ⊤ : Prop :=
| I : ⊤.
Inductive ⊥ : Prop := .
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
Definition not (X : Prop) : Prop := X → ⊥.
```

```
Definition iff (X Y : Prop) : Prop := (X → Y) ∧ (Y → X).
```

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

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

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

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

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

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

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

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

2.15 An Observation

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

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

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

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

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

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

Proof. intros x y Z. auto. Qed.

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

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

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

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

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

2.16 Excluded Middle

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

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

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

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

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

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

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

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

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

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

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

Goal $\forall\, X\, Y : \textbf{Prop},$
$\neg\neg\,((X \to Y) \leftrightarrow (\neg\, Y \to \neg\, X)).$

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

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

a) $\forall\, X : \textbf{Prop},\ \neg\,(X \lor \neg\, X)$

b) $\exists\, X : \textbf{Prop},\ \neg\,(X \lor \neg\, X)$

c) $\forall\, P : \textbf{Prop},\ \exists\, f : \textbf{Prop} \to \textbf{Prop},\ \forall\, X\, Y : \textbf{Prop},$
$(X \land P \to Y) \leftrightarrow (X \to f\, Y)$

d) $\forall\, P : \textbf{Prop},\ \exists\, f : \textbf{Prop} \to \textbf{Prop},\ \forall\, X\, Y : \textbf{Prop},$
$(X \to Y \land P) \leftrightarrow (f\, X \to Y)$

2.17 Discussion and Remarks

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

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

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

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

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

2.18 Tactics Summary

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

3 Definitional Equality and Propositional Equality

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

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

3.1 Conversion Principle

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

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

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

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

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

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

```
Goal ¬¬⊤.
Proof.                                  ¬¬True
  change (¬⊤ → ⊥).                ¬True → False
  change (¬(⊤ → ⊥)).              ¬(True → False)
  change (¬¬⊤).                       ¬¬True
  hnf.                                  ¬True → False
  change (¬¬⊤).                       ¬¬True
  cbv.                                  (True → False) → False
  change (¬¬⊤).                       ¬¬True
  simpl.                                ¬¬True
  pattern ⊤.                          (λp : Prop.¬¬p) True
  pattern not at 2.                     (λf : Prop→Prop.(λp : Prop.¬f p) True) not
  hnf.                                  ¬True → False
  exact (fun f ⇒ f I).
  Show Proof.
Qed.
```

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

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

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

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

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

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

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

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

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

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

```
Goal demo plus.
```

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

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

Next we demo alpha conversion.

```
Goal demo (fun x : nat ⇒ x).

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

```
  apply demoI.
Qed.
```

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

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

Here is a conversion demo involving match and fix conversions.

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

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

Finally, we demonstrate eta conversion.

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

  Goal demo f.
```

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

```
End Demo.
```

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

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

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

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

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

(a) `Goal plus 1 = S.`

(b) `Goal (fun y ⇒ 3+y) = plus (4−1).`

(c) `Goal S = fun x ⇒ x + 1.`

(d) `Goal S = fun x ⇒ 1 + x.`

(e) `Goal S = fun x ⇒ 2+x+1−2.`

(f) `Goal plus 3 = fun x ⇒ 5+x−2.`

(g) `Goal mult 2 = fun x ⇒ x + (x + 0).`

(h) `Goal S = fun x ⇒ S (pred (S x)).`

(i) `Goal minus = fun x y ⇒ x−y.`

3.2 Disjointness and Injectivity of Constructors

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Exercise 3.2.3 Prove the following goals.

(a) `Goal ∀ x, negb x ≠ x.`

(b) `Goal ∀ x, S x ≠ x.`

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

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

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

Exercise 3.2.4 Prove the following goal.

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

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

Exercise 3.2.5 Prove $\mathit{False} \neq \mathit{True}$.

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

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

3.3 Leibniz Equality

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

$$x = y \;\leftrightarrow\; \forall p : X \to \mathit{Prop}.\; px \leftrightarrow py$$

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

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

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

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

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

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

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

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

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

Proof. hnf. auto. Qed.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

3.4 By Name Specification of Implicit Arguments

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

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

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

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

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

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

3.5 Local Definitions

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

let $x : t := s$ in u

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

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

Here are examples.

```
Compute let x := 2 in x + x.
% 4 : nat
Compute let x := 2 in let x := x + x in x.
% 4 : nat
Compute let f := plus 3 in f 7.
% 10 : nat
```

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

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

will type check while the corresponding beta redex will not.

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

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

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

3.6 Proof of nat ≠ bool

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

```
Goal bool ≠ nat.
```

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

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

Exercise 3.6.1 Prove the following goals.

(a) **Goal** bool ≠ option bool.
(b) **Goal** option bool ≠ prod bool bool.
(c) **Goal** bool ≠ ⊥.

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

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

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

3.7 Cantor's Theorem

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

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

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

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

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

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

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

Exercise 3.7.1 Prove the following goals.

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

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

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

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

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

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

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

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

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

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

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

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

3.8 Kaminski's Equation

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

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

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

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

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

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

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

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

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

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

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

3.9 Boolean Equality Tests

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

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

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

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

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

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

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

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

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

Coq Summary

Conversion Tactics

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

Constructor Tactics

discriminate, *injection*, *congruence*.

Other Tactics

pose, *assert*.

Tacticals

try

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

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

New Features of the Term Language

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

Sections

See Section 3.1.

4 Induction and Recursion

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

4.1 Induction Lemmas

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

```
Check nat_ind.
```

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

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

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

```
Goal ∀ n, n + 0 = n.

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

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

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

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

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

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

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

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

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

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

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

Exercise 4.1.2 We consider an induction lemma for list types.

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

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

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

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

4.2 Primitive Recursion

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

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

$$\mathit{prec}\ x\ f\ 0 = x$$
$$\mathit{prec}\ x\ f\ (S\,n) = f\ n\ (\mathit{prec}\ x\ f\ n)$$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Exercise 4.2.1 Prove *prec* = *nat_rect*.

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

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

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

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

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

4.3 Size Induction

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

```
Lemma size_induction X (f : X → nat) (p : X → Prop) :
  (∀ x, (∀ y, f y < f x → p y) → p x) →
  ∀ x, p x.

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

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

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

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

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

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

b) Prove the lemma using natural induction.

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

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

4.4 Equational Specification of Functions

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

```
Definition addition (f : nat → nat → nat) : Prop :=
  ∀ x y,
    f x 0 = x ∧
    f x (S y) = f (S x) y.

Lemma addition_existence :
  addition plus.

Proof. intros x y. omega. Qed.

Lemma addition_uniqueness f g :
  addition f → addition g → ∀ x y, f x y = g x y.

Proof.
  intros A B x y. revert x. induction y ; intros x.
  - destruct (A x 0) as [A' _]. destruct (B x 0) as [B' _]. congruence.
  - destruct (A x y) as [_ A']. destruct (B x y) as [_ B']. specialize (IHy (S x)). congruence.
Qed.
```

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

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

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

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

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

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

```
Definition ack : nat → nat → nat :=
  fix f m := match m with
             | O ⇒ S
             | S m' ⇒ fix g n := match n with
                                 | O ⇒ f m' 1
                                 | S n' ⇒ f m' (g n')
                               end
           end.
```

Note that *ack* is defined as a recursive function that yields a recursive function when give an argument greater than 0. Each of the two recursions is structural on its argument. The correctness proof for *ack* is straightforward.

```
Goal ackermann ack.
Proof. unfold ackermann. auto. Qed.
```

We can also show that any two functions satisfying the specification *ackermann* agree on all arguments.

```
Goal ∀ f g x y,  ackermann f → ackermann g → f x y = g x y.
Proof.
  intros f g x y A B. revert y. induction x ; intros y.
  - destruct (A 0 y) as [C _]. destruct (B 0 y) as [D _]. congruence.
  - induction y.
    + destruct (A x 0) as [_ [C _]]. destruct (B x 0) as [_ [D _]]. congruence.
    + destruct (A x y) as [_ [_ C]]. destruct (B x y) as [_ [_ D]]. congruence.
Qed.
```

Exercise 4.4.1 Write an equational specification for multiplication and prove that Coq's multiplication satisfies the specification. Also prove that two functions agree on all arguments if they satisfy the specification. Do the same for subtraction.

Exercise 4.4.2 Write an Ackermann function using *prec* rather than fix and match. Prove that your function satisfies the specification *ackermann*.

Exercise 4.4.3 We specify primitive recursion as follows.

```
Definition primitive_recursion
(r : ∀ p : nat → Type, p 0 → (∀ n, p n → p (S n)) → ∀ n, p n)
: Prop :=
  ∀ p x f n,
    let r := r p x f in
    r 0 = x  ∧
    r  (S n) = f n (r n).
```

Note that a local declaration with let is used to write the specifying equations in compact form. Show that *prec* satisfies the specification. Also prove that two functions agree on all arguments if they satisfy the specification.

Exercise 4.4.4 Give an equational specification for *nat_iter*. Prove that *nat_iter* satisfies the specification and that the specification is unique.

Coq Summary

New Features of the Term Language

- Matches can be specified with a return type function. See Section 4.1, definition of *nat_ind*.

5 Truth Value Semantics and Elim Restriction

Coq's type theory is designed such that the truth value semantics of propositions commonly used in Mathematics can be consistently assumed. This design comes at the price of the elim restriction, which restricts matches on proofs such that proofs must be returned. The elim restriction severely restricts the computational use of proofs.

5.1 Truth Value Semantics

In Mathematics one assumes that a proposition is either true or false. More specifically, one assumes that every proposition denotes a truth value, which is either true or false. With the boolean definition of disjunction it then follows that for any proposition p the disjunction $p \vee \neg p$ is true.

Given the fact that propositions denote truth values, we could consider two propositions equal if they denote the same truth value. This assumption is made in boolean logic as well as in Church-Henkin simple type theory.

We formulate the mathematical assumption that a proposition is either true or false as a proposition in Coq:

```
Definition TVS : Prop := ∀ X : Prop, X=⊤ ∨ X=⊥.
```

We can now ask whether Coq can prove *TVS* or $\neg$*TVS*. It turns out that Coq can prove neither of the two. That Coq cannot prove *TVS* seems intuitively clear since there is nothing in the basic proof rules that would give us a proof of *TVS*. On the other hand, that Coq cannot prove $\neg$*TVS* is not clear at all given the fact that propositions in Coq are obtained as types.

We call a proposition p **consistent** in Coq if Coq cannot prove $\neg p$. Moreover, we call a proposition p **independent** in Coq if Coq can prove neither p nor $\neg p$. Note that every independent proposition is consistent, but not vice versa (e.g. *True* is consistent but not independent).

Above we have stated that *TVS* is independent in Coq. The consistency of *TVS* in Coq does not come for free. In fact, the design of Coq's type theory has been carefully arranged so that *TVS* is consistent. To obtain the consistency of *TVS*,

Coq imposes a severe typing restriction known as the *elim restriction.* To prepare the discussion of the elim restriction, we first consider some conseqences of *TVS*.

First we show that *TVS* implies *XM* (excluded middle). This follows from the fact that *True* and *False* satisfy *XM* (i.e., $True \lor \neg True$ and $False \lor \neg False$ are provable).

```
Goal TVS → XM.
```

```
Proof. intros A X. destruct (A X) as [B|B] ; rewrite B ; auto. Qed.
```

Another important consequence of *TVS* is **proof irrelevance**.

```
Definition PI : Prop := ∀ (X : Prop) (A B : X), A=B.
```

Proof irrelevance says that a proposition has at most one proof. Here is a proof that truth value semantics implies proof irrelevance.

```
Goal TVS → PI.
```

```
Proof.
  intros A X B C. destruct (A X) ; subst X.
  - destruct B, C. reflexivity.
  - contradiction B.
Qed.
```

The proof exploits that the proposition *True* has exactly one proof, a fact following from the inductive definition of *True.* The script uses the tactic *subst*, which eliminates a variable x if there is an assumption $x = s$ such that x does not occur in s.

A third consequence of *TVS* is **propositional extensionality**.

```
Definition PE : Prop := ∀ X Y : Prop, (X ↔ Y) → X=Y.
```

Propositional extensionality says that two propositions are equal if they are equivalent.

```
Goal TVS → PE.
```

```
Proof. intros A X Y B. destruct (A X), (A Y) ; subst X Y ; τto. Qed.
```

Finally, we show that excluded middle and propositional extensionality together imply truth value semantics.

```
Goal XM → PE → TVS.
```

```
Proof.
  intros xm pe X. destruct (xm X) as [A|A].
  - left. apply pe. τto.
  - right. apply pe. τto.
Qed.
```

We now know that *TVS* and $XM \land PE$ are equivalent.

Exercise 5.1.1 Make sure you can prove $TVS \leftrightarrow XM \wedge PE$

Exercise 5.1.2 Prove $TVS \rightarrow PE$ without using the tactic *subst*. Use the tactic *rewrite* instead.

5.2 Elim Restriction

If we have a surjective function $f : X \rightarrow bool$, then the type X must contain at least two elements.

```
Goal ∀ X (f : X → bool), surjective f → ∃ x y : X, x ≠ y.
Proof.
  intros X f A. destruct (A true) as [x B]. destruct (A false) as [y C].
  ∃ x, y. congruence.
Qed.
```

Now consider the inductive proposition

```
Inductive bp : Prop := P1 : bp | P2 : bp.
```

which by definition has two proofs *P1* and *P2*. Given the match for *bp*, it is easy to construct a surjective function $bp \rightarrow bool$, it seems. However, Coq rejects the following match on $x : bp$:

```
Check fun x : bp ⇒ match x with P1 ⇒ true | P2 ⇒ false end.
% Error : Incorrect elimination of "x" ...
```

The reason is the so-called **elim restriction**: A match on a proof (i.e., on an element of a proposition) is only allowed if the match returns a proof. The above match does not return a proof and hence it is rejected by the elim restriction. One important reason for the elim restriction is that it is needed so that truth value semantics is consistent in Coq. Without the elim restriction, we get a surjective function from *bp* to *bool*, which entails that the proposition *bp* has two different elements. This however contradicts proof irrelevance, which says that no proposition has more than one proof. Since *TVS* entails *PI*, not having the elim restriction would mean that truth value semantics is inconsistent in Coq.

There are a few exceptions to the elim restriction, all of them being consistent with proof irrelevance. For instance, there is no restriction on the matches for *True* and *False*. The remaining exceptions will be discussed in Section 7.5.

We give another example illustrating the elim restriction. Consider the following lemma, which establishes the existence of a so-called *Skolem function* for total predicates $p : X \rightarrow Y \rightarrow Prop$ where Y is a proposition.

```
Lemma Prop_Skolem (X : Type) (Y : Prop) (p : X → Y → Prop) :
  (∀ x, ∃ y, p x y) → ∃ f, ∀ x, p x (f x).
```

```
Proof.
 intros A.
 ∃ (fun x ⇒ let (y,_) := A x in y).
 intros x.
 destruct (A x) as [y B].
 exact B.
Qed.
```

The let notation in the definition of the Skolem function is notational sugar for the one-constructor match

```
match A x return Y with ex_intro y _ ⇒ y end
```

This match on the proof $A\ x$ is legal since it returns a proof y. If we generalize the lemma to $Y : \mathit{Type}$, the proof script fails since the match now violates the elim restriction.

We will speak of proper types and proper values. A **proper type** is a type that is not a proposition, and a **proper value** is an element of a proper type. Thus a type is not proper if and only if it is a proposition, and a value is not proper if and only if it is a proof. Examples of proper types are *nat*, *list nat*, and $\mathit{nat} \to \mathit{nat}$. Example of proper values are 5, *cons*, and $\lambda x : \mathit{nat}. x$.

Exercise 5.2.1 Prove that the proposition $\forall X : \mathit{Type}.\ X = \mathit{True} \vee X = \mathit{False}$ is inconsistent in Coq. Note that *TVS* is a weaker statement where X is restricted to propositional types. Find out where your proof breaks if you apply it to *TVS*.

5.3 Propositional Extensionality Entails Proof Irrelevance

It turns out that propositional extensionality entails proof irrelevance. This is a surprising result with a very interesting proof. The proof rests on a fixpoint theorem that given a surjective function $X \to X \to Y$ states that every function $Y \to Y$ has a fixpoint.

```
Lemma sur_fixpoint X Y (f : X → X → Y) (g : Y → Y) :
  surjective f → ∃ y, g y = y.
Proof.
  intros A.
  pose (h x := g (f x x)).
  destruct (A h) as [x B].
  ∃ (h x). unfold h at 2. rewrite ← B. reflexivity.
Qed.
```

The proof of the theorem should remind you of Cantor's theorem. In fact, we can obtain Cantor's theorem as a corollary of the surjective fixpoint theorem by specializing to $Y := \mathit{Prop}$ and $g := \mathit{not}$.

We now assume *PE* and prove *PI*. It suffices to prove $P1 = P2$ for the two constructors of *bp* since given two proofs x and y of a proposition X we can obtain a function f such that $f\,P1 = x$ and $f\,P2 = y$ using the match for *bp*. For $P1 = P2$ it suffices to show that the "negation" function mapping *P1* to *P2* and *P2* to *P1* has a fixpoint. This we obtain with the surjective fixpoint theorem. To do so, we need a surjective function $bp \to bp \to bp$. Such a function is easy to obtain if we have the equation $(bp \to bp) = bp$. This finishes the proof since the equation is a straightforward consequence of *PE*.

```
Goal PE → PI.
Proof.
  intros pe.
  cut (P1=P2).
  { intros A X B C.
    change (B = match P1 with P1 ⇒ C | P2 ⇒ B end).
    rewrite A. reflexivity. }
  pose (neg x := match x with P1 ⇒ P2 | P2 ⇒ P1 end).
  cut (∃ P, neg P = P).
  { unfold neg. intros [[|] C].
    – symmetry. exact C.
    – exact C. }
  cut (∃ f : bp → bp → bp, surjective f).
  { intros [f A]. apply (sur_fixpoint (f:=f)). exact A. }
  cut (bp = (bp → bp)).
  { intros A. rewrite ← A. ∃ (fun x ⇒ x). intros x. ∃ x. reflexivity. }
  apply pe. split ; auto using P1.
Qed.
```

Note the use of the tactic *cut* to realize the backwards reasoning of the proof outline. In the last line the automation tactic *auto* is used with a suffix telling it to use the proof constructor *P1* : *bp*.

Exercise 5.3.1 Prove Cantor's theorem using the surjective fixpoint theorem *sur_fixpoint*.

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

Exercise 5.3.2 Two types are *isomorphic* if there are commuting functions back and forth.

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

Propositional univalence is the property that propositions are equal if they are isomorphic.

```
Definition PU : Prop := ∀ X Y : Prop, iso X Y → X = Y.
```

It turns out that propositional extensionality factors into propositional univalence and proof irrelevance, that is, $PE \leftrightarrow PU \land PI$. We have already shown $PE \to PI$. Prove $PE \to PU$ and $PU \to PI \to PE$ to establish the equivalence.

5.4 A Simpler Proof

There is a simpler proof of $PE \to PI$.[1] The idea is to use PE to reduce proving proof irrelevance of a general proposition to proof irrelevance for *True*. Since *True* has only one proof by definition, we are done. The Coq proof looks as follows.

```
Goal PE → PI.
intros D X E F.
assert (C: X=⊤) by (apply D; τto).
subst. destruct E, F. reflexivity.
Qed.
```

Exercise 5.4.1 Prove the following subgoals which together prove $PE \to PI$.

```
Goal PE → ∀ X:Prop, X → X = ⊤.
```

```
Goal (∀ X:Prop, X → X = ⊤) → P1 = P2.
```

Coq Summary

New Tactics

subst, *cut*.

Tactic *auto* with *using*

The tactic *auto* can be enhanced with lemmas and constructors specified with a *using* suffix. See the proof of the goal $PE \to PI$ in Section 5.3.

[1] This was pointed out by Jonas Oberhauser and Fabian Kunze during the teaching of Introduction to Computational Logic in 2014.

6 Sum and Sigma Types

Sum types and sigma types are non-propositional variants of disjunctions and existential quantifications. Since they are proper types, sum and sigma types are not subject to the elim restriction. The elements of sum and sigma types are computational values carrying a proof. With sum and sigma types we can write certifying functions whose results contain correctness proofs.

6.1 Boolean Sums and Certifying Tests

Boolean sums are disjunctions placed in *Type* rather than *Prop*. Coq's standard library defines boolean sums as follows.

```
Inductive sumbool (X Y : Prop) : Type :=
| left : X → sumbool X Y
| right : Y → sumbool X Y.

Arguments left {X} {Y} _.
Arguments right {X} {Y} _.

Notation "{ X } + { Y }" := (sumbool X Y).
```

Boolean sums are like disjunctions except for the crucial difference that they are proper types rather than propositions. Thus boolean sums are not subject to the elim restriction. We call the elements of boolean sums **decisions**. We can think of a decision as a proof-carrying boolean value, or as a proof of a disjunction on which we can freely match.

A **certifying test** is a function that yields a decision. Coq's library provides many certifying tests. For instance, there is a certifying test for the order on natural numbers:

$$le_dec : \forall x\, y : nat,\ \{x \le y\} + \{\neg(x \le y)\}$$

The type of *le_dec* tells us that *le_dec* is a function that takes two numbers x and y and returns a decision containing a proof of either $x \le y$ or $\neg x \le y$. With *le_dec* a minimum function can be written as follows:

```
Definition min (x y : nat) : nat :=
  if le_dec x y then x else y.
```

```
Compute min 7 3.
% 3 : nat
```

Note the use of the if-then-else notation in the definition of *min*. The if-then-else notation is available for all inductive types with two constructors and expands to a match. The definition of *min* expands as follows.

```
Set Printing All.
Print min.
  min = fun x y : nat ⇒ match le_dec x y with left _ ⇒ x | right _ ⇒ y end
Unset Printing All.
```

We prove the correctness of *min*.

```
Goal ∀ x y, (x ≤ y → min x y = x) ∧ (y ≤ x → min x y = y).
Proof.
  intros x y. split ; intros A.
  - unfold min. destruct (le_dec x y) as [B|B].
    + reflexivity.
    + omega.
  - unfold min. destruct (le_dec x y) as [B|B].
    + omega.
    + reflexivity.
Qed.
```

The proof can be shortened to a one-liner.

```
intros x y. split ; intros A ; unfold min ; destruct (le_dec x y) ; omega.
```

The Coq library *Compare_dec* offers many certifying tests for natural numbers. Here are a few.

$$
\begin{aligned}
\mathit{le_lt_dec} &: \forall xy : \mathit{nat},\ \{x \le y\} + \{y < x\} \\
\mathit{le_ge_dec} &: \forall xy : \mathit{nat},\ \{x \le y\} + \{x \ge y\} \\
\mathit{le_gt_dec} &: \forall xy : \mathit{nat},\ \{x \le y\} + \{x > y\} \\
\mathit{lt_eq_lt_dec} &: \forall xy : \mathit{nat},\ \{x < y\} + \{x = y\} + \{y < x\}
\end{aligned}
$$

The type of *lt_eq_lt_dec* needs explanation. Since boolean sums are proper types taking propositions as arguments, they cannot be nested. Coq solves the problem with an additional sum type *sumor* and a concomitant notation.

```
Set Printing All.
Check {⊤} + {⊥} + {⊥}.
% sumor (sumbool True False) False : Type
Unset Printing All.
```

The type *sumor* and the accompanying notation are defined as follows.

```
Inductive sumor (X : Type) (Y : Prop) : Type :=
| inleft : X → sumor X Y
| inright : Y → sumor X Y.

Notation "X + { Y }" := (sumor X Y).
```

Exercise 6.1.1 Prove the following goal.

```
Goal ∀ X Y : Prop, {X} + {Y} → X ∨ Y.
```

Explain why you cannot prove the other direction $\forall X\ Y : Prop,\ X \vee Y \to \{X\} + \{Y\}$.

Exercise 6.1.2 Prove the following goals.

```
Goal ∀ x y, if le_dec x y then x ≤ y else ¬ x ≤ y.

Goal ∀ x y, if le_dec x y then x ≤ y else x > y.
```

6.2 Inhabitation and Decidability

An **inhabitant** of a type is an element of a type. So saying that x is an inhabitant of a type X means the same as saying that x is a member of X, or that x is an element of X. We say that a type is **inhabited** if it has at least one inhabitant. So a type is inhabited if and only if it is nonempty. Coq's library comes with an inductive predicate for inhabitation.

```
Inductive inhabited (X : Type) : Prop :=
| inhabits : X → inhabited X.
```

A proposition is inhabited if and only if it is provable.

```
Goal ∀ X : Prop, inhabited X ↔ X.

Proof.
  split.
  - intros [A] ; exact A.
  - intros A. constructor. exact A.
Qed.
```

Note the use of the tactic *constructor*. Here it has the same effect as the command *apply inhabits*. In general, the tactic *constructor* tries to prove an inductive proposition by applying a constructor of the definition of the proposition. The tactic *constructor* is convenient since the name of the constructor needs not to be given.

We say that a proposition p is **decidable** if the sum $\{p\} + \{\neg p\}$ is inhabited. To have a concise notation for decidable propositions, we define the function

```
Definition dec (X : Prop) : Type := {X} + {¬ X}.
```

Note that *dec* is not a predicate. An element of *dec X* is a decision that gives us a proof of either X or $\neg X$. We call a member of *dec X* a **decision of X**.

The certifying test *le_dec* from the standard library tells us that all propositions of the form $x \le y$ are decidable.

```
Check le_dec : ∀ x y : nat, dec (x ≤ y).
```

We define a function that converts a decision to a boolean by forgetting the proof coming with the decision.

```
Definition dec2bool (X : Prop) (d : dec X) : bool :=
  if d then true else false.
Compute dec2bool (le_dec 2 3).
% true : bool
```

We now establish the decidability of *True*. To do so, we construct a decision of type *dec True*. This is easy since the constructor *I* is a proof of *True*.

```
Definition ⊤_dec : dec ⊤ := left I.
```

The decidability of *False* is also easy to establish.

```
Definition ⊥_dec : dec ⊥ := right (fun A ⇒ A).
```

In the next section we will show that implications, conjunctions, and disjunctions of decidable propositions are decidable.

Exercise 6.2.1 Prove the following goal.

```
Goal ∀ X : Type, X → inhabited X.
```

Note that $X \to \textit{inhabited}\ X$ is notation for the proposition $\forall x{:}X,\ \textit{inhabited}\ X$. Explain why you cannot prove that the type $\forall X : \textit{Type},\ \textit{inhabited}\ X \to X$ is inhabited.

Exercise 6.2.2 Prove $\forall X\ Y{:}\textit{Prop}.\ \ X \vee Y \leftrightarrow \textit{inhabited}\,(\{X\} + \{Y\})$.

Exercise 6.2.3 Prove $\forall X{:}\textit{Prop}.\ \ \textit{dec}\,X \to X \vee \neg X$.

6.3 Writing Certifying Tests

We show that implication preserves decidability of propositions.

```
Definition impl_dec (X Y : Prop) : dec X → dec Y → dec (X → Y).
  intros A [B|B].
  - left. auto.
  - destruct A as [A|A].
    + right. auto.
    + left. tauto.
Defined.
```

The definition of the function *impl_dec* should come as a surprise. This is the first time we construct a member of a type that is not a proposition with a script. Use the command *Print impl_dec* to see that the function constructed is in fact similar to what you would have written by hand. Note that the tactics *left* and *right* so far used for disjunctions also work for boolean sums. In fact, *left* and *right* will work for every inductive type with two constructors. We can compute with the certifying test *impl_dec*.

```
Check impl_dec (le_dec 3 2) ⊥_dec.
% dec(3 ≤ 2 → False)
```

```
Compute (dec2bool (impl_dec (le_dec 3 2) ⊥_dec)).
% true : bool
```

Here is a certifying equality test for *nat*.

```
Definition nat_eq_dec (x y : nat) : dec (x=y).
  revert y. induction x ; simpl ; intros [|y].
  - left. auto.
  - right. auto.
  - right. auto.
  - destruct (IHx y).
    + left. congruence.
    + right. congruence.
Defined.
```

```
Compute dec2bool (nat_eq_dec 3 3).
% true : bool
```

This is the first time we use the induction tactic to synthesize a function returning a proper value. When you print *nat_eq_dec*, you will see that the induction tactic realizes the necessary recursion with *nat_rect*, an automatically generated function providing primitive recursion for *nat*.

A more convenient way to obtain a certifying equality test for *nat* is using the automation tactic *decide equality*.

```
Goal ∀ x y : nat, dec (x=y).
Proof. unfold dec. decide equality. Qed.
```

The standard library offers a boolean test *leb* for the order on *nat* and a correctness lemma

$$leb_iff : \forall x\, y : nat,\ leb\, x\, y = true \leftrightarrow x \le y$$

We can use *leb* and *leb_iff* to write a certifying test for the order on *nat*.

```
Definition le_dec (x y : nat) : dec (x ≤ y).
  destruct (leb x y) eqn:A.
  - left. apply leb_iff. exact A.
  - right. intros B. apply leb_iff in B. congruence.
Defined.
```

Note that the script for the second subgoal applies the correctness lemma *leb_iff* to the assumption B using the tactic *apply*. This is the first time we apply a lemma to an assumption using the tactic *apply*. It is also possible to rewrite assumptions with the tactic *rewrite*. We have already mentioned that the conversion tactics can be applied to assumptions. To apply a tactic to an assumption A, one ends the command with "*in A*".

Decidability of propositions propagates through logical equivalences. That is, if X and Y are equivalent propositions, then X is decidable if and only if Y is decidable.

```
Definition dec_prop_iff (X Y : Prop) : (X ↔ Y) → dec X → dec Y.
  intros A [B|B].
  - left. τto.
  - right. τto.
Defined.
```

There are many undecidable propositions in Coq. A prominent example of an undecidable proposition is excluded middle (i.e., $XM := \forall X : Prop, X \vee \neg X$). In fact, a proposition is undecidable in Coq if and only if it is independent in Coq.

Exercise 6.3.1 Prove the following goals.

```
Goal ∀ X : Prop, inhabited X → dec X.
Goal ∀ X : Prop, dec X → dec (inhabited X).
Goal ∀ X : Prop, dec (inhabited X) → dec X.
```

Exercise 6.3.2 Complete the following definitions establishing the fact that decidable propositions are closed under conjunction and disjunction.

```
Definition and_dec (X Y : Prop) : dec X → dec Y → dec (X ∧ Y).
Definition or_dec (X Y : Prop) : dec X → dec Y → dec (X ∨ Y).
```

Exercise 6.3.3 Write a certifying test $\forall x\, y : nat.\ \{x < y\} + \{x = y\} + \{y < x\}$.

Exercise 6.3.4 Write a certifying equality test for *bool*.

a) Use the automation tactic *decide equality*.

b) Write the test without using *decide equality*.

Exercise 6.3.5 Compare the certifying equality test *nat_eq_dec* from this section with the boolean equality test *nat_eqb* and its correctness lemma *nat_eqb_agrees* from Section 3.9. We can say that the certifying test combines the boolean test and its correctness lemma into a single function.

a) Define *nat_eqb* and *nat_eqb_agrees* using *nat_eq_dec*.

b) Define *nat_eq_dec* using *nat_eqb* and *nat_eqb_agrees*.

Exercise 6.3.6 Consider the boolean test *leb* and the certifying test *le_dec* from the standard library.

a) Prove the correctness lemma for *leb*.

```
Lemma leb_iff x y : leb x y = true ↔ x ≤ y.
```

b) Define the certifying test *le_dec* using the induction tactic. Follow the definition of *nat_eq_dec* shown above. Compare this definition of *le_dec* with the proof of *leb_iff*.

Exercise 6.3.7 Write a function that given a certifying equality test for a type X yields a certifying equality test for *list* X. Write the function with and without the automation tactic *decide equality*.

Exercise 6.3.8 Complete the following definition. It establishes a function translating a boolean decision of a proposition X into a certifying decision of X.

```
Definition bool2dec (X : Prop) (b : bool) : (X ↔ b = true) → dec X.
```

Exercise 6.3.9 (Program Synthesis) One can use tactics to synthesize ordinary functions not involving proofs. Here are two examples.

```
Definition cas (X Y Z : Type) : (X * Y → Z) → X → Y → Z.
  intros f x y. exact (f (x,y)).
Defined.
Definition car (X Y Z : Type) : (X → Y → Z) → X * Y → Z.
  intros f [x y]. exact (f x y).
Defined.
```

Use the command *Print* to see the synthesized functions. It is also possible to synthesize recursive functions like addition.

```
Definition add : nat → nat → nat.
  fix f 1. intros x y. destruct x as [|x'].
  - exact y.
  - exact (S (f x' y)).
Defined.
```

Use the command *Show Proof* after each tactic to see the partial code of the function synthesized by the tactic.

6.4 Definitions and Lemmas

A definition in Coq is either **inductive** or **plain**. Inductive definitions extend the underlying type theory with new inhabitants obtained with constructors. Plain definitions do not extend the type theory but introduce names for already existing inhabitants. A **plain definition** takes the form $x : t := s$ where x is a name and t and s are terms. The term t must be a type and s must be a member of t. We call x the **name of the definition**, t the **type of the definition**, and s the **body of the definition**. The type of a plain definition acts as the type of the name of the definition.

A plain definition can be either **transparent** or **opaque**. If the definition is transparent, the name and the body of the definition are convertible (i.e., unfolding and folding of the name, known as delta conversion). If the definition is opaque, the name is abstract and cannot be unfolded. Thus all we know about an opaque name is that it is an inhabitant of its type.

Opaque definitions are a means of abstraction. Given an **opaque name** x (i.e., a name introduced by an opaque definition), we can use the specification of x (i.e., the type of x) but not the implementation of x (i.e., the body of the opaque definition of x). Thus every use of an opaque name x will be compatible with every implementation of x. Opaque names are as abstract as **variables** introduced with lambda abstractions, matches, or sections.

A transparent definition can be stated in Coq with a command of the form *Definition* $x : t := s$ or a sequence of commands taking the form

Definition $x : t$. $tactic_1 \cdots tactic_n$ **Defined**.

The tactics in the long form synthesize the body s of the definition. If we replace the command *Defined* in the long form with the command *Qed*, we obtain an opaque definition. The command *Definition* in the long form can be replaced with the command *Lemma*, which has no effect. We use the command *Lemma* only for sequences of the following form.[1]

Lemma $x : t$. **Proof**. $tactic_1 \cdots tactic_n$ **Qed**.

We also use the command sequence

Goal t. **Proof**. $tactic_1 \cdots tactic_n$ **Qed**.

This sequence has the same effect as the sequence starting with *Lemma* except that the missing name x is automatically generated by Coq.

In Coq, a **lemma** is an opaque name established with an opaque definition. The statement of the lemma is the type of the lemma, and the proof of the

[1] The Coq library doesn't always follow this convention. For instance, *le_dec* is defined with *Theorem* and *Defined*.

lemma is the hidden body of the definition. The proof of a lemma certifies that the type of the lemma is inhabited. If we work with a lemma, the uses of the lemma cannot see the proof of the lemma. So all uses of a lemma are abstract in that they do not make any assumptions about the proof of the lemma. This agrees with the mathematical use of lemmas.

There is also an engineering reason for representing lemmas as opaque names in Coq. If lemmas were transparent names, they would be subject to unfolding and their (possibly complex) proofs would unnecessarily participate in conversion checking and type checking.

A **strong lemma** is a lemma whose type is not a proposition. Strong lemmas are a speciality of constructive type theory that don't seem to have a counterpart in Mathematics.

6.5 Decidable Predicates

Every function definable in Coq is computable. Because of opaque definitions, Coq's interpreter may fail to fully evaluate a function application. So the above statement is made with respect to an idealized interpreter treating all plain declarations as transparent.

Functions are described with terms in Coq. When an idealized interpreter evaluates a term describing a function, it will always end up with a term having one of the following forms:

- A lambda abstraction.
- A recursive abstraction.
- A constructor.
- A constructor application $ct_1 \dots t_n$ where c is a constructor and $t_1, \dots, t_n$ are $n \geq 1$ terms. Examples of functions obtained as constructor applications are *cons 3* and *prod nat*.

We call a predicate $p : X \to Prop$ **decidable** if the there is some function that yields for every $x : X$ a decision of $p\,x$.

```
Definition decidable (X : Type) (p : X → Prop) : Type := ∀ x, dec (p x).
```

If p and some function f : *decidable* p are definable in Coq, then f is a decision procedure for p and p is computationally decidable.

Coq can define undecidable predicates. An example of an undecidable predicate is $\lambda X{:}Prop.\, X \lor \neg X$. The undecidability of this predicate follows from the fact that *XM* is independent in Coq.[2]

[2] Note that by undecidable we mean not decidable in Coq. Predicates that are undecidable in Coq may be computationally decidable. On the other hand, predicates that are decidable in Coq are

```
Goal decidable (fun X : Prop ⇒ X ∨ ¬X) → XM.
Proof. intros A X. destruct (A X) as [B|B] ; τto. Qed.
```

The notion of decidability extends to predicates with more than one argument. As is, Coq doesn't give us the possibility to define decidability of predicates in one go, we have to give the definition for each number $n \geq 1$ of arguments. If, more generally, Coq would allow us to define decidability for $n \geq 0$ arguments, decidability of propositions would fall out for $n = 0$.

Exercise 6.5.1 Every predicate equivalent to a boolean test is decidable. Prove the following goal to show this fact.

```
Goal ∀ (X : Type) (p : X → Prop) (f : X → bool), (∀ x, p x ↔ f x = true) → decidable p.
```

6.6 Sigma Types

Sigma types are existential quantifications expressed as proper types. The elim restriction does not apply to sigma types. Given a type X and a predicate $p : X \to Prop$, the elements of the sigma type $\{\,x{:}X \mid p\,x\,\}$ can be seen as pairs consisting of a value $x : X$ and a proof of $p\,x$. Coq defines sigma types as follows.

```
Inductive sig (X : Type) (p : X → Prop) : Type :=
  exist : ∀ x : X, p x → sig p.

Notation "{ x | p }" := (sig (fun x ⇒ p)).
Notation "{ x : X | p }" := (sig (fun x : X ⇒ p)).
```

Consider the type $\forall x{:}nat,\ \{\,y \mid y = 2 * x\,\}$. The elements of this types are functions that take a number x and return a pair consisting of the number $2x$ and a proof of the proposition $y = 2 * x$. Here is a construction of such a function.

```
Definition double (x : nat) : { y | y = 2*x}.
  ∃ (2*x). reflexivity.
Defined.

Compute let (y,_) := double 4 in y.
% 8 : nat
```

Note the use of the tactic *exists* to construct a member of a sigma type. We will refer to functions that yield an element of a sum or sigma type as **certifying functions**. A certifying function combines a function and a correctness proof into a single object. The types of certifying functions can be seen as specifications. For instance, while the type $nat \to nat$ gives us little information about its inhabitants, the type $\forall x : nat,\ \{\,y \mid y = 2x\,\}$ gives us much more information.

We define a certifying function that divides its argument by 2.

always computationally decidable.

```
Definition div2_cert (n : nat) : {k | n = 2*k} + {k | n = 2*k + 1}.
  induction n.
  - left. ∃ 0. reflexivity.
  - destruct IHn as [[k A]|[k A]].
    + right. ∃ k. omega.
    + left. ∃ (S k). omega.
Defined.
```

The result type of *div2_cert* is obtained with yet another kind of sum type (needed since both constituents are proper types).

```
Inductive sum (X Y : Type) :=
| inl : X → sum X Y
| inr : Y → sum X Y.
```

```
Notation "x + y" := (sum x y) : type_scope.
```

Note that the definition of the "+" notation for *sum* is restricted to types. This way the string $2+4$ will still elaborate to the term *plus 2 4*.

Based on the certifying division function *div2_cert* we define ordinary modulo and division functions and prove a correctness lemma.

```
Definition mod2 x := if div2_cert x then 0 else 1.
```

```
Definition div2 x := match div2_cert x with
                       | inl (exist k _) ⇒ k
                       | inr (exist k _) ⇒ k
                     end.
```

```
Goal ∀ x, x = 2 * div2 x + mod2 x.
```

```
Proof.
  intros x. unfold div2, mod2.
  destruct (div2_cert x) as [[k A]|[k A]] ; omega.
Qed.
```

Exercise 6.6.1 Prove $\forall x.\ mod2\ x \le 1$.

Exercise 6.6.2 Prove the following fact about Skolem functions and sigma types.

```
Lemma Sigma_Skolem (X Y : Type) (p : X → Y → Prop) :
 (∀ x, {y | p x y}) → { f | ∀ x, p x (f x) }.
```

Exercise 6.6.3 Establish the following goal and explain why the opposite direction from an existential quantification to a sigma type cannot be established.

```
Goal ∀ X (p : X → Prop), {x | p x} → ∃ x, p x.
```

Exercise 6.6.4 Prove $\forall X{:}Type\ \forall p{:}X{\to}Prop.\ (\exists x.\ p\,x) \leftrightarrow inhabited\,\{\,x \mid p\,x\,\}$.

Exercise 6.6.5 There is a function that for every decidable predicate yields an equivalent boolean test. Prove the following goal to establish this fact.

```
Goal ∀ (X : Type) (p : X → Prop), decidable p → {f : X → bool | ∀ x, p x ↔ f x = true}.
```

Exercise 6.6.6 Write a certifying function that divides its argument by 3.

```
Definition div3_cert (n : nat) : {k | n = 3*k} + {k | n = 3*k + 1} + {k | n = 3*k + 2}.
```

6.7 Strong Truth Value Semantics

There is a canonical injective embedding of *bool* into *Prop*:

```
Definition b2P (x : bool) : Prop := if x then ⊤ else ⊥.
```

Proving the injectivity of *b2P* is straightforward. If we assume *TVS*, we can also prove that *b2P* is surjective. In Mathematics, an injective and surjective function $f : X \to Y$ always comes with an inverse function g such that $g(fx) = x$ and $f(gy) = y$ for all $x : X$ and $y : Y$. So it is natural to ask whether under *TVS* we can define the inverse of *b2p*. The answer is no.

It is, however, consistent to assume **strong truth value semantics**.

```
Definition STVS : Type := ∀ X : Prop, {X=⊤} + {X=⊥}.
```

Assuming *STVS* means assuming a function that for every proposition X yields a decision of type $\{X = True\} + \{X = False\}$. Clearly, *STVS* implies *TVS*.

```
Goal STVS → TVS.
Proof. intros stvs X. destruct (stvs X) ; subst X ; auto. Qed.
```

If we assume *STVS*, we can construct an inverse function for *b2p*.

```
Section STVS.
  Variable stvs : STVS.
  Definition P2b (X : Prop) : bool := if stvs X then true else false.
  Lemma P2b⊤ : P2b ⊤ = true.
  Proof.
    unfold P2b. destruct (stvs ⊤) as [A|A].
    + reflexivity.
    + exfalso. rewrite ← A. exact I.
  Qed.
  Lemma P2b⊥ : P2b ⊥ = false.
  Proof.
    unfold P2b. destruct (stvs ⊥) as [A|A].
    + exfalso. rewrite A. exact I.
    + reflexivity.
  Qed.
```

```
  Goal ∀ x : bool, P2b (b2P x) = x.
  Proof. intros [|] ; simpl. exact P2b⊤. exact P2b⊥. Qed.
  Goal ∀ X : Prop, b2P (P2b X) = X.
  Proof.
    intros X. destruct (stvs X) ; subst X.
    – rewrite P2b⊤. reflexivity.
    – rewrite P2b⊥. reflexivity.
  Qed.
End STVS.
```

The names defined in a section remain defined after a section is closed. Their types are modified such that the variables of the section used in the definitions are taken as arguments. For instance:

```
Print P2b.
> fun (stvs : STVS) (X : Prop) ⇒ if stvs X then true else false
> : STVS → Prop → bool
```

Exercise 6.7.1 Prove that *b2P* is injective.

Exercise 6.7.2 Prove that *TVS* implies that *b2P* is surjective.

Exercise 6.7.3 Prove that *P2b* is injective.

```
Goal ∀ A : STVS, ∀ X Y : Prop, P2b A X = P2b A Y → X = Y.
```

Exercise 6.7.4 Show that *STVS* implies that every proposition is decidable.

```
Goal STVS → ∀ X : Prop, dec X.
```

Exercise 6.7.5 Show that *STVS* implies that every predicate is decidable.

```
Goal STVS → ∀ (X : Type) (p : X → Prop), decidable p.
```

Coq Summary

New Tactics

constructor, *decide equality*.

New Inductive Types from the Standard Library

sumbool, *sumor*, *sum*, *sig*, *inhabited*.

Applying Tactics to Assumptions

To apply a tactic to an assumption A, end the tactic command with "*in* A". See the definition of *le_dec* in Section 6.3 for an example.

7 Inductive Predicates

An inductive definition introduces a type constructor together with a family of value constructors. If the type constructor yields a proposition, we speak of an inductive predicate and of proof constructors. We already know Coq's inductive predicates for conjunctions (*and*), disjunctions (*or*), and existential quantifications (*ex*) (see Chapter 2). Another inductive predicate we have introduced is *inhabited*.

In this chapter we will take a closer look at inductive predicates. Coq's facility for inductive definitions is extremely powerful and supports many advanced applications. The idea of inductive definitions originated with Peano's axioms (i.e., the inductive definition of *nat* with *O* and *S*) and developed further with proof systems for logical systems.

When we define an inductive predicate, we define a family of inductive propositions by specifying the syntax and the proof rules for the propositions. The inductive predicates *and*, *or* and *ex* give us a first idea of the flexibility of this approach. It turns out that we can go much further. Every recursively enumerable predicate can be defined as an inductive predicate in Coq. This is in contrast to computable functions, which are not necessarily definable in Coq.

7.1 Nonparametric Arguments and Linearization

We start our explanation of inductive predicates with an extreme case: A definition of a predicate on numbers that holds exactly for the number 0.

```
Inductive zero : nat → Prop :=
| zeroI : zero 0.
```

The definition provides exactly one proof *zeroI*, which proves the proposition *zero 0*. The propositions *zero 1*, *zero 2*, *zero 3*, and so forth are all unprovable. We characterize the inductive predicate *zero* as follows.

```
Lemma zero_iff x :
  zero x ↔ x = 0.
```

```
Proof.
  split ; intros A.
  - destruct A. reflexivity.
  - subst x. constructor.
Qed.
```

The interesting step of the proof is *destruct A*, which does a case analysis on the proof of *zero x*. Since there is only a single proof constructor *zeroI* : *zero 0*, the case analysis yields a single subgoal where the variable x is instantiated to 0.

The argument of the inductive predicate *zero* is called **nonparametric** since it is instantiated by the proof constructor *zeroI* : *zero 0*. This is the first time we see an inductive predicate with a nonparametric argument. Check the definitions of the inductive predicates *and*, *or*, *ex*, and *inhabited* to see that all arguments of these predicates are parametric.

There is an important technicality one has to know about nonparametric arguments: When the tactics *destruct* and *induction* are applied to an inductive assumption $A : ct_1 \dots t_n$, the terms t_i for the nonparametric arguments of c must be variables not appearing in the other terms. We say that inductive assumptions must be **linear** when they are used with *destruct* and *induction*. Coq offers the tactic *remember* to **linearize** inductive assumptions.

```
Goal ¬ zero 2.
Proof. intros A. remember 2 as x. destruct A. discriminate Heqx. Qed.
```

Exercise 7.1.1 Make sure you can prove the propositions *zero 0*, ¬*zero 7*, and $\forall x.\ \neg zero\,(S\,x)$ without using lemmas.

Exercise 7.1.2 Prove that the predicate *zero* is decidable.

Exercise 7.1.3 Prove the following lemma.

```
Lemma remember (X : Type) (p : X → Type) (x : X) :
  (∀ y, y = x → p y) → p x.
```

Try to understand why the lemma justifies the tactic *remember*. Use the lemma and the tactic *pattern* to prove the proposition $\forall x.\ \neg zero\ (S\ x)$.

Exercise 7.1.4 Prove the following impredicative characterization of *zero*.

```
Goal ∀ x, zero x ↔ ∀ p : nat → Prop, p 0 → p x.
```

Exercise 7.1.5 Define a boolean test $zerob : nat \to bool$ and prove the correctness condition $\forall x.\ zero\ x \leftrightarrow zerob\ x = true$.

Exercise 7.1.6 Define an inductive predicate $leo : nat \to Prop$ with two proof constructors $leo0 : leo\ 0$ and $leo1 : leo\ 1$.

a) Prove $\forall x,\ leo\ x \leftrightarrow x \leq 1$.

b) Characterize *leo* impredicatively and prove the correctness.

c) Characterize *leo* with a boolean test $leob : nat \to bool$ and prove the correctness of the characterization.

7.2 Even

Our next example is an inductive predicate *even* that holds exactly for the even numbers. This time we use two proof rules, one for *even 0* and one for $even\ (S\,(S\,x))$.

$$\frac{}{even\ 0} \qquad\qquad \frac{even\ x}{even\ (S(S\ x))}$$

The two proof rules can be expressed with two proof constructors:

$$evenO \;:\; even\ 0$$
$$evenS \;:\; \forall x{:}nat.\ even\ x \to even\ (S\,(S\,x))$$

From the types of the proof constructors it is clear that the argument of *even* is nonparametric. We now introduce the predicate *even* and the proof constructors *evenO* and *evenS* with a single inductive definition.

```
Inductive even : nat → Prop :=
| evenO : even 0
| evenS x : even x → even (S (S x)).
```

The type of the constructor *evenS* is specified with a notational convenience we have seen before in the statement of lemmas. The convenience makes it possible to specify argument variables of a constructor without types, leaving it to Coq to infer the types.

We prove a lemma characterizing *even* non-inductively.

```
Lemma even_iff x :
  even x ↔ ∃ k, x = 2*k.
```

```
Proof.
  split ; intros A.
  - induction A.
    + ∃ 0. reflexivity.
    + destruct IHA as [k IHA]. subst x. ∃ (S k). simpl. omega.
  - destruct A as [k A]. subst x. induction k ; simpl.
    + constructor.
    + replace (S(k+S(k+0))) with (S (S (2*k))) by omega.
      constructor. exact IHk.
Qed.
```

Both directions of the proof deserve careful study. The direction from left to right is by induction on the proof $A : even\ x$. The induction does a case analysis for the two proof constructors of *even*. In each case the nonparametric argument x is instantiated as specified by the type constructor. For *evenS* we get $x = S(S\,x')$ and the inductive hypothesis $IHA : \exists k.\ x' = 2 * k$.[1]

The direction from right to left first eliminates the existential quantification for k and then proves the so obtained claim by induction on $k : nat$. The induction step uses the tactic *replace* to rewrite with the equation $S(k + S(k + 0)) = S(S(2*k))$, which is established by the tactic *omega*. This is the first time we use the tactic *replace*. If the annotation *by omega* is omitted, *replace* will introduce an extra subgoal to establish the equation.

The next two lemmas prove simple facts about *even* using case analysis on proofs of propositions obtained with *even*. In each case the linearization of the inductive assumption with the tactic *remember* is essential.

```
Goal ¬ even 3.

Proof.
  intros A. remember 3 as x. destruct A.
  - discriminate Heqx.
  - destruct A ; discriminate Heqx.
Qed.

Lemma even_descent x :
  even (S (S x)) → even x.

Proof.
  intros A. remember (S (S x)) as y.
  destruct A as [|y A].
  - discriminate Heqy.
  - congruence.
Qed.
```

[1] The proof script reuses the variable x for x'. You can get the variable x' by annotating the induction command with *as* $[\,|x'\ A']$.

Exercise 7.2.1 Prove *even 6* and ¬*even 5* without using lemmas.

Exercise 7.2.2 Prove the following goals without using lemmas.

(a) **Goal** ∀ x y, even x → even y → even (x+y).

(b) **Goal** ∀ x y, even x → even (x+y) → even y.

(c) **Goal** ∀ x, even x → even (S x) → ⊥.

Exercise 7.2.3 Prove the so-called **inversion lemma** for *even*.

```
Lemma even_inv x : even x → x = 0 ∨ ∃ x', x = S (S x') ∧ even x'.
```

Exercise 7.2.4 Prove the following impredicative characterization of evenness.

```
Goal ∀ x, even x ↔ ∀ p : nat → Prop, p 0 → (∀ y, p y → p (S (S y))) → p x.
```

Exercise 7.2.5 Some proofs need ideas. Try to prove $\forall x,\ \neg even\ x \to even\ (S\ x)$. As is, the induction on $x : nat$ will not go through. The problem is that the induction on $x : nat$ takes away a single S while the constructor *evenS* takes away two $S's$. The standard cure consists in generalizing the claim so that the inductive hypothesis becomes strong enough. Convince yourself that the proof of the following lemma generalizing the claim is doable.

```
Lemma even_succ x : (¬ even x → even (S x)) ∧ (¬ even (S x) → even x).
```

Hint: The apply tactic can be used with a proof of a conjunction of implications. In this case *apply* attempts to apply one of the implications.

Exercise 7.2.6 Prove $\forall x,\ even\ x \leftrightarrow \neg even\ (S\ x)$. Hint: Use the lemma *even_succ* from Exercise 7.2.5.

Exercise 7.2.7 Prove that the predicate *even* is decidable. Hint: Use the lemma *even_succ* from Exercise 7.2.5.

Exercise 7.2.8 Here is an inductive definition of an evenness predicate with a parametric argument.

```
Inductive even' (x : nat) : Prop :=
| even'O : x=0 → even' x
| even'S y : even' y → x = S (S y) → even' x.
```

a) Prove $\neg even'\ 3$.

b) Prove $\forall x.\ even'\ x \leftrightarrow even\ x$.

Exercise 7.2.9 Here is a boolean test for evenness.

```
Fixpoint evenb (x : nat) : bool :=
  match x with
    | 0 ⇒ true
    | S (S x') ⇒ evenb x'
    | _ ⇒ false
  end.
```

Try to prove $\forall x.\ even\ x \leftrightarrow evenb\ x = true$. The direction from left to right is a straightforward induction on a proof of *even x*. The direction from right to left is problematic since an induction on $x : nat$ takes away one S while the constructor *evenS* takes away two S's. Proving the following more general claim solves the problem.

```
Lemma evenb_even x : (evenb x = true → even x) ∧ (evenb (S x) = true → even (S x)).
```

7.3 Less or Equal

Coq defines the order predicate "≤" for natural numbers inductively based on the following proof rules.[2]

$$\frac{}{x \le x} \qquad\qquad \frac{x \le y}{x \le S\,y}$$

The exact definition is

```
Inductive le (x : nat) : nat → Prop :=
| le_n : le x x
| le_S y : le x y → le x (S y).
Notation "x ≤ y" := (le x y) (at level 70).
```

Note that the first argument of the inductive predicate *le* is parametric and that the second argument is nonparametric. We will always write inductive definitions such that all parametric arguments appear as **parameters** in the head of the inductive definition. Note that *le* is the first inductive predicate we see having both parametric and nonparametric arguments.

To get familiar with *le*, we prove a lemma characterizing *le* non-inductively.

```
Lemma le_iff x y :
  x ≤ y ↔ ∃ k, k + x = y.
```

[2] Use the commands *Locate* and *Print* to see Coq's definition of "≤" for *nat*.

```
Proof.
  split.
  - intros A. induction A as [|y A].
    + ∃ 0. reflexivity.
    + destruct IHA as [k B]. ∃ (S k). simpl. congruence.
  - intros [k A]. subst y. induction k ; simpl.
    + constructor.
    + constructor. exact IHk.
Qed.
```

The proof deserves careful study. The direction from left to right is by induction on a proof of $x \leq y$. The other direction is by induction on k.

Next we write an informative test for *le*. This takes some preparation. We leave the proofs of the first three lemmas as exercises.

```
Lemma le_O x :  0 ≤ x.

Lemma le_SS x y :  x ≤ y → S x ≤ S y.

Lemma le_Strans x y :  S x ≤ y → x ≤ y.

Lemma le_zero  x :
  x ≤ 0 → x = 0.

Proof.
  intros A. remember 0 as y. destruct A as [|y A].
  - reflexivity.
  - discriminate Heqy.
Qed.

Lemma le_SS' x y :
  S x ≤ S y → x ≤ y.

Proof.
  intros A. remember (S y) as y'. induction A as [|y' A].
  - injection Heqy'. intros A. subst y. constructor.
  - injection Heqy'. intros B. subst y'. apply le_Strans, A.
Qed.

Definition le_dec x y : dec (x ≤ y).
  revert y. induction x ; intros y.
  - left. apply le_O.
  - destruct y.
    + right. intros A. apply le_zero in A. discriminate A.
    + destruct (IHx y) as [A|A].
      * left. apply le_SS, A.
      * right. intros B. apply A, le_SS', B.
Defined.
```

Exercise 7.3.1 Prove the lemmas *le_O*, *le_SS*, and *le_Strans* without using *omega*. Hints: Lemma *le_O* follows by induction on x. Lemmas *le_SS* and *le_Strans* follow by induction for *le*.

Exercise 7.3.2 Prove the inversion lemma for *le*.

```
Lemma le_inv x y : x ≤ y → x = y ∨ ∃ y', y = S y' ∧ x ≤ y'.
```

Exercise 7.3.3 Prove that *le* is transitive. Do not use *omega*.

```
Lemma le_trans x y z : x ≤ y → y ≤ z → x ≤ z.
```

Hint: Do the proof by induction for $y \leq z$.

Exercise 7.3.4 Prove the following goal not using *omega*.

```
Goal ∀ x y, S x ≤ y → x ≠ y.
```

Hint: Proceed by induction on y and use the lemmas *le_zero* and *le_SS'*.

Exercise 7.3.5 Prove that *le* is anti-symmetric. Do not use *omega*.

```
Goal ∀ x y, x ≤ y → y ≤ x → x=y.
```

Hint: Proceed by induction on x and use *le_zero*, *le_Strans*, and *le_SS'*.

Exercise 7.3.6 Prove that *le* and the boolean test *leb* from the standard library agree. Do not use omega.

```
Goal ∀ x y, x ≤ y ↔ leb x y = true.
```

Hint: For the direction from left to right you will need two straightforward lemmas for *leb*. For the other directions use the lemmas *le_O* and *le_SS*.

7.4 Equality

Coq defines equality inductively.

```
Inductive eq (X : Type) (x : X) : X → Prop :=
| eq_refl : eq x x.
Notation "x = y" := (eq x y) (at level 70).
```

Note that the first two arguments of the inductive predicate *eq* are parametric and that the third argument is nonparametric. It is easy to establish the Leibniz characterization of equality.

```
Lemma Leibniz (X : Type) (x y : X) :
  x = y ↔ ∀ p : X → Prop, p x → p y.
```

```
Proof.
  split ; intros A.
  - destruct A. auto.
  - apply (A (fun z ⇒ x = z)). constructor.
Qed.
```

7.5 Exceptions to the Elim Restriction

The exceptions to the elim restriction can be stated as follows: If an inductive predicate has at most one proof constructor and the nonparametric arguments of the proof constructor are all proofs, then the elim restriction does not apply to matches for this predicate.

An interesting exception to the elim restriction is the inductive predicate *eq* whose single proof constructor

$$eq_refl \;:\; \forall X : Type\; \forall x : X.\; eq\; x\; x$$

has only parametric arguments (i.e., arguments fixed in the head of the inductive definition of *eq*). Thus the elim restriction does not apply to matches on equality proofs. This provides for the definition of the following casting function.

```
Definition cast (X : Type) (x y : X) (f : X → Type) :  x = y → f x → f y.
  intros A B. destruct A. exact B.
Defined.
```

The function *cast* gives us a function that given a proof of $x = y$ converts from type fx to type fy. Here is an example for the use of *cast*.

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

```
Goal ∀ n, fin n → fin (n+0).
```

```
Proof. intros n. apply cast. omega. Qed.
```

Note that the cast is needed since the terms *fin* n and *fin* $(n + 0)$ are not convertible.

Exercise 7.5.1 Prove the following goal. Explain why the elim restriction does not apply to conjunctions.

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

Exercise 7.5.2 Explain why the elim restriction applies to matches for the inductive predicate *inhabited*.

Exercise 7.5.3 Complete the following definition. Explain why your definition exploits an exception to the elim restriction.

```
Definition exfalso : ⊥ → ∀ X : Type, X := · · ·
```

Exercise 7.5.4 Prove the following goal.

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

Note that goal is formulated without making use of inductive types. Yet it can only be proven using inductive types.

7.6 Safe and Nonuniform Parameters

Our final example is an inductive predicate $safe : (nat \to Prop) \to nat \to Prop$ such that $safe\ p\ n$ holds if and only if p holds for some $k \geq n$. We base the inductive definition on the following rules.

$$\frac{p\ n}{safe\ p\ n} \qquad\qquad \frac{safe\ p\ (S\ n)}{safe\ p\ n}$$

Defining *safe* in Coq is straightforward.

```
Inductive safe (p : nat → Prop) (n : nat) : Prop :=
| safeB : p n → safe p n
| safeS : safe p (S n) → safe p n.
```

One reason for considering *safe* is that it has both a **uniform** and a **nonuniform** parameter.[3] The parameter p is uniform since it is not instantiated in the types of the proof constructors *safeB* and *safeS*. The parameter n is nonuniform since it is instantiated to $S\ n$ in the type of the constructor *safeS*. The argument n does not qualify as a nonparametric argument of *safe* since the instantiation appears in argument position rather than in result position. This is the first time we encounter a type constructor with a nonuniform parameter.

When we use the tactic *induction* on a proof of a proposition obtained with an inductive predicate, both the nonuniform parametric arguments and the nonparametric arguments of the proposition must be linearized. If we use the tactic *destruct*, it suffices if the nonparametric arguments are linearized. For instance, if we have an assumption $A : safe\ p\ 0$, *destruct* can be applied to A but *induction* must not be applied to A.

We prove that $safe\ p$ is downward closed.

```
Lemma safe_dclosed k n p :
  k ≤ n → safe p n → safe p k.
```

[3] A parameter is a parametric argument.

```
Proof.
  intros A B. induction A as [|n A].
  - exact B.
  - apply IHA. right. exact B.
Qed.
```

The proof is by induction on a proof of $k \leq n$. Note the use of the tactic *right* to apply the second constructor of *safe*. The tactics *left* and *right* can be used with every type constructor that has two value constructors.

We prove that *safe* satisfies its specification.

```
Lemma safe_iff p n :
  safe p n ↔ ∃ k, n ≤ k ∧ p k.
Proof.
  split ; intros A.
  - induction A as [n B|n A].
    + ∃ n. auto.
    + destruct IHA as [k [B C]].
      ∃ k. split. omega. exact C.
  - destruct A as [k [A B]].
    apply (safe_dclosed A). left. exact B.
Qed.
```

The direction from left to right is by induction on a proof of *safe p n*. From the destructuring pattern for the induction we learn that a name for the nonuniform parameter n must be given for both subgoals. This must be done for nonuniform parameters in general.

The direction from right to left follows with the lemma *safe_dclosed*. Using this lemma is essential since a direct proof of the more specific claim we have at this point seems impossible.

The predicate *safe* is different from the other inductive predicates we saw in this chapter in that it is impossible to express it with a boolean test. This is the case even if we assume that the argument p is a decidable predicate.[4]

Exercise 7.6.1 We define a predicate *least* such that *least p n k* holds if and only if k is the least number such that $n \leq k$ and $p\,k$ holds.

```
Inductive least (p : nat → Prop) (n : nat) : nat → Prop :=
| leastB : p n → least p n n
| leastS k : ¬ p n → least p (S n) k → least p n k.
```

Note that the first argument of *least* is a uniform parameter, the second argument is a nonuniform parameter, and the third argument is nonparametric. Prove the following correctness lemmas for *least*.

[4] Think of pn as the statement saying that a particular Turing machine halts on a particular input in at most n steps.

```
Lemma least_correct1 p n k : least p n k → p k.
Lemma least_correct2 p n k : least p n k → n ≤ k.
Lemma least_correct3 p n k : least p n k → ∀ k', n ≤ k' → p k' → k ≤ k'.
Lemma least_correct4 p n k : (∀ x, dec (p x)) → p (n+k) → ∃ k', least p n k'.
Lemma least_correct p n k (p_dec : ∀ x, dec (p x)) :
  least p n k ↔ p k ∧ n ≤ k ∧ ∀ k', n ≤ k' → p k' → k ≤ k'.
```

Hint: Use *le_lt_eq_dec* from the standard library for the proof of *least_correct3*.

7.7 Constructive Choice for Nat

We will now construct a function

```
cc_nat : ∀ p : nat → Prop, (∀ x, dec (p x)) → (∃ x, p x) → {x | p x}
```

we call *constructive choice for nat*. For a decidable predicate p on numbers constructive choice yields a function that for every proof of an existential quantification $\exists x.\, px$ yields a value of the sigma type $\{\, x \mid px \,\}$. Thus *cc_nat* bypasses the elim restriction for existential quantifications of decidable predicates on numbers. We will obtain this remarkable result with a new proof technique based on the inductive predicate *safe* from the last section.

For convenience, we declare a decidable predicate p in a section.

```
Section First.
Variable p : nat → Prop.
Variable p_dec : ∀ n, dec (p n).
```

We now write a function *first* that from a proof of *safe* $p\ n$ obtains a value of $\{\, k \mid pk \,\}$. The function *first* is the cornerstone of the construction of *cc_nat*. Clearly, *first* overcomes the elim restriction. We define *first* by recursion on the given proof of *safe* $p\ n$.

```
Fixpoint first (n : nat) (A : safe p n) : {k | p k} :=
  match p_dec n with
    | left B ⇒ exist p n B
    | right B ⇒ first match A with
                       | safeB C ⇒ match B C with end
                       | safeS A' ⇒ A'
                     end
  end.
```

Given that *first* computes by recursion on A, one would expect that *first* first matches on A. However, this is impossible because of the elim restriction. So we first match on *p_dec* n. If we obtain a proof of $p\ n$, we are done. Otherwise, we recurse on a proof of *safe* $p\ (S\,n)$ we obtain by matching on the proof A of

safe p n. This time the elim restriction does not apply since we are constructing a proof. We obtain two cases. The case for *safeS* is straightforward since we get a proof of *safe p* $(S\,n)$ by taking off the constructor. The case for *safeB* yields a proof C of $p\,n$. Since we have a proof B of $\neg p n$, we can match on the proof $B\,C$ of *False*. Now we are done since each rule of the match returns a proof of *safe p* $(S\,n)$ that is obtained by taking off a constructor of the proof A (vacuous reasoning).

The recursion scheme underlying *first* is nonstandard. The standard recursion scheme would first match on the proof and than recurse. The recursion scheme we see with *first* first recurses and only then matches on the proof. This way the elim restriction can be bypassed. We speak of an **eager proof term recursion**.

It is now straightforward to construct the certifying function *cc_nat*. We obtain the result by applying *first* to a proof of *safe p 0*. The proof of *safe p 0* we obtain with the lemma *safe_dclosed* from a proof of *safe p n* for some n. The n and the proof of *safe p n* we obtain from the given proof of $\exists x.\, p x$.

```
Lemma cc_nat : (∃ x, p x) → {x | p x}.

Proof.
  intros A. apply first with (n:=0).
  destruct A as [n A].
  apply safe_dclosed with (n:=n). omega. left. exact A.
Qed.
```

Note the "with" annotations used with the tactic *apply*. They provide a convenient means for specifying implicit arguments of the function being applied.

There is a straightforward algorithmic idea underlying *cc_nat* we may call linear search: To find the least $k \geq n$ such that $p n$, increment n until $p n$ holds. What is interesting about linear search from our perspective is that linear search is not structurally recursive and that it may not always terminate. We can see *first* as a logical reformulation of linear search that is structurally recursive.

Exercise 7.7.1 Write a constructive choice function for *bool*.

```
Definition cc_bool (p : bool → Prop) (p_dec : ∀ x, dec (p x)) : (∃ x, p x) → {x | p x}.
```

Exercise 7.7.2 Complete the definitions of the following recursive and certifying functions with scripts. Assume a section declaring a decidable predicate p. Follow the eager proof term recursion scheme from *first*.

```
Fixpoint first1 (n : nat) (A : safe p n) : {k | p k ∧ k ≥ n}.

Fixpoint first2 (n : nat) (A : safe p n) : {k | p k ∧ k ≥ n ∧ ∀ k', n ≤ k' → p k' → k ≤ k'}.
```

Hint: First redefine *first* with a script. Check the partial proof terms you obtain with the command *Show Proof*. Then refine the script for *first* to obtain the script for *first1*.

Exercise 7.7.3 Write constructive choice functions for the finite types *fin n*.

```
Definition cc_fin (n : nat) (p : fin n → Prop) (p_dec : ∀ x, dec (p x))
 : (∃ x, p x) → {x | p x}.
```

7.8 Technical Summary

An inductive definition introduces a family of typed names called constructors. One of the constructors yields types and is called type constructor. The remaining constructors are called value constructors and yield the elements of the types obtainable with the type constructor. An inductive value is a value obtained with a constructor. Thus an inductive predicate is a predicate obtained with a constructor, a proof constructor is a value constructor yielding a proof, and inductive proposition is a proposition obtained with a type constructor.

An inductive definition comes with a list of named parameters specified in the head of the definition. The parameters appear as leading arguments of every constructor introduced by the inductive definition. We speak of the parametric arguments of a constructor. The constructors may have additional arguments, which we call nonparametric arguments. There is the constraint that the result type of a value constructor must not instantiate parametric arguments of the type constructor. The parametric arguments of a value constructor do not appear in matches and the type specification of the constructor in the introducing inductive definition.

As example we consider the following inductive definition.

```
Inductive least (p : nat → Prop) (n : nat) : nat → Prop :=
| leastB : p n → least p n n
| leastS k : ¬ p n → least p (S n) k → least p n k.
```

The definition introduces the constructors

$$\begin{aligned} least &: (nat \to Prop) \to nat \to nat \to Prop \\ leastB &: \forall p{:}nat \to Prop\ \forall n{:}nat.\ p\,n \to least\ p\ n\ n \\ leastS &: \forall p{:}nat \to Prop\ \forall n{:}nat\ \forall k{:}nat.\ \neg p\,n \to least\ p\ (S\,n)\ k \to least\ p\ n\ k \end{aligned}$$

The leading two arguments of each constructor are parametric, the remaining arguments are nonparametric. The type constructor *least* and the value constructor *leastB* have one nonparametric argument each, and the value constructor *leastS* has three nonparametric arguments.

Our inductive definitions will always be such that for every nonparametric argument of the type constructor there will be at least one value constructor that instantiates this argument in its result type. Coq does not enforce this condition.

The elim restriction applies to matches on proofs of inductive propositions where the underlying inductive definition either has more than one proof constructor or has a single proof constructor taking a nonparametric argument specified with a proper type. For instance, the elim restriction applies to matches on proofs of disjunctions and existential quantifications, but it does not apply to matches on proofs of equations and conjunctions.

Coq distinguishes between uniform and nonuniform parameters of inductive definitions. A parameter of an inductive definition is nonuniform if it is instantiated in argument position in the type specification of a value constructor. For instance, the inductive definition *least* has the uniform parameter p and the nonuniform parameter n. The nonuniformity of n is due to the type of the third nonparametric argument of the value constructor *leastS*.

When we apply the tactic *destruct* to a proof A of an inductive proposition $Ct_1 \dots t_n$, all terms t_i giving nonparametric arguments must be variables that do not appear in the other terms. Similarly, when we apply the tactic *induction* to a proof A of an inductive proposition $Ct_1 \dots t_n$, all terms t_i giving nonparametric or nonuniform parametric arguments must be variables that do not appear in the other terms. We say that inductive propositions are linear if they satisfy this condition. Inductive propositions can be linearized with the tactic *remember*.[5]

7.9 Induction Lemmas

When we apply the tactic *induction* to an assumed value of an inductive type, the induction lemma for the underlying type constructor is applied. To have an example, we consider the inductive definition of *even*.

```
Inductive even : nat → Prop :=
| evenO : even 0
| evenS x : even x → even (S (S x)).
```

[5] Unfortunately, the tactics *destruct* and *induction* do not give warnings when they are applied to proofs of nonlinear inductive propositions. Instead, they linearize the proposition automatically and forget the equations relating the fresh variables with the moved away argument terms. This often leads to unprovable subgoals.

The type of the induction lemma *even_ind* Coq derives for *even* is as follows.

$$\forall p\!:\!nat \to Prop.$$
$$p\,0 \to$$
$$(\forall x\!:\!nat.\ even\,x \to p\,x \to p(S(S\,x))) \to$$
$$\forall x\!:\!nat.\ even\,x \to p\,x$$

Note that each constructor contributes a premise of the implication. When we apply the tactic *induction* to an assumption $A : even\ x$, the goal is rearranged by moving all assumptions depending on the variable x to the claim. Thus these assumptions become part of the induction predicate p and hence appear in the inductive hypothesis px of the premise for the constructor *evenS*.

Our second example is the inductive definition of *le*.

```
Inductive le (x : nat) : nat → Prop :=
| le_n : le x x
| le_S y : le  x y → le x (S y).
```

The induction lemma *le_ind* Coq generates for *le* quantifies the uniform parameter x at the outside.

$$\forall x\!:\!nat\ \forall p\!:\!nat \to Prop.$$
$$p\,x \to$$
$$(\forall\ y\!:\!nat.\ le\,x\,y \to p\,y \to p(S\,y)) \to$$
$$\forall y\!:\!nat.\ le\,x\ y \to p\,y$$

If you look at the induction lemma Coq generates for *least*, you will see that the nonuniform parameter is treated like the nonparametric argument in that it appears as an argument of the induction predicate p.

When we work with paper and pencil, doing inductive proofs based on inductive definitions requires considerable training and great care. When we work with Coq, the tedious details are taken care of automatically and proof correctness is guaranteed.

Exercise 7.9.1 Complete the following definitions of the induction lemmas for *even* and *le*.

```
Definition even_ind' (p : nat → Prop) (r1 : p 0) (r2 : ∀ x, even x → p x → p (S (S x)))
  : ∀ x, even x → p x :=  · · · .
```

```
Definition le_ind' (x : nat) (p : nat → Prop) (r1 : p x) (r2 : ∀ y, le x y → p y → p (S y))
  : ∀ y, le x y → p y :=  · · · .
```

Coq Summary

New Tactics

remember, *replace*.

Automation Tactic *inversion*

The automation tactic *inversion* subsumes the capabilities of the tactics *destruct*, *discriminate*, and *injection*. The use of *inversion* is convenient if it solves the goal. Otherwise *inversion* often creates subgoals with many equational assumptions. We will use *inversion* only if it solves the goal. Here are two examples.

```
Goal ¬ even 1.  Proof. intros A. inversion A. Qed.
Goal ¬ 7 ≤ 0.  Proof. intros A. inversion A.  Qed.
```

With Annotations for *apply*

With annotations used with the tactic *apply* are a convenient means for specifying implicit arguments of the function being applied. See the definition of *cc_nat* in Section 7.7.

8 Lists

In this chapter we study lists in constructive type theory. Lists are a basic data structure providing for the representation of finite sequences and finite sets. We will consider predicates like list membership, list inclusion, list equivalence, list disjointness, and duplicate freeness. We will also consider functions for concatenation, mapping, product, filtering, element removal, length, cardinality, and power lists.

Decidability issues play an important role for data structures in general and lists in particular. For instance, if the base type comes with decidable equality, membership as well as inclusion, equivalence, and equality of lists are decidable. Moreover, quantification over the elements of a list preserves decidability. Finally, list functions like cardinality and element removal require a decider for base type equality.

Lists and decidability play a major role in many formal developments we are interested in. As is, Coq's library does not provide adequate support for lists and decidability. Thus we provide a base library *Base.v* realizing the infrastructure for lists and decidability introduced in this chapter. The base library comes with several automation features including an automatic handling of side conditions for decidability.

In this chapter we explain the infrastructure provided by the base library from the user's point of view. We will not say much about the implementation of the base library, which uses several advanced features of Coq we have not seen so far. The interested reader may consult the source code of the base library.

From now on we assume that our Coq developments start with the command

```
Require Export Base.
```

The command loads the base library designed for this course. The base library switches Coq into implicit arguments mode and provides Coq's basic infrastructure for numbers and setoid rewriting. It also provides the infrastructure for lists and decidability presented in this chapter. The base library can be made availably by placing a compiled version of the file *Base.v* in Coq's load path (use the command *coqc Base.v* for compilation).

8.1 Constructors and Notations

Lists can be seen as finite sequences $[x_1 ; \dots ; x_n]$ of values. Formally, lists are obtained with two constructors *nil* and *cons*.

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

The constructor *nil* provides the **empty list**. The constructor *cons* yields for a value x and a list $[x_1 ; \dots ; x_n]$ the list $[x;x_1 ; \dots ; x_n]$. Given a list *cons x A*, we call x the **head** and A the **tail** of the list. Given a list $[x_1 ; \dots ; x_n]$, we call n the **length** of the list and $x_1, \dots, x_n$ the **elements** of the list. An element may appear more than once in a list. For instance, $[2;2;3]$ is a list of length 3 that has 2 elements.

We are now ready for the formal definition of lists:

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

The definition provides three constructors:

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

The formal definition of lists ensures that all elements of a list are taken from the same type. For every type X we obtain a type *list* X. The elements of *list* X are the lists we can obtain with *nil* and *cons* from elements of X. We speak of **lists over X**.

We arrange things such that the *base type* X is an implicit argument of the constructors *nil* and *cons*. While X is implicit for *cons* by default, this arrangement need to be declared explicitly for *nil* (since *nil* has no explicit arguments determining X). For *nil* the implicit argument X will be determined from the surrounding context.

For *cons* we employ a right associative infix notation:

$$x :: A \;:=\; cons\ x\ A$$

We can now write $x :: y :: A$ for *cons* x (*cons* y A).

8.2 Recursion and Induction

Like numbers, lists are a *recursive data structure*. Thus functions on lists are typically defined by structural recursion on lists, and lemmas about lists are often shown by structural induction on lists. Recursion and induction on lists are quite similar to recursion and induction on numbers.

Length

We define the **length** $|A|$ of a list A as follows:

$$|nil| := 0$$
$$|x :: A| := S|A|$$

Using bracket notation, we have

$$|[x_1 ; \dots ; x_n]| = n$$

In Coq, we realize $|A|$ with a recursive function

```
length : ∀ X : Type, list X → nat
```

and an accompanying notation.

Concatenation

We define the **concatenation** $A +\!\!+ B$ of two lists A and B as follows:

$$nil +\!\!+ B := B$$
$$x :: A +\!\!+ B := x :: (A +\!\!+ B)$$

Using bracket notation, we have

$$[x_1 ; \dots ; x_m] +\!\!+ [y_1 ; \dots ; y_n] = [x_1 ; \dots ; x_m ; y_1 ; \dots ; y_n]$$

In Coq, we realize $A +\!\!+ B$ with a recursive function

```
app : ∀ X : Type, list X → list X → list X
```

and an accompanying notation.

Reversal

We define the **reversal** of a list as follows:

$$rev\ nil := nil$$
$$rev\ (x :: A) := rev\ A +\!\!+ [x]$$

Using bracket notation, we have

$$rev\ [x_1 ; \dots ; x_n] = [x_n ; \dots ; x_1]$$

In Coq, we realize $rev\ A$ with a recursive function

```
rev : ∀ X : Type, list X → list X
```

Map

We define the **map** of a list under a function f as follows:

$$map\ f\ nil := nil$$
$$map\ f\ (x :: A) := fx :: map\ f\ A$$

Using bracket notation, we have

$$map\ f\ [x_1\,;\,\dots\,;x_n] \ = \ [fx_1\,;\,\dots\,;fx_n]$$

In Coq, we realize $map\ f\ A$ with a recursive function

```
map : ∀ X Y : Type, (X → Y) → list X → list Y
```

Product

The **product** $A \times B$ of two lists A and B is a list containing all pairs (a, b) such that a is an element of A and b is an element of B:

$$nil \times B := nil$$
$$(a :: A) \times B := map\ (pair\ a)\ B +\!\!+ (A \times B)$$

For instance, $[1\,;2] \times [5\,;6\,;7] = [(1,5)\,;(1,6)\,;(1,7)\,;(2,5)\,;(2,6)\,;(2,7)]$. In Coq, we realize $A \times B$ with a recursive function

```
list_prod : ∀ X Y : Type, list X → list Y → list (X * Y)
```

Structural Induction

We prove a simple fact about length and concatenation of lists using structural induction on lists.

Lemma 8.2.1 $|A +\!\!+ B| = |A| + |B|$

Proof By induction on A.

$A = nil$. We have $|nil +\!\!+ B| = |B| = 0 + |B| = |nil| + |B|$.

$A = x :: A$. We have $|(x :: A) +\!\!+ B| = |x :: (A +\!\!+ B)| = S|A +\!\!+ B| =^{\mathrm{IH}} S(|A| + |B|) = S|A| + |B| = |x :: A| + |B|$. Note the use of the inductive hypothesis. ∎

Use Coq to study every detail of this proof.

Simplification Rules and Tactic *simpl_list*

Figure 8.1 shows simplification rules for list operations the base library provides under the given names. The rules are registered with the tactic *simpl_list*, which will rewrite with these rules as long as this is possible. The rules are applied left to right except for *app_assoc*, which is applied from right to left.

$A +\!\!+ nil = A$	app_nil_r
$A +\!\!+ (B +\!\!+ C) = (A +\!\!+ B) +\!\!+ C$	app_assoc
$rev\ (A +\!\!+ B) = rev\ B +\!\!+ rev\ A$	rev_app_distr
$map\ f\ (A +\!\!+ B) = map\ f\ A +\!\!+ map\ f\ B$	map_app
$\|A +\!\!+ B\| = \|A\| + \|B\|$	app_length
$\|rev\ A\| = \|A\|$	rev_length
$\|map\ f\ A\| = \|A\|$	map_length
$\|A \times B\| = \|A\| \cdot \|B\|$	prod_length
$rev\ (rev\ A) = A$	rev_involutive
$rev\ (A +\!\!+ [x]) = x :: rev\ A$	rev_unit

Figure 8.1: Simplification rules registered with *simpl_list*

Exercise 8.2.2 (Lemma *list_cycle*) Prove $A \neq x :: A$.

Exercise 8.2.3 Prove the simplification rules in Figure 8.1.

Exercise 8.2.4 If you are familiar with functional programming, you will know that the function *rev* defined in the previous section takes quadratic time to reverse a list. This is due to the fact that each recursion step involves an application of the function *app*. One can write a tail-recursive function that reverses lists in linear time. The trick is to move the elements of the main list to a second list passed as an additional argument.

$$revi\ nil\ B := B$$
$$revi\ (x :: A)\ B := revi\ A\ (x :: B)$$

Prove the following correctness properties of *revi* in Coq.

a) $revi\ A\ B = rev\ A +\!\!+ B$

b) $rev\ A = revi\ A\ nil$

Exercise 8.2.5 Define a tail-recursive function *lengthi* and prove $length\ A = lengthi\ A\ nil$.

$$
\begin{aligned}
x \in A + B &\leftrightarrow x \in A \lor x \in B && \text{in_app_iff} \\
x \in rev\ A &\leftrightarrow x \in A && \text{in_rev_iff} \\
x \in map\ f\ A &\leftrightarrow \exists y.\ f y = x \land y \in A && \text{in_map_iff} \\
(a, b) \in A \times B &\leftrightarrow a \in A \land b \in B && \text{in_prod_iff}
\end{aligned}
$$

Figure 8.2: Membership equivalences for list operations

8.3 Membership

We define **list membership** $x \in A$ as follows:

$$
\begin{aligned}
(x \in nil) &:= \bot \\
(x \in y :: A) &:= (x = y \lor x \in A)
\end{aligned}
$$

Using bracket notation, we have

$$
(x \in [x_1 ; \dots ; x_n]) = (x = x_1 \lor \cdots \lor x = x_n \lor \bot)
$$

In Coq, we realize list membership with a recursive predicate

```
In : ∀ X : Type, X → list X → Prop
```

and an accompanying notation.

List membership is similar to membership in finite sets. We can see lists as representations of finite sets. This representation ist not unique since different lists can represent the same set. For instance, $[1;2]$, $[2;1]$, $[1;1;2]$ and $[1;2;2]$ are different lists all representing the set $\{1,2\}$. In contrast to finite sets, lists are ordered structures providing for multiple occurrences of elements.

Figure 8.2 shows membership equivalences for the list operations we defined in the last section. To the right of the equivalences appear the names of the respective lemmas provided by the base library.

Figure 8.3 shows further useful lemmas for list membership the base library provides. The starred lemmas are registered with *auto*.

Exercise 8.3.1 Given the lemmas marked with * in Figure 8.3, *auto 7* can solve the goal $3 \notin nil \land 2 \in A + (1 :: 2 :: B) + C$. Do the proof by hand using the tactic *apply* to understand how this works.

Exercise 8.3.2 Prove the lemmas in Figure 8.3 in Coq.

Exercise 8.3.3 Prove the equivalences in Figure 8.2 in Coq.

$x \in x :: A$	*in_eq
$x \in A \to x \in y :: A$	*in_cons
$x \in A \vee x \in B \to x \in A +\!\!\!+ B$	*in_or_app
$x \notin nil$	*in_nil
$x \in [y] \to x = y$	in_sing
$x \in y :: A \to x \neq y \to x \in A$	in_cons_neq
$x \notin y :: A \to x \neq y \wedge x \notin A$	not_in_cons

Figure 8.3: Membership lemmas

Exercise 8.3.4 Define an inductive predicate *con* satisfying the equivalence $con\ A\ B\ C \leftrightarrow A +\!\!\!+ B = C$.

a) Give the type of *con*.
b) Define *con* with rules.
c) Define *con* with constructors.
d) State the induction lemma for *con*.
e) Prove $con\ A\ B\ C \leftrightarrow A +\!\!\!+ B = C$.

8.4 Inclusion and Equivalence

We define **list inclusion** $A \subseteq B$ and **list equivalence** $A \equiv B$ as follows:

$$A \subseteq B \ := \ \forall x.\ x \in A \to x \in B$$
$$A \equiv B \ := \ A \subseteq B \wedge B \subseteq A$$

We say that A is a **sublist** of B if $A \subseteq B$. Note that two lists are equivalent if and only if they contain the same elements.

In Coq, we realize list inclusion and list equivalence with two predicates

```
incl : ∀ X : Type, list X → list X → Prop
equi : ∀ X : Type, list X → list X → Prop
```

and accompanying notations.

Figure 8.4 shows some inclusion lemmas the base library registers with *auto*. With these lemmas *auto 6* can, for instance, solve the goal

$$\forall A\ B\ C.\ \ A \subseteq B \ \to\ 2 :: A +\!\!\!+ 3 :: A \subseteq C +\!\!\!+ 3 :: 2 :: B$$

The base library also registers the constant *equiv* with *auto* so that it can be unfolded automatically.

$A \subseteq A$	*incl_refl
$A \subseteq B \to A \subseteq x :: B$	*incl_tl
$x \in B \to A \subseteq B \to x :: A \subseteq B$	*incl_cons
$A \subseteq B \to A \subseteq B +\!\!+ C$	*incl_appl
$A \subseteq C \to A \subseteq B +\!\!+ C$	*incl_appr
$A \subseteq C \to B \subseteq C \to A +\!\!+ B \subseteq C$	*incl_app
$nil \subseteq A$	*incl_nil

Figure 8.4: Inclusion lemmas registered with auto

$A \subseteq nil \to A = nil$	incl_nil_eq
$A \subseteq B \to x :: A \subseteq x :: B$	incl_shift
$x :: A \subseteq B \leftrightarrow x \in B \wedge A \subseteq B$	incl_lcons
$x :: A \subseteq [y] \to x = y \wedge A \subseteq [y]$	incl_sing
$x :: A \subseteq x :: B \to x \notin A \to A \subseteq B$	incl_lrcons
$A +\!\!+ B \subseteq C \to A \subseteq C \wedge B \subseteq C$	incl_app_left
$A \subseteq B \to map\ f\ A \subseteq map\ f\ B$	incl_map

Figure 8.5: More inclusion lemmas

Figures 8.5 and 8.6 show further lemmas the base library provides for list inclusion and list equivalence.

The base library registers list equivalence with setoid rewriting so that the basic equality tactics[1] become available for list equivalences. In addition, the following **morphism laws** are registered with setoid rewriting to provide for deep rewriting with list equivalences.

$$\frac{A \equiv A'}{x :: A \equiv x :: A'} \qquad \frac{A \equiv A' \qquad B \equiv B'}{A +\!\!+ B \equiv A' +\!\!+ B'} \qquad \frac{A \equiv A'}{x \in A \leftrightarrow x \in A'} \qquad \frac{A \equiv A' \qquad B \equiv B'}{A \subseteq B \leftrightarrow A' \subseteq B'}$$

Here is a proof rewriting with two equivalences from Figure 8.6.

```
Goal ∀ X (x y : X) A B,  x::A ++ [y] ++ A ++ B ≡ A ++ [y;x] ++ A ++ B.
Proof.
  intros X x y A B. simpl. rewrite equi_swap. rewrite equi_shift at 1. reflexivity.
Qed.
```

[1] *rewrite*, *reflexivity*, *symmetry*, *transitivity*.

$x \in A \to A \equiv x :: A$	equi_push
$x :: A \equiv x :: x :: A$	equi_dup
$x :: y :: A \equiv y :: x :: A$	equi_swap
$x :: A + B \equiv A + x :: B$	equi_shift
$x :: A \equiv A + [x]$	equi_rotate

Figure 8.6: Equivalence lemmas

The base library registers list inclusion as a preorder (i.e., a reflexive and transitive relation) with setoid rewriting. This makes it possible to use the tactics *reflexivity* and *transitivity* for list inclusions. Moreover, list inclusions can be rewritten with list inclusions.

Goal ∀ A B C D : list nat, A ⊆ B → B ⊆ C → C ⊆ D → A ⊆ D.

Proof.

```
intros A B C D F G H. rewrite F. rewrite G. exact H.
```

Qed.

The base library registers the following morphism laws with setoid rewriting to provide for deep rewriting with list inclusions:

$$\frac{A \subseteq A'}{x :: A \subseteq x :: A'} \qquad \frac{A \subseteq A' \qquad B \subseteq B'}{A + B \subseteq A' + B'} \qquad \frac{A \subseteq A'}{x \in A \to x \in A'}$$

Exercise 8.4.1 Decide for each of the following propositions whether it can be shown with the lemmas registered with auto (Figures 8.3 and 8.4).

a) $A \subseteq B \to x :: A \subseteq x :: B$

b) $x \in B \to x \in y :: (A + B)$

c) $x \in y :: A \to x \in y :: (A + B)$

d) $x \in A + (y :: x :: B)$

e) $A + [x] \subseteq x :: A$

f) $(A + B) + C \equiv C + (B + A)$

g) $[x; z] \subseteq [y; x; z]$

Exercise 8.4.2 Prove the lemmas in Figures 8.4, 8.5, and 8.6

Exercise 8.4.3 Prove that set inclusion is a preorder and that list equivalence is an equivalence relation.

Exercise 8.4.4 Prove the morphism laws for set equivalence.

$A \parallel B \leftrightarrow \forall x.\, x \in A \to x \notin B$	disjoint_forall
$A \parallel B \to B \parallel A$	disjoint_symm
$B' \subseteq B \to A \parallel B \to A \parallel B'$	disjoint_incl
$nil \parallel B$	*disjoint_nil
$A \parallel nil$	*disjoint_nil'
$x :: A \parallel B \leftrightarrow x \notin B \wedge A \parallel B$	disjoint_cons
$A +\!\!+ B \parallel C \leftrightarrow A \parallel C \wedge B \parallel C$	disjoint_app

Figure 8.7: Disjointness lemmas

Exercise 8.4.5 Define an inductive predicate *mem* satisfying $mem\ x\ A \leftrightarrow x \in A$.

a) Give the type of *mem*.

b) Define *mem* with rules.

c) Define *mem* with constructors.

d) State the induction lemma for *mem*.

e) Prove $mem\ x\ A \leftrightarrow x \in A$.

8.5 Disjointness

Two lists are **disjoint** if they don't have a common element.

$$A \parallel B \;:=\; \neg\exists x.\, x \in A \wedge x \in B$$

The base library provides the lemmas shown in Figure 8.7. The starred lemmas are registered with *auto*.

Exercise 8.5.1 Prove the lemmas in Figure 8.7.

8.6 Decidability

Figure 8.8 shows four basic decidability laws for lists.[2] We may paraphrase the laws as follows:

1. A list type has decidable equality if the base type has decidable equality.
2. List membership is decidable if the base type has decidable equality.
3. Universal quantification over the members of a list preserves decidability.

[2] The notation $s = t :> T$ says that both sides of the equation have type T.

$$\frac{eq_dec\ X}{eq_dec\ (list\ X)} \quad *\text{list_eq_dec}$$

$$\frac{A : list\ X \qquad eq_dec\ (list\ X)}{dec\ (x \in A)} \quad *\text{list_in_dec}$$

$$\frac{\forall x.\ dec\ (p\,x)}{dec\ (\forall x.\ x \in A \to p\,x)} \quad *\text{list_forall_dec}$$

$$\frac{\forall x.\ dec\ (p\,x)}{dec\ (\exists x.\ x \in A \wedge p\,x)} \quad *\text{list_exists_dec}$$

$$eq_dec\ X \ := \ \forall x\,y : X.\ dec\ (x = y)$$

Figure 8.8: Decidability laws for lists

4. Existential quantification over the members of a list preserves decidability.

Note that the laws for the quantifiers do not require that the base type of the list has decidable equality. All four laws follow by induction on the list A.

The base library provides the laws in Figure 8.8 as lemmas with the names given at the right. For the purpose of proof automation, the base library registers the function *dec* as a so-called **type class**, for which in turn the decidability laws in Figure 8.8 are registered as so-called **instance rules**. The base library also registers instance rules for the logical connectives:

$$\frac{dec\ X \qquad dec\ Y}{dec\ (X \to Y)} \qquad \frac{dec\ X \qquad dec\ Y}{dec\ (X \wedge Y)} \qquad \frac{dec\ X \qquad dec\ Y}{dec\ (X \vee Y)} \qquad \frac{dec\ X}{dec\ (\neg X)}$$

The resulting infrastructure provides for the automation of many decidability proofs.

```
Goal ∀ X A B (p : X → Prop),
  eq_dec X → (∀ x, dec (p x)) → dec (A=B ∨ ∀ x, x ∈ A → ∃ y, y ∈ B ∧ (y ∈ A ∨ ¬ p x)).
Proof. auto. Qed.
```

Additional instance rules for *dec* (or other registered type classes) can be registered with the command

```
Existing Instance L.
```

$$\frac{\forall x.\ dec\ (px)}{\{\, x \mid x \in A \wedge px \,\} + \{\forall x.\ x \in A \to \neg px\}} \quad \text{list_sigma_forall}$$

$$\frac{\forall x.\ dec\ (px) \qquad \neg\forall x.\ x \in A \to \neg px}{\exists x.\ x \in A \wedge px} \quad \text{list_exists_DM}$$

$$\frac{eq_dec\ X \qquad A \not\subseteq B}{\exists x.\ x \in A \wedge x \notin B} \quad \text{list_exists_not_incl}$$

$$\frac{\forall x.\ dec\ (px) \qquad \exists x.\ x \in A \wedge px}{\{\, x \mid x \in A \wedge px \,\}} \quad \text{list_cc}$$

Figure 8.9: Quantifier laws for lists and decidable predicates

where L is the lemma formalizing the rule. It is possible to combine the definition of L with the registration of L by writing the keyword *Instance* rather than the keyword *Lemma*.

```
Instance iff_dec (X Y : Prop) :
  dec X → dec Y → dec (X ↔ Y).
Proof. unfold dec; τto. Qed.
```

Figure 8.9 shows three quantifier laws for lists exploiting decidability assumptions. The second law is a de Morgan-style law for existential list quantification, and the third law is a constructive choice principle for lists. The first law in Figure 8.9 is a basic decision principle for lists from which the other two laws in Figure 8.9 as well as the decidability laws for list quantification in Figure 8.8 can be obtained.

Exercise 8.6.1 Explain why *auto* can prove

```
Goal ∀ X (A B : list X), eq_dec X → dec (A ≡ B)
```

Do the proof not using *auto*.

Exercise 8.6.2 Prove the goal given below not using *auto*. Use the lemmas in Figure 8.8 and the lemmas *or_dec* and *not_dec* from the base library.

```
Goal ∀ X A B (p : X → Prop),
  eq_dec X → (∀ x, dec (p x)) → dec (A=B ∨ ∀ x, x ∈ A → ∃ y, y ∈ B ∧ (y ∈ A ∨ ¬ p x)).
```

Exercise 8.6.3 Prove the decidability laws for lists shown in Figure 8.8. All four laws can be shown by induction on A. Some of the laws require further case analysis. Use the tactics *auto*, *eauto*, *tauto*, *inv*, and *discriminate*.

Exercise 8.6.4 Prove the quantifier laws for lists shown in Figure 8.9. The first law follows by induction on the list A. The other two laws follow from the first law.

8.7 Filtering

The base library provides a function *filter* that for a decidable predicate and a list yields the sublist containing all elements satisfying the predicate.

$$\begin{aligned} \mathit{filter}\ p\ \mathit{nil} &:= \mathit{nil} \\ \mathit{filter}\ p\ (x :: A) &:= \text{if } \ulcorner px \urcorner \text{ then } x :: \mathit{filter}\ p\ A \text{ else } \mathit{filter}\ p\ A \end{aligned}$$

The notation $\ulcorner px \urcorner$ stands for a decision for the proposition px. The type of *filter* is

$$\forall X : \mathit{Type}.\ (X \to \mathit{Prop}) \to (\forall x.\ \mathit{dec}\ (px)) \to \mathit{list}\ X \to \mathit{list}\ X$$

where the first and the third argument are implicit. Coq will attempt to derive the third argument (a decider for p) using the instance rules registered for *dec*. Here is an example:

```
Compute filter (fun x ⇒ 3 ≤ x ≤ 7) [1;2;3;4;5;6;7;8;9].
% [3; 4; 5; 6; 7]
```

The implicit argument of *filter* is determined as

$$\lambda x.\ \mathit{and_dec}\ (\mathit{le_dec}\ 3\ x)\ (\mathit{le_dec}\ x\ 7)$$

The base library realizes the notation $\ulcorner X \urcorner$ with an application *decision* X where *decision* is an identity function for decisions:

```
Definition decision (X : Prop) (D : dec X) : dec X := D.
Arguments decision X {D}.
```

By default, X would be the implicit and D be the explicit argument of **decision**. The default is overwritten with the command *Arguments*. There is a fair chance Coq can derive D since *dec* is a type class with instance rules. Using *decision*, *filter* is defined as follows:

$x \in filter\ p\ A \leftrightarrow x \in A \wedge px$	in_filter_iff
$filter\ p\ A \subseteq A$	filter_incl
$A \subseteq B \to filter\ p\ A \subseteq filter\ p\ B$	filter_mono
$(\forall x.\ x \in A \to px \to qx) \to filter\ p\ A \subseteq filter\ q\ A$	filter_pq_mono
$(\forall x.\ x \in A \to (px \leftrightarrow qx)) \to filter\ p\ A = filter\ q\ A$	filter_pq_eq
$(\forall x.\ x \in A \to px) \to filter\ p\ A = A$	filter_id
$filter\ p\ (filter\ q\ A) = filter\ (\lambda x.\ px \wedge qx)\ A$	filter_and
$filter\ p\ (filter\ q\ A) = filter\ q\ (filter\ p\ A)$	filter_comm
$filter\ p\ (A + B) = filter\ p\ A + filter\ p\ B$	filter_app
$px \to filter\ p\ (x :: A) = x :: filter\ p\ A$	filter_fst
$\neg px \to filter\ p\ (x :: A) = filter\ p\ A$	filter_fst'

Figure 8.10: Lemmas for *filter*

```
Definition filter (X : Type) (p : X → Prop) (p_dec : ∀ x, dec (p x)) : list X → list X :=
  fix f A := match A with
             | nil ⇒ nil
             | x::A' ⇒ if decision (p x) then x :: f A' else f A'
            end.
Arguments filter {X} p {p_dec} A.
```

The implicit argument of *decision* is determined as *p_dec x*.

Figure 8.10 shows lemmas for *filter* the standard library provides.

We discuss the definition of *in_filter_iff*.

```
Lemma in_filter_iff X (p : X → Prop) {p_dec : ∀ x, dec (p x)} x A :
  x ∈ filter p A ↔ x ∈ A ∧ p x.
Proof.
  induction A as [|y A]; simpl.
  - τto.
  - decide (p y) as [B|B]; simpl;
    rewrite IHA; intuition; subst; τto.
Qed.
```

Note that the argument *p_dec* is declared with curly braces. This enforces that *p_dec* is an implicit argument of the lemma. The proof is by induction on A. The induction step does a case analysis on *decision* (px) : *dec* (px)

using the tactic *decide*. The tactic *decide* is defined in the base library as $decide\ X := destruct\ (decision\ X)$.

Exercise 8.7.1 Prove the lemmas in Figure 8.10.

Exercise 8.7.2 (Intersection) Define an intersection function *inter* for lists and prove $x \in inter\ A\ B \leftrightarrow x \in A \land x \in B$.

8.8 Element Removal

We use the notation $A \setminus x$ for the sublist of A obtained by deleting all occurrences of x. Formally, we define **element removal $A \setminus x$** using *filter*:

$$A \setminus x \ := \ filter\ (\lambda y.y \neq x)\ A$$

The base library realizes element removal with a function

$$rem \ : \ \forall X : Type.\ eq_dec\ X \to list\ X \to X \to list\ X$$

whose first and second argument are implicit. The second argument provides a decider for equality on X. The notation $eq_dec\ X$ was defined in Figure 8.8.

Figure 8.11 shows lemmas the base library provides for list removal.

Exercise 8.8.1 Prove the lemmas in Figure 8.11.

8.9 Cardinality

The cardinality $card\ A$ of a list A is the number of different elements in A. For instance,

$$\begin{aligned} card\ [1;2] &= 2 \\ card\ [1;2;1;2;3] &= 3 \end{aligned}$$

Formally, we define cardinality of lists with a recursive function:

$$\begin{aligned} card\ nil &= 0 \\ card\ (x :: A) &= \text{if } \ulcorner x \in A \urcorner \text{ then } card\ A \text{ else } 1 + card\ A \end{aligned}$$

Intuitively, we may say that the function *card* counts only the last occurrence of an element. The base library accommodates cardinality as a function

$$card \ : \ \forall X : Type.\ eq_dec\ X \to list\ X \to \mathbb{N}$$

$x \in A \setminus y \leftrightarrow x \in A \land x \neq y$	in_rem_iff
$x = y \lor x \notin A \to x \notin A \setminus y$	*rem_not_in
$A \setminus x \subseteq A$	*rem_incl
$A \subseteq B \to A \setminus x \subseteq B$	*rem_mono
$A \subseteq B \to x \notin A \to A \subseteq B \setminus x$	*rem_inclr
$A \subseteq B \to (x :: A) \setminus x \subseteq B$	*rem_cons
$x \in B \to A \setminus y \subseteq B \to (x :: A) \setminus y \subseteq B$	*rem_cons'
$x \in A \to B \subseteq A +\!\!+ (B \setminus x)$	*rem_app
$A \setminus x \subseteq C \to B \setminus x \subseteq C \to (A +\!\!+ B) \setminus x \subseteq C$	*rem_app'
$x \in A \setminus y \to x \in A$	*rem_in
$x \neq y \to x \in A \to x \in A \setminus y$	*rem_neq
$x :: A \equiv x :: (A \setminus x)$	rem_equi
$x \in A \to A \equiv x :: (A \setminus x)$	rem_reorder
$(A \setminus x) \setminus y = (A \setminus y) \setminus x$	rem_comm
$(x :: A) \setminus x = A \setminus x$	rem_fst
$x \neq y \to (x :: A) \setminus y = x :: (A \setminus y)$	rem_fst'
$x \notin A \to A \setminus x = A$	rem_id

Figure 8.11: Lemmas for element removal

whose first and second argument are implicit.

Figure 8.11 shows basic cardinality laws the base library provides. The cardinality laws for lists are similar to cardinality laws for finite sets. The laws are useful for proofs that employ size induction based on list cardinality.

Lemma *card_eq* is registered as a morphism law with setoid rewriting.

Exercise 8.9.1 Why do the equations

$$\begin{aligned} card\ nil &= 0 \\ card\ (x :: A) &= 1 + card\ (A \setminus x) \end{aligned}$$

not provide for a recursive definition of the cardinality function in Coq?

$x \in A \to card\ A = 1 + card\ (A \setminus x)$	card_in_rem
$x \notin A \to card\ A = card\ (A \setminus x)$	card_not_in_rem
$A \subseteq B \to card\ A \le card\ B$	card_le
$A \equiv B \to card\ A = card\ B$	card_eq
$A \subseteq B \to card\ A = card\ B \to A \equiv B$	card_equi
$card\ A < card\ B \to \exists x.\ x \in B \land x \notin A$	card_ex
$A \subseteq B \to x \in B \to x \notin A \to card\ A < card\ B$	card_lt
$A \subseteq B \to A \equiv B \lor card\ A < card\ B$	card_or
$card\ (x :: A) = 1 + card\ (A \setminus x)$	card_cons_rem
$card\ A = 0 \to A = nil$	card_0

Figure 8.12: Cardinality laws for lists

Exercise 8.9.2 Prove the lemma *card_in_rem*. Hint: Use induction on A and the lemmas *in_rem_iff* and *rem_id*.

Exercise 8.9.3 Prove the lemma *card_le*. Use induction on A and the lemmas *incl_lcons* and *card_in_rem*.

8.10 Duplicate-Freeness

A list is duplicate-free if it contains no element twice. Formally, we define **duplicate-free lists** with an inductive predicate:

$$dupfree : \quad \forall X : Type.\ list\ X \to Prop$$

$$\frac{}{dupfree\ nil} \qquad\qquad \frac{x \notin A \qquad dupfree\ A}{dupfree\ (x :: A)}$$

The inductive definition of *dupfree* gives us a convenient induction principle for duplicate-free lists. Duplicate-free lists have two important properties:

- The cardinality of a duplicate-free list is the length of the list.
- For every list there exists an equivalent duplicate-free list.

$dec\ (dupfree\ A)$	dupfree_dec
$dupfree\ A \to card\ A = \lvert A \rvert$	dupfree_card
$dupfree\ (x :: A)\ \leftrightarrow\ x \notin A \land dupfree\ A$	dupfree_cons
$A \parallel B \to dupfree\ A \to dupfree\ B \to dupfree\ (A +\!\!+ B)$	dupfree_app
$dupfree\ A \to f$ injective on $A \to dupfree\ (map\ f\ A)$	dupfree_map
$dupfree\ A \to dupfree\ (filter\ p\ A)$	dupfree_filter
$dupfree\ (undup\ A)$	dupfree_undup
$undup\ A \equiv A$	undup_id_equi
$A \equiv B\ \leftrightarrow\ undup\ A \equiv undup\ B$	undup_equi
$A \subseteq B\ \leftrightarrow\ undup\ A \subseteq undup\ B$	undup_incl
$dupfree\ A \to undup\ A = A$	undup_id
$undup\ (undup\ A) = undup\ A$	undup_idempotent

Figure 8.13: Lemmas for duplicate-free lists

We define a function *undup* mapping lists to equivalent duplicate-free lists:

$$undup\ nil\ :=\ nil$$
$$undup\ (x :: A)\ :=\ \text{if } \ulcorner x \in A \urcorner \text{ then } undup\ A \text{ else } x :: undup\ A$$

The base library accommodates *undup* as a function

$$undup :\ \forall X : Type.\ eq_dec\ X \to list\ X \to list\ X$$

whose first and second argument are implicit. Figure 8.13 shows lemmas the base library provides for *undup* and duplicate-free lists.

Exercise 8.10.1 Assume the base type has decidable equality.

a) Prove the lemma *dupfree_card*.
b) Prove the lemma *dupfree_dec*.
c) Prove the lemma *dupfree_undup*.
d) Prove the lemma *undup_id_equi*.

Hint: Use induction on the derivation of *dupfree A* where possible. Otherwise use induction on A.

Exercise 8.10.2 Prove the lemmas *dupfree_app*, *dupfree_map*, and *dupfree_filter*. Hint: Use induction on the derivation of *dupfree A*.

Exercise 8.10.3 Prove that the existence of an undup function implies that base type equality is decidable:

$$\forall X.\ (\forall A : \mathit{list}\ X.\ \{B \mid \mathit{dupfree}\ B \wedge A \equiv B\}) \to \mathit{eq_dec}\ X$$

Hint: Use the lemmas *incl_sing*, *incl_nil_eq*, *equi_dup*, and *dupfree_cons*. First prove a lemma $x :: y :: A \subseteq [z] \to x = y$.

8.11 Power Lists

For every list U we will define a list $\mathcal{P}U$ satisfying the following conditions:

1. Every element of $\mathcal{P}U$ is a sublist of U.
2. Every sublist of U is equivalent to an element of $\mathcal{P}U$.
3. If U is duplicate-free, then every element of $\mathcal{P}U$ is duplicate-free and every sublist of U is equivalent to exactly one element $\mathcal{P}U$.

We define the **power list** $\mathcal{P}U$ as follows:

$$\begin{aligned} \mathcal{P}\ \mathit{nil} &:= [\mathit{nil}] \\ \mathcal{P}\ (x :: A) &:= \mathcal{P}A + \mathit{map}\ (\mathit{cons}\ x)\ (\mathcal{P}A) \end{aligned}$$

The base library accommodates power lists with a function

$$\mathit{power} :\ \forall X : \mathit{Type}.\ \mathit{list}\ X \to \mathit{list}\ (\mathit{list}\ X)$$

We also define a **representation function**

$$\ulcorner A \urcorner^U\ :=\ \mathit{filter}\ (\lambda x. x \in A)\ U$$

satisfying the $\ulcorner A \urcorner^U \in \mathcal{P}U$ and $\ulcorner A \urcorner^U \equiv A$ whenever $A \subseteq U$. The base library accommodates $\ulcorner A \urcorner^U$ with a function

$$\mathit{rep} :\ \forall X : \mathit{Type}.\ \mathit{eq_dec}\ X \to \mathit{list}\ X \to \mathit{list}\ X \to \mathit{list}\ X$$

whose first and second argument are implicit.

Figure 8.14 shows the lemmas the base library provides for power lists and the accompanying representation function.

Exercise 8.11.1 Prove the lemmas for power lists shown in Figure 8.14.

$A \in \mathcal{P}U \to A \subseteq U$	power_incl
$nil \in \mathcal{P}U$	power_nil
$\ulcorner A \urcorner^U \in \mathcal{P}U$	rep_power
$\ulcorner A \urcorner^U \subseteq U$	rep_incl
$A \subseteq U \to x \in A \to x \in \ulcorner A \urcorner^U$	rep_in
$A \subseteq U \to \ulcorner A \urcorner^U \equiv A$	rep_equi
$A \subseteq B \to \ulcorner A \urcorner^U \subseteq \ulcorner B \urcorner^U$	rep_mono
$A \equiv B \to \ulcorner A \urcorner^U = \ulcorner B \urcorner^U$	rep_eq
$A \subseteq U \to B \subseteq U \to \ulcorner A \urcorner^U = \ulcorner B \urcorner^U \to A \equiv B$	rep_injective
$\ulcorner \ulcorner A \urcorner^U \urcorner^U = \ulcorner A \urcorner^U$	rep_idempotent
$dupfree\ U \to dupfree\ (\mathcal{P}U)$	dupfree_power
$A \in \mathcal{P}U \to dupfree\ U \to dupfree\ A$	dupfree_in_power
$dupfree\ U \to A \in \mathcal{P}U \to \ulcorner A \urcorner^U = A$	rep_dupfree
$dupfree\ U \to A \in \mathcal{P}U \to B \in \mathcal{P}U \to A \equiv B \to A = B$	power_extensional

Figure 8.14: Lemmas for power lists

9 Syntactic Unification

We consider a basic problem in computational logic known as syntactic unification: Given two terms, find variable instantiations making the terms syntactically equal. Algorithms for syntactic unification are employed for type inference in Coq and in functional programming languages. Syntactic unification is also used by Coq's tactic interpreter to find instantiations for universally quantified variables of lemmas to be applied with the tactics *apply* and *rewrite*.

We formalize the syntactic unification problem and verify a unification algorithm. Syntactic unification serves us as a basic syntactic theory where we can study the formalization of terms and substitutions. Syntactic unification also provides us with an example of a nontrivial algorithm that in its natural formulation is not structurally recursive. Given that in Coq all recursion is structural, this looks like a problem at first. However, there is a standard technique known as *bounded recursion* that transforms general recursion to structural recursion on a numeric bound provided termination of the general recursion follows with a numeric size function.

9.1 Terms, Substitutions, and Unifiers

We start with an informal discussion of the syntactic unification problem. We consider a minimal version of the problem where **terms** are obtained with **variables** and a single binary operation called **dot**:

$$s, t ::= x \mid s \cdot t$$

Thus a term is either a variable or an ordered pair of two terms. Two terms are **unifiable** if there are instantiations for the variables in the terms such that the terms become equal:

- x and y are unifiable with $x \doteq y$.
- $x \cdot (z \cdot x)$ and $(z \cdot z) \cdot y$ are unifiable with $x \doteq z \cdot z$ and $y \doteq z \cdot (z \cdot z)$.
- x and $x \cdot y$ are not unifiable.

A **unifier** of two terms is a list of variable instantiations making the terms equal. A unifier of two terms is **principal** if the variable instantiations do not involve

unnecessary commitments. The unifiers given above are principal for the respective terms. Given the terms x and y, the instantiation list $[x \doteq z;\ y \doteq z]$ is a non-principal unifier. Our goal is a **unification algorithm** that given two terms decides whether the terms are unifiable and returns a principal unifier if the terms are unifiable.

There is the complication that two terms may have more than one principal unifier. For instance, the terms x and y have two principal unifiers, $x \doteq y$ and $y \doteq x$. The ambiguity can be eliminated by insisting that principal unifiers replace larger variables with smaller variables. We will not impose such a constraint.

It is helpful to see unification as a special form of equation solving: Given an equation $s \doteq t$ between two terms, we are looking for variable instantiations that solve the equation. It will also be helpful to consider list of equations rather than single equations. Under this view, unifiers appear as solutions of equation lists.

We now give formal definitions of terms, equations, and (principal) unifiers in Coq. We start with the definition of variables, terms, and equations.

```
Definition var := nat.

Inductive ter : Type :=
| V : var → ter
| T : ter → ter → ter.

Definition eqn := prod ter ter.
```

The Coq definitions make the informal definitions precise: Variables are numbers, terms are binary trees whose leaves are labelled with variables, and equations are pairs of terms.

Terms are a recursive data structure. Numbers and lists are recursive data structures we are already familiar with. While numbers and lists are obtained with linear recursion, terms are obtained with binary recursion. Inductions on terms will involve two inductive hypotheses, one for each component of a pair.

It is common mathematical practice to always use the same letters for the same type of objects. This way the type of an object is clear from the letter used to denote it. Coq supports this mathematical convention with a command that declares implicit types for identifiers:

```
Implicit Types x y z : var.
Implicit Types s t u v : ter.
Implicit Type e : eqn.
Implicit Types A B C : list eqn.
Implicit Types σ τ : ter → ter.
Implicit Types m n k : nat.
```

The declarations will spare us many type declarations in lemmas and definitions. The conventions automatically extend to variations of the letters like x' or s_2.

When an identifier is used without type declaration, Coq will automatically give it the declared implicit type. Implicit type declarations also make Coq's output more readable since the types of identifiers with implicit types are omitted.

In mathematical mode, we use x to denote both the variable x and the term Vx. Formally, x and $V\,x$ are quite different: While x is a variable, $V\,x$ is a term. Coq will not allow us to write an equation $x = V\,x$.

Next we come to the formal definition of unifiers. The basic idea is that a unifier is a substitution making two terms equal. A substitution can be seen as a function that maps variables to terms. For the purposes of unification finite substitutions that can be represented as equation lists $[x_1 \doteq s_1, \dots, x_n \doteq s_n]$ suffice.

There are different possibilities for the formal representation of substitutions and it will be necessary to work with more than one representation. For the official definition we need to fix one representation. Which representation we choose for the definition will matter a lot for the Coq development. It turns out that an abstract functional representation is most convenient.

We define **substitutions** as functions $\sigma : \mathit{ter} \to \mathit{ter}$ that distribute over dot:

```
Definition subst σ : Prop :=
  ∀ s t, σ (T s t) = T (σ s) (σ t).
```

The distribution property ensures that σs can be obtained from s by replacing every variable x in s with σx.

Exercises 9.6.1 and 9.6.2 are concerned with alternative representations of substitutions.

A substitution σ **unifies** an equation $s \doteq t$ if $\sigma s = \sigma t$. A **unifier** of an equation list is a substitution that unifies every equation in the list:

```
Definition unif σ A : Prop :=
  subst σ ∧
  ∀ s t, (s,t) ∈ A → σ s = σ t.
```

An equation list is **unifiable** if it has a unifier:

```
Definition unifiable A : Prop :=
  ∃ σ, unif σ A.
```

A **principal unifier** of an equation list is a unifier of the list that is subsumed by every unifier of the list:

```
Definition principal_unifier σ A : Prop :=
  unif σ A ∧
  ∀ τ, unif τ A → ∀ s, τ (σ s) = τ s.
```

Exercise 9.1.1 Show that two substitutions agree on all terms if they agree on all variables.

Exercise 9.1.2 A function $f : X \to X$ is **idempotent** if $f(fx) = fx$ for every x in X. Show that every principal unifier is idempotent.

Exercise 9.1.3 Prove the following facts about unification:

a) $\mathit{unif}\ \sigma\ (y \doteq s :: A) \ \leftrightarrow\ \sigma s = \sigma t \wedge \mathit{unif}\ \sigma\ A$

b) $\mathit{unif}\ \sigma\ (A + B) \ \leftrightarrow\ \mathit{unif}\ \sigma\ A \wedge \mathit{unif}\ \sigma\ B$

Exercise 9.1.4 Prove that an equation list is non-unifiable if some sublist is non-unifiable.

9.2 Solved Equation Lists

A list of equations may be solved by transforming it to a solved form using unifier-preserving rules. An equation list is *solved* if it has the form

$$x_1 \doteq s_1, \dots, x_n \doteq s_n$$

where the variables $x_1, \dots, x_n$ are distinct and, for all $i \in \{1, \dots, n\}$, the variable x_i does not appear in $s_1, \dots, s_i$. Here is an example of a solved equation list:

$$[x_1 \doteq x_0 \cdot x_0;\ x_2 \doteq x_1 \cdot x_1;\ x_3 \doteq x_2 \cdot x_2]$$

For a solved list we can always obtain a principal unifier. The above list has a unique principal unifier σ, which satisfies the equations

$$\begin{aligned} \sigma x_1 &= x_0 \cdot x_0 \\ \sigma x_2 &= \sigma x_1 \cdot \sigma x_1 \\ \sigma x_3 &= \sigma x_2 \cdot \sigma x_2 \\ \sigma x &= x \qquad \text{if } x \notin \{x_1, x_2, x_3\} \end{aligned}$$

The formal definition of solved equation lists requires a number of auxiliary definitions. We start with the notations $\mathcal{V}s$ and $\mathcal{V}A$, which stand for lists containing exactly the variables occurring in s and A, respectively. We realize the notations with two recursive functions:

$$\begin{aligned} \mathcal{V}x &:= [x] & \mathcal{V}\,\mathit{nil} &:= \mathit{nil} \\ \mathcal{V}(s \cdot t) &:= \mathcal{V}s + \mathcal{V}t & \mathcal{V}(s \doteq t :: A) &:= \mathcal{V}s + \mathcal{V}t + \mathcal{V}A \end{aligned}$$

For the definition of solved equation lists, we also need the **domain** $\mathcal{D}A$ of an equation list A. If $A = [x_1 \doteq t_1; \dots; x_n \doteq t_n]$, then $\mathcal{D}A = [x_1; \dots; x_n]$. If A is

of a different form, we do not care about $\mathcal{D}A$. Formally, we define $\mathcal{D}A$ for all equation lists A. We use the following definition:

$$\begin{aligned}
\mathcal{D}\,nil &:= nil \\
\mathcal{D}\,(x \doteq t :: A) &:= x :: \mathcal{D}A \\
\mathcal{D}\,(s \doteq t :: A) &:= nil \qquad \text{if } s \text{ is not a variable}
\end{aligned}$$

We can now define **solved** equation lists with an inductive predicate *solved*:

$$\frac{}{solved\ nil} \qquad \frac{x \notin \mathcal{V}s \qquad x \notin \mathcal{D}A \qquad \mathcal{V}s \parallel \mathcal{D}A \qquad solved\ A}{solved\ (x \doteq s :: A)}$$

Next we need an operation s^x_t that in a term s replaces every occurrence of the variable x with a term t. We speak of **variable replacement**. We will also need variable replacement for lists. Formally, we define s^x_t and A^x_t with two recursive functions:

$$\begin{aligned}
y^x_t &:= \text{if } y = x \text{ then } t \text{ else } y & \qquad nil^x_t &:= nil \\
(s_1 \cdot s_2)^x_t &:= {s_1}^x_t \cdot {s_2}^x_t & \qquad (u \doteq v :: A)^x_t &:= u^x_t \doteq v^x_t :: A^x_t
\end{aligned}$$

Next we define a function φ that yields a principal unifier for every solved equation list A. Formally, we define φ on all equation lists:

$$\begin{aligned}
\varphi\ nil\ s &:= s \\
\varphi\ (x \doteq t :: A)\ s &:= (\varphi\ A\ s)^x_t \\
\varphi\ (u \doteq v :: A)\ s &:= s \qquad \text{if } u \text{ is not a variable}
\end{aligned}$$

Lemma 9.2.1 Let A be solved. Then φA is a principal unifier of A.

A **bad equation** is an equation of the form $x \doteq s$ where $x \neq s$ and $x \in \mathcal{V}s$.

Lemma 9.2.2 No equation list containing a bad equation is unifiable.

We leave the proofs of the lemmas as exercises.

Exercise 9.2.3 Prove the following facts about variable replacement.

a) If $x \notin \mathcal{V}s$, then $s^x_t = s$.
b) If $x \notin \mathcal{V}A$, then $A^x_t = A$.
c) If $x \notin \mathcal{D}A$, then $\mathcal{D}(A^x_t) = \mathcal{D}A$.
d) If σ is a substitution such that $\sigma x = \sigma t$, then $\sigma(s^x_t) = \sigma s$.
e) $\lambda s.\, s^x_t$ is a substitution.

Exercise 9.2.4 Prove the following facts about φ:

a) $\varphi\,A$ is a substitution.

b) If $\mathcal{D}A \parallel \mathcal{V}s$, then $\varphi\,A\,s = s$.

c) If A is solved, then $\varphi\,A$ is a unifier of A.

d) If σ is a unifier of A, then $\sigma\,(\varphi\,A\,s) = \sigma s$.

e) If A is solved, then $\varphi\,A$ is a principal unifier of A.

Exercise 9.2.5 Prove the bad equation lemma 9.2.2. Hint: Define a function $\mathit{size} : \mathit{ter} \to \mathbb{N}$ such that $\mathit{size}\; s < \mathit{size}\,(s \cdot t)$ and proceed by proving the following facts:

a) If $x \in \mathcal{V}s$ and σ is a substitution, then $\mathit{size}\,(\sigma x) \le \mathit{size}\,(\sigma s)$.

b) No bad equation is unifiable.

Exercise 9.2.6 Prove the following facts about variables:

a) $\mathcal{D}A \subseteq \mathcal{A}$

b) $\mathcal{V}(A +\!\!+ B) = \mathcal{V}A +\!\!+ \mathcal{V}B$

c) $s \doteq t \in A \to \mathcal{V}s \subseteq \mathcal{V}A \wedge \mathcal{V}t \subseteq \mathcal{V}A$

d) $A \subseteq B \to \mathcal{V}A \subseteq \mathcal{V}B$

Exercise 9.2.7 Write a function $\mathit{gen} : \mathbb{N} \to \mathit{ter}$ for which you can prove that $\mathit{gen}\; m$ and $\mathit{gen}\; n$ are non-unifiable if m and n are different.

Exercise 9.2.8 Prove that the concatenation $A +\!\!+ B$ of two solved lists A and B is solved if $\mathcal{V}A$ and $\mathcal{D}B$ are disjoint.

9.3 Unification Rules

Every unifiable equation list can be transformed with the so-called unification rules to a solved equation list having the same unifiers as the initial list. We prepare this result with the definitions of an equivalence relation $A \approx B$ (**unifier equivalence**) and a preorder $A \triangleright B$ (**refinement**):

$$A \approx B \;:=\; \forall \sigma.\; \mathit{unif}\;\sigma\;A \leftrightarrow \mathit{unif}\;\sigma\;B$$

$$A \triangleright B \;:=\; \mathcal{V}B \subseteq \mathcal{V}A \wedge A \approx B$$

We say that B is a **refinement** of A if $A \triangleright B$.

Lemma 9.3.1 Refinement of equation lists is a preorder compatible with cons, concatenation, and unification.

1. $A \rhd A$.
2. If $A \rhd B$ and $B \rhd C$, then $A \rhd C$.
3. If $A \rhd A'$, then $x :: A \rhd x :: A'$.
4. If $A \rhd A'$ and $B \rhd B'$, then $A +\!\!+ B \rhd A' +\!\!+ B'$.
5. If $A \rhd B$, then $\mathit{unif}\ \sigma\ A \leftrightarrow \mathit{unif}\ \sigma\ B$.

Lemma 9.3.2 (Unification Rules)

1. **Deletion** $[s \doteq s] \rhd \mathit{nil}$.
2. **Swap** $[s \doteq t] \rhd [t \doteq s]$.
3. **Decomposition** $[s_1 \cdot s_2 \doteq t_1 \cdot t_2] \rhd [s_1 \doteq t_1;\ s_2 \doteq t_2]$.
4. **Replacement** $x \doteq t :: A \rhd x \doteq t :: A^x_t$.

The **unification rules** are obtained as the operational readings of the facts about refinement stated in Lemma 9.3.2:

1. Trivial equations may be deleted.
2. The two sides of an equation may be swapped.
3. An equation $s_1 \cdot s_2 \doteq t_1 \cdot t_2$ may be replaced by $s_1 \doteq t_1$ and $s_2 \doteq t_2$.
4. Given an equation $x \doteq s$, the variable x may be replaced with s in other equations.

We say that a solved equation list B is a **solved form** for an equation list A if $A \rhd B$. We will show that the unification rules suffice to refine every unifiable equation list into a solved form. We will also show that the unification rules are strong enough to refine every non-unifiable equation list into a list containing a bad equation.

Exercise 9.3.3 Prove the facts stated by Lemma 9.3.1.

Exercise 9.3.4 Prove the correctness of the deletion, swap, and decomposition rule (i.e., the first three facts stated by Lemma 9.3.2).

Exercise 9.3.5 Prove the correctness of the replacement rule (i.e., the last fact stated by Lemma 9.3.2). Proceed by proving the following facts in the order stated.

a) If σ is a substitution such that $\sigma x = \sigma t$, then $\sigma(s^x_t) = \sigma s$.
b) If $\sigma x = \sigma t$, then $\mathit{unif}\ \sigma\ A \leftrightarrow \mathit{unif}\ \sigma\ (A^x_t)$.
c) $x \doteq t :: A \approx x \doteq t :: A^x_t$
d) $\mathcal{V}(s^x_t) \subseteq \mathcal{V}s +\!\!+ \mathcal{V}t$
e) $\mathcal{V}(A^x_t) \subseteq \mathcal{V}A +\!\!+ \mathcal{V}t$

f) $x \doteq t :: A \;\triangleright\; x \doteq t :: A^x_t$

Exercise 9.3.6 Prove the following fact about principal unifiers: If $A \approx B$ and σ is a principal unifier of A, then σ is a principal unifier of B.

Exercise 9.3.7 Give a solved equation list that has more than one principal unifier.

Exercise 9.3.8 (Confrontation Rule)
Prove $[x \doteq s_1 \cdot s_2;\ x \doteq t_1 \cdot t_2] \;\triangleright\; [x \doteq s_1 \cdot s_2;\ s_1 \doteq t_1;\ s_2 \doteq t_2]$. The operational reading of this fact yields the so-called *confrontation rule*. The confrontation rule can often be used in place of the replacement rule when an equation list is transformed to solved form. In contrast to the replacement rule it has the virtue that it introduces only terms that are subterms of the original terms. This fact matters for efficient unification algorithms.

9.4 Presolved Equation Lists

An equation list A is **presolved** if it is either empty or starts with an equation of the form $x \doteq s$ where $x \neq s$. Every equation list can be refined into a presolved equation list using the unification rules for deletion, swapping, and decomposition of equations. The refinement can be realized with two recursive functions:

$$
\begin{aligned}
pre'\ s\ t \;:=\;& \text{if } s = t \text{ then } nil \text{ else} \\
& \text{match } s, t \text{ with} \\
& \mid x, _ \;\Rightarrow\; [s \doteq t] \\
& \mid _, x \;\Rightarrow\; [t \doteq s] \\
& \mid s_1 \cdot s_2,\ t_1 \cdot t_2 \;\Rightarrow\; pre'\ s_1\ t_1 +\!\!+ pre'\ s_2\ t_2 \\[1em]
pre\ nil \;:=\;& nil \\
pre\ (s \doteq t :: A) \;:=\;& pre'\ s\ t +\!\!+ pre\ A
\end{aligned}
$$

Lemma 9.4.1

1. $[s \doteq t] \;\triangleright\; pre'\ s\ t$ and $pre'\ s\ t$ is presolved.
2. $A \;\triangleright\; pre\ A$ and $pre\ A$ is presolved.

Exercise 9.4.2 Prove Lemma 9.4.1.

9.5 Unification Algorithm

Our goal is a function *solve* satisfying the following specification:

```
Theorem solve_correct C :
  match solve C with
    | Some A ⇒ C ⊳ A ∧ solved A
    | None ⇒ ¬ unifiable C
  end
```

Our starting point is the presolver from the previous section. We distinguish three cases:

1. *pre* C is empty. Then *nil* is a solved form for C and we are done.
2. *pre* C starts with a bad equation. Then C is not unifiable and we are done.
3. *pre* $C = x \doteq t :: D$ and $x \notin \mathcal{V}t$. In this case we collect $x \doteq t$ as first equation of a possible solved form for C. The correctness of *pre* and the replacement rule give us

   $$C \ \triangleright \ x \doteq t :: D \ \triangleright \ x \doteq t :: D^x_t$$

   Note that the list D^x_t contains one variable less than D since the variable replacement eliminates the variable x. We speak of an **elimination step**.

We continue recursively by eliminating variables as long as this is possible. Things can be arranged such that the equations $x \doteq t$ used for variable elimination yield a solved form (see Exercise 9.2.8).

The outlined idea can be realized with a function *solveE* satisfying the following specification:[1]

```
Lemma solveE_correct A B C :
  C ⊳ A ++ B →
  solved A →
  𝒟A ∥ 𝒱B →
  match solveE A B with
    | Some D ⇒ C ⊳ D ∧ solved D
    | None ⇒ ¬ unifiable C
  end
```

Note that the lemma lists the necessary preconditions for *solveE*. Given *solveE*, we define

$$solve\ C \ := \ solveE\ nil\ C$$

and prove *solve_correct* using *solveE_correct*.

The obvious way to write the function *solveE* is by size recursion on the cardinality of $\mathcal{V}B$. However, Coq admits only structural recursion. The trick now is to

[1] You may wonder why the correctness lemma is not formulated more compactly without the variable C and the accompanying precondition. The reason is that the presence of C simplifies the inductive proof. See Exercise 9.5.4 for a discussion of the issue.

introduce an additional argument n serving as a bound for the recursion depth. The recursion can then be realized as structural recursion on the bound n. One speaks of **bounded recursion**. We realize the idea with a function *solveN* satisfying the following specification:

```
Lemma solveN_correct A B C n :
  C ▷ A ++ B →
  solved A →
  𝒟 A ∥ 𝒱 B →
  card (𝒱 B) < n →
  match solveN n A B with
    | Some D ⇒ C ▷ D ∧ solved D
    | None ⇒ ¬ unifiable C
  end
```

Note that the precondition $card(\mathcal{V}B) < n$ requires that the bound n is larger than the recursion depth needed for B. Given *solveN*, we define

$$solveE\ A\ B\ :=\ solveE\ (1 + card(\mathcal{V}B))\ A\ B$$

and prove *solveE_correct* using *solveN_correct*.

It remains to define *solveN* and prove *solveN_correct*. We define *solveN* refining the initial idea for *solve* with bounded recursion.

```
Fixpoint solveN n A B : option (list eqn) :=
  match n, pre B with
    | O, _ ⇒ None
    | S n', x ≐ t :: C ⇒ if ⌜x ∈ 𝒱t⌝ then None else solveN n' (x ≐ t :: A) (C^x_t)
    | S n', _ ⇒ Some A
 end
```

The proof of *solveN_correct* is pleasant and leads to subgoals expressing proof obligations one would expect from an informal correctness argument for *solve*. One first reverts A and B and then continues by induction on n. The base case is trivial. For the inductive case one simulates the case analysis of the function *solveN*. For the recursion step one applies the inductive hypothesis, which produces subgoals for the preconditions.

Exercise 9.5.1 Define the functions *solveE* and *solve* and prove their correctness lemmas (based on *solveN* and *solveN_correct*).

Exercise 9.5.2 Prove that an equation list either has a solved form or is non-unifiable.

Exercise 9.5.3 Prove that an equation list either has a principal unifier or is non-unifiable.

Exercise 9.5.4 If you look at the correctness lemmas for *solveE* and *solveN*, you may notice that the lemmas can be formulated without the list *C* and the precondition for *C*. The alternative formulation of the correctness lemma for *solveN* looks as follows:

```
Lemma solveN_correct' A B n :
  solved A →
  𝒟 A ∥ 𝒱 B →
  card (𝒱 B) < n →
  match solveN n A B with
    | Some D ⇒ A ++ B ▷ D ∧ solved D
    | None ⇒ ¬ unifiable (A ++ B)
  end
```

We have chosen the less compact formulation of the correctness lemma with the variable *C* since the presence of *C* considerably simplifies the proof. In the formulation with *C* the inductive hypothesis applies directly to the claim for the recursive call and yields subgoals for the preconditions. In the formulation without *C*, the inductive hypothesis does not apply directly and needs to be transformed using forward reasoning.

Try to prove the lemma *solveN_correct′* to understand the issue.

Exercise 9.5.5 From the correctness theorem for the function *solve* it follows that every unifiable equation list has a solved form. Try to prove this fact without using the function *solve* and its variants (using the function *pre* is fine, however). Hint: Use size induction to prove a lemma compensating for *solveN_correct*. Much of the proof script for *solveN_correct* can be reused.

Mathematically, one may argue that there is no need for an explicit function *solveN* since the proof of the desired existence lemma (unifiable equation lists have solved forms) can be carried out with size induction. One may also argue that defining the function *solveN* explicitly makes the proof more transparent.

Exercise 9.5.6 Extend the Coq development of syntactic unification to terms with constants.

9.6 Alternative Representations

In this section we explore alternative representations for substitutions and a more explicit solved form.

Exercise 9.6.1 (Representing Substitutions as Functions *var* → *ter*)
It is natural to see substitutions as functions from variables to terms. In fact,

every function from variables to terms represents a substitution, and every substitution can be represented as a function from variables to terms. Our representation of substitutions as functions from terms to terms is less direct since the functions representing substitutions need to be filtered out with the predicate *subst*.

Define a function $hat : (var \to ter) \to ter \to ter$ for which you can prove the following statements.

a) $subst\ (hat\ f)$

b) $hat\ f\ x = fx$

c) $hat\ (\lambda x.x)\ s = s$

d) $subst\ \sigma \to hat\ (\lambda x.\sigma x)\ s = \sigma s$

e) $s^x_t = hat\ (\lambda z.\ \text{if}\ \ulcorner z = x \urcorner\ \text{then}\ t\ \text{else}\ z)\ s$

Exercise 9.6.2 (Representing Substitutions as Lists)

We are mostly interested in finite substitutions σ that can be represented with an equation list $A = [x_1 \doteq s_1; \ldots; x_n \doteq s_n]$ such that

$$\sigma x = \begin{cases} s & \text{if } x \doteq s \in A \\ x & \text{otherwise} \end{cases}$$

Define a function $sub : list\ eqn \to var \to ter$ for which you can prove the following:

a) $sub\ (x \doteq s :: A)\ x = s$

b) $sub\ (x \doteq s :: A)\ y = sub\ A\ y\ \text{ if } x \neq y$

c) $sub\ A\ x = x\ \text{ if } x \notin \mathcal{D}A$

d) $hat\ (sub\ (x \doteq s :: A))\ t = hat\ (sub\ A)\ t\ \text{ if } x \neq \mathcal{V}t$

e) $hat\ (sub\ A)\ s = s\ \text{ if } \mathcal{D}A \parallel \mathcal{V}s$

f) $s^x_t = hat\ (sub\ [x \doteq t])\ s$

Exercise 9.6.3 (Fully Solved Equation Lists)

Consider the following inductive definition of *fully solved* equation lists:

$$\frac{}{fsolved\ nil} \qquad \frac{x \notin \mathcal{V}s \quad x \notin \mathcal{V}A \quad \mathcal{V}s \parallel \mathcal{D}A \quad fsolved\ A}{fsolved\ (x \doteq s :: A)}$$

a) Show that every fully solved equation list is solved.

b) Give a solved equation list that is not fully solved.

c) Convince yourself that every fully solved refinement of the solved list

$$A_n = [x_1 \doteq x_0 \cdot x_0;\ x_2 \doteq x_1 \cdot x_1;\ \ldots\ ;\ x_{n+1} \doteq x_n \cdot x_n]$$

is exponentially larger than A_n.

d) Let A be fully solved. Prove that *hat* (*sub* A) is a principal unifier of A.

e) Let A be fully solved and $x \doteq s \in A$. Prove that $\mathcal{D}A$ and $\mathcal{V}s$ are disjoint.

f) Define a function *unfold* that refines solved lists into fully solved lists. Prove the following for your function *unfold*:

   i) If A is solved, then $A \triangleright$ *unfold* A.

   ii) If A is solved, then $\mathcal{D}(\mathit{unfold}\ A) = \mathcal{D}A$.

   iii) If A is fully solved, $x \notin \mathcal{D}A$, and $\mathcal{D}A \parallel \mathcal{V}s$, then A^x_s is fully solved.

   iv) If A is solved, then *unfold* A is fully solved.

Exercise 9.6.4 (Exponential Running Time)

Consider the solved equation list A_n from Exercise 9.6.3. Convince yourself that the function *solve* from the previous section yields a fully solved refinement of A_n that is exponentially larger than A_n. This suggests that *solve* has exponential running time.

Exercise 9.6.5 (Extensional Representation of Finite Substitutions)

Two equation lists A and B are **substitution equivalent** if *sub* $A\ x$ = *sub* $B\ x$ for every variable x. Define a predicate *fsubst* on equation lists that fixes unique normal forms for substitution equivalence.

a) Prove that two equation lists satisfying *fsubst* are equal if they are substitution equivalent.

b) Define a function that for every equation list yields a substitution equivalent equation list satisfying *fsubst*.

The proofs for this exercise require considerable effort and provide for an interesting project. Auxiliary lemmas will be needed.

9.7 Notes

Syntactic unification was identified by Robinson [17] in 1965 as base algorithm of his resolution calculus designed for automated theorem proving. Syntactic unification was rediscovered several times for other applications, for instance by Knuth and Bendix [11] for critical pair analysis of term rewriting systems, and by Milner [14] for polymorphic type inference. The unification rules and the view of syntactic unification as equation solving appear in the work of Martelli and Montanari [13].

Papers on syntactic unification include Martelli and Montanari [13], Baader and Snyder [1], Jaffar and Lassez [12], and Eder [2]. Baader and Snyder [1] also cover unification modulo equational theories.

Robinson's initial unification algorithm [17] has exponential runtime (as does the naive algorithm developed in this chapter). Martelli and Montanari [13] present a quasi-linear unification algorithm. Unifiability can be decided in linear time [15].

Paulson [16] reports about an early formal verification of the unification algorithm in LCF. Ruiz-Reina et al [18] have formally verified an efficient quadratic unfication algorithm in ACL2.

10 Propositional Entailment

Propositional logic is a logical system for propositional formulas. Propositional formulas consist of atomic propositions (e.g., propositional variables, $\bot$ and $\top$) and are closed under logical connectives (e.g., implication, conjunction and disjunction). In this chapter we will restrict ourselves to propositional logic with propositional variables and $\bot$ and closed under implication. The systems and results can be extended to include other connectives, but we leave such extensions to exercises.

We will first study the type of propositional formulas. Next we consider the general notion of an entailment relation and properties an entailment relation may have. We then define a particular entailment relation by giving a natural deduction style proof system for intuitionistic propositional logic. The natural deduction system will correspond closely to the proof system in Coq. We next consider a classical natural deduction style proof system. We will prove a result of Glivenko: a propositional formula is classically provable if and only if its double negation is intuitionistically provable. We will finally consider a Hilbert style proof system and prove the equivalence of the natural deduction system and the Hilbert system.[1]

10.1 Propositional Formulas

We now define **(propositional) formulas** given by the following grammar where x ranges over variables and s and t range over propositional formulas.

$$s, t \;::=\; x \mid \bot \mid s \to t$$

We would like to have infinitely many variables. In addition, we would like equality of variables to be decidable. A natural way to ensure both is to use natural numbers to represent variables. We use *var* be the type of variables (which is defined to be *nat*). After fixing the representation of variables, we can represent propositional formulas in Coq using an inductive type in the usual way. Essentially formulas are binary trees where each node with children is an implication $s \to t$ (with a child for s and a child for t) and each leaf is either $\bot$ or a variable x.

[1] Proof systems are often called "calculi." In this context, "calculus" is a synonym for "system."

Just as in Coq, we consider $\neg s$ as meaning $s \to \bot$. Note that equality of formulas is decidable.

We can compute a list of the variables which occur in a formula as follows. The function is defined by recursion over formulas. Note that when computing the variables in $s \to t$ we make two recursive calls: one for s and one for t. We done the list of variables of a formula s by $\mathcal{V}s$.

We can map formulas to booleans in an obvious way. To handle variables we need assignments. An assignment is a function from var to bool. We define when an assignment φ **satisfies** a formula s by recursion on s as follows:

1. φ satisfies x if $\varphi x = true$.
2. φ satisfies $s_1 \to s_2$ if φ satisfies s_1 implies φ satisfies s_2.
3. No assignment satisfies $\bot$.

The satisfies predicate can be easily proven decidable by induction on the formula.

Exercise 10.1.1 Prove the following goal.

```
Goal ∃ f: assn, satis f (Not (Imp (Var 0) (Var 1))).
```

Exercise 10.1.2 Prove the following goal.

```
Goal ∀ f: assn, ∀ s:form, satis f (Imp (Not (Not s)) s).
```

Exercise 10.1.3 Prove the following goal.

```
Goal ∀ f: assn, ∀ s t:form, satis f (Imp (Imp (Imp s t) s) s).
```

10.2 Structural Properties of Entailment Relations

In this section we will consider the general notion of an entailment relation and define structural properties an entailment relation may satisfy. When an entailment relation holds for a list A of assumed formulas and a formula s, the intention is that the formula s is a logical consequence of the assumptions in A. For the entailment relations considered in this section, we need not commit to propositional formulas, but can work with a general type F instead of the type of propositional formulas.

Suppose F is a type. An **entailment relation for** F is a predicate of type $list\ F \to F \to Prop$. Suppose E is an entailment relation for F. We write $A \vdash s$ when a given entailment relation holds between a list A of formulas and a formula s. The symbol $\vdash$ is a “turnstile.” We write $A \nvdash s$ to mean the negation of $A \vdash s$. When A is empty, we may write $\vdash s$.

Several properties of an entailment relation can be stated without using special properties of the type F of formulas. These are called **structural properties**. We define four structural properties in this section.

We say an entailment relation is **monotone** if $A' \vdash s$ holds whenever $A \vdash s$ holds and $A \subseteq A'$. That is, if s is a logical consequence of the assumptions in A, then it remains a logical consequence if we add more assumptions to A.

We say the entailment relation is **reflexive** if $A \vdash s$ whenever $s \in A$. That is, each assumed formula in A is a logical consequence of A.

We say the entailment relation **satisfies cut** if $A \vdash s$ and $A, s \vdash t$ imply $A \vdash t$. That is, if we extend A with logical consequences of A, then we do not obtain new logical consequences.

Finally, we say the entailment relation **is consistent** if there is some s such that $\nvdash s$.

We give one simple example of an entailment relation. The exercises at the end of this section give a few more examples.

We will define an entailment relation for *Prop*. That is, we will give a predicate of type *list Prop* $\rightarrow$ *Prop* $\rightarrow$ *Prop*. We start by recursively defining a function which forms a conjunction from a list of propositions.

```
Fixpoint andlist (A:list Prop) : Prop :=
match A with
| P::A' ⇒ P ∧ andlist A'
| nil ⇒ ⊤
end.
```

The following property can easily be proven by induction on the list A.

```
Lemma andlistEq (A:list Prop) : andlist A ↔ ∀ s, s ∈ A → s.
```

Consider the entailment relation $A \vdash s$ for *Prop* defined by *andlist* $A \rightarrow s$. The four structural properties can be easily verified using the lemma.

The only entailment relations we will consider after the exercises below will be for propositional formulas. That is, after this section we will only consider entailment relations for the type *form*.

Exercise 10.2.1 Consider the entailment relation $A \vdash s$ for *bool* which holds if s either s is *true* or if *false* $\in A$. Prove the four structural properties hold.

```
Goal
  let E : list bool → bool → Prop := fun A s ⇒ if s then ⊤ else false ∈ A in
  Reflexivity E ∧ Monotonicity E ∧ Cut E ∧ Consistency E.
```

Exercise 10.2.2 Let X be an inhabited type. Consider the entailment relation $A \vdash s$ for $X \rightarrow$ *Prop* defined by $\forall x : X, (\forall P, P \in A \rightarrow Px) \rightarrow sx$. Prove the four structural properties hold.

```
Goal ∀ X:Type, inhabited X →
  let E : list (X → Prop) → (X → Prop) → Prop
    := fun A s ⇒ forall x:X, (∀ P, P ∈ A → P x) → s x in
  Reflexivity E ∧ Monotonicity E ∧ Cut E ∧ Consistency E.
```

Exercise 10.2.3 Consider the entailment relation $A \vdash s$ for *nat* defined by $\exists n. n \in A \land s \leq n$. Prove the four structural properties hold.

```
Goal
  let E : list nat → nat → Prop := fun A s ⇒ exists n, n ∈ A ∧ s ≤ n in
  Reflexivity E ∧ Monotonicity E ∧ Cut E ∧ Consistency E.
```

Exercise 10.2.4 Let F be a type. Consider the entailment relation defined by list membership. Without extra assumptions, only three of the four structural properties hold. Determine which of the three hold and prove them. For the one which cannot be proven, choose a type F and prove the property fails for list membership on your chosen type F.

10.3 Logical Properties of Entailment Relations

A **context** is a list of formulas.

```
Definition context := list form.
```

An entailment relation for formulas is a predicate of type *context* → *form* → *Prop*. For entailment relations for formulas we can now define the following **logical properties**. The reader should compare these to the structural properties defined for generic entailment relations in the previous section. From now on when we refer to an entailment relation, we will be referring to an entailment relation for formulas.

We say an entailment relation **satisfies the characteristic property of** $\rightarrow$ if $A \vdash s \rightarrow t$ is equivalent to $A, s \vdash t$.

We say an entailment relation **satisfies the characteristic property of** $\perp$ if $A \vdash \perp$ is equivalent to forall s, $A \vdash s$.

Suppose we are working with a reflexive entailment relation which satisfies the characteristic properties of $\rightarrow$ and $\perp$. Let s and t be formulas. We can prove $\vdash s \rightarrow \neg s \rightarrow t$ as follows. First, using the characteristic property of $\rightarrow$ twice it is enough to prove $s, \neg s \vdash t$. Using the characteristic property of $\perp$ it is enough to prove $s, \neg s \vdash \perp$. Using the characteristic property of $\rightarrow$ in the other direction it is enough to prove $\neg s \vdash s \rightarrow \perp$. We know $\neg s \vdash s \rightarrow \perp$ by reflexivity.

We can use boolean assignments to define the following entailment relation which we call **boolean semantic consequence**: $A \vDash s$ holds if for every assignment φ, if φ satisfies every element of A, then φ satisfies s.

Exercise 10.3.1 Prove $\vDash$ (boolean semantic consequence) satisfies all the structural properties defined in the previous section and the two logical properties defined in this section.

Exercise 10.3.2 Prove that if E satisfies the characteristic properties of $\rightarrow$ and $\bot$ also satisfies the following characteristic property of $\neg$.

```
Goal ∀ E, CharImp E → CharFal E → ∀ A s, E A (Not s) ↔ ∀ t, E (s::A) t.
```

Exercise 10.3.3 Prove the following.

```
Goal ∀ E, Cut E → CharImp E → ∀ A s t, E A (Imp s t) → E A s → E A t.
```

Exercise 10.3.4 Prove that if an entailment relation is reflexive and satisfies the characteristic property of $\rightarrow$, then it is nonempty. In particular, prove there is a formula s such that $\vdash s$.

```
Lemma Reflexivity_CharImp_nonempty E :
 Reflexivity E → CharImp E → ∃ s, E nil s.
```

Exercise 10.3.5 We say a formula is **closed** if it contains no variables. We can define this in Coq as follows.

```
Inductive closed : form → Prop :=
| closedFal : closed Fal
| closedImp s t : closed s → closed t → closed (Imp s t).
```

Suppose $\vdash$ is a reflexive entailment relation satisfying cut and the characteristic properties of $\rightarrow$ and $\bot$. Prove for all closed formulas s we either have $A \vdash s$ (for every context A) or $A \vdash \neg s$ (for every context A).

```
Lemma ReflexivityCutChar_closed_or E s :
  Reflexivity E → Cut E → CharImp E → CharFal E →
  closed s → (∀ A, E A s) ∨ (∀ A, E A (Not s)).
```

10.4 Variables and Substitutions

Variables are intended to be placeholders which can, for example, be substituted with arbitrary formulas. A **substitution** is a mapping σ from variables to formulas. By recursion on formulas we can define a substitution operation lifting the action of σ on variables to all formulas.

```
Fixpoint subst (σ : var → form) (s : form) : form :=
match s with
| Var x ⇒ σ x
```

```
| Imp s t ⇒ Imp (subst σ s) (subst σ t)
| Fal ⇒ Fal
end.
```

We will write σs for *subst* σ s.

We say an entailment relation **respects substitution** if $\sigma A \vdash \sigma s$ whenever $A \vdash s$.

Exercise 10.4.1 Prove that if two substitutions σ_1 and σ_2 agree on the variables which occur in s, then $\sigma_1 s = \sigma_2 s$.

Exercise 10.4.2 Consider the following function *emb* from *form* to *Prop*.

```
Fixpoint emb (s : form) : Prop :=
  match s with
    | Var x ⇒ ⊥
    | Imp s1 s2 ⇒ emb s1 → emb s2
    | Fal ⇒ ⊥
  end.
```

Prove the entailment relation $\lambda As.(\forall t.t \in A \rightarrow emb\ t) \rightarrow emb\ t$ has all the properties except that it does not respect substitution.

```
Goal
  let E : list form → form → Prop := fun A s ⇒ (∀ t, t ∈ A → emb t) → emb s in
  Reflexivity E ∧ Monotonicity E ∧ Cut E ∧ Consistency E
    ∧ CharImp E ∧ CharFal E
    ∧ ¬ Substitution E.
```

Exercise 10.4.3 Prove that if two entailment relations are extensionally the same and one satisfies all the properties, then so does the other.

```
Lemma EntailRelAllProps_ext E E' :
  EntailRelAllProps E → (∀ A s, E A s ↔ E' A s) → EntailRelAllProps E'.
```

10.5 Natural Deduction System

In this section we consider our first proof system for propositional formulas. A proof system defines when a formula s is provable from a context A. We write $A \vdash s$ to mean s is provable from A in the particular proof system under discussion. Note that $\vdash$ is an entailment relation on A and s and we will prove it has all the properties discussed in the previous sections. In fact, it will be the least relation satisfying those properties.

Deduction rules for logical connectives were given in Figure 2.1. The introduction and elimination rules for $\rightarrow$ together with the elimination rule for $\bot$

$$\text{A}\ \frac{}{A \vdash s}\ s \in A \qquad \text{II}\ \frac{A, s \vdash t}{A \vdash s \to t} \qquad \text{IE}\ \frac{A \vdash s \to t \quad A \vdash s}{A \vdash t} \qquad \text{E}\ \frac{A \vdash \bot}{A \vdash s}$$

Figure 10.1: Natural Deduction Rules

essentially give a proof system for propositional formulas. We use contexts (lists of formulas) to represent the collection of assumptions. Note that the introduction rule for $\to$ changes the assumptions and this corresponds to changing the context. We can check if a formula is an assumption in the context using an **assumption rule** to check if the formula is an element of the list. These rules are given in Figure 10.1.

The rules in Figure 10.1 define when $A \vdash s$ in the system $\mathcal{N}$. That is, the rules define when s is provable from A in $\mathcal{N}$. One can use the rules in Figure 10.1 to justify $A \vdash s$.

Consider the following example.

Example 10.5.1 Let A be a context and s and t be formulas. We can use the rules of $\mathcal{N}$ to derive $A \vdash s \to \neg s \to t$ as follows:

$$\text{II}\ \cfrac{\text{II}\ \cfrac{\text{E}\ \cfrac{\text{IE}\ \cfrac{\text{A}\ \cfrac{}{A, s, \neg s \vdash s \to \bot} \qquad \text{A}\ \cfrac{}{A, s, \neg s \vdash s}}{A, s, \neg s \vdash \bot}}{A, s, \neg s \vdash t}}{A, s \vdash \neg s \to t}}{A \vdash s \to \neg s \to t}$$

We can represent the natural deduction system $\mathcal{N}$ in Coq as an inductive predicate *nd*. The proposition *nd* A s is provable precisely when $A \vdash s$. Note that A is a nonuniform parametric argument of *nd* and s is a nonparametric argument of *nd*.

```
Inductive nd : context → form → Prop :=
| ndA A s :   s ∈ A → nd A s
| ndII A s t : nd (s::A) t → nd A (Imp s t)
| ndIE A s t : nd A (Imp s t) → nd A s → nd A t
| ndE A s :   nd A Fal → nd A s.
```

We can now reconsider Example 10.5.1 as a proof in Coq. Compare the Coq proof script with the diagram in Example 10.5.1.

```
Goal ∀ A s t, nd A (Imp s (Imp (Not s) t)).
```

$$\text{app}\ \frac{A \vdash s}{A \vdash u}\ s \to u \in A \qquad \text{weak}\ \frac{A \vdash s}{A' \vdash s}\ A \subseteq A' \qquad \text{W}\ \frac{A \vdash s}{A, t \vdash s}$$

$$\text{IEweak}\ \frac{B \vdash s \to t \qquad B \subseteq A \qquad A \vdash s}{A \vdash t} \qquad \text{DN}\ \frac{A \vdash s}{A \vdash \neg\neg s}$$

Figure 10.2: Some Admissible Rules

```
Proof.
  intros A s t. apply ndII, ndII. apply ndE. apply ndIE with (s := s).
  - apply ndA. left. reflexivity.
  - apply ndA. right. left. reflexivity.
Qed.
```

From now on we will tend to work at the mathematical level and leave the reader to examine the available Coq version.

A rule is **admissible** in a proof system if adding the rule to the proof system does not change what is provable in the system. This is equivalent to saying that the conclusion of the rule is provable whenever the premises are provable. We prove all the rules in Figure 10.5 are admissible in $\mathcal{N}$.

We begin this process by proving admissibility of app. This rule allows us to simulate Coq's *apply* tactic. If an implication $s \to t$ is in the context A and we want to prove $A \vdash t$, then it is enough to prove $A \vdash s$. Admissibility of app is the content of the following lemma whose proof is simply a combination of the IE and A rules.

$$\frac{\overline{A \vdash s \to u} \qquad A \vdash s}{A \vdash u}$$

We will often do proofs by induction over $A \vdash s$, i.e., over the inductive predicate *nd*. In the process of doing such an inductive proof we must consider each of the four rules in Figure 10.1. Each rule has the form

$$\frac{A_1 \vdash s_1 \cdots A_n \vdash s_n}{A \vdash s}$$

for $n \in \{0, 1, 2\}$. For such a rule we assume the desired property for A_i and s_i as inductive hypotheses and prove the desired property for A and s. In Coq the induction principle of *nd* corresponds to the type of *nd_ind*:

```
Check (nd_ind :
  ∀ p : context → form → Prop,
  (∀ (A : context) (s : form), s ∈ A → p A s) →
  (∀ (A : context) (s t : form), nd (s :: A) t → p (s :: A) t → p A (Imp s t)) →
  (∀ (A : context) (s t : form), nd A (Imp s t) → p A (Imp s t) → nd A s → p A s → p A t) →
  (∀ (A : context) (s : form), nd A Fal → p A Fal → p A s) →
  ∀ (A : context) (s : form), nd A s → p A s).
```

Note that the induction principle makes precise that we must prove a case for each rule in Figure 10.1 and what inductive hypotheses we obtain in each case. We can also visualize these four proof obligations in the form of rules.

$$\text{A}\ \frac{}{p\ A\ s}\ s \in A \qquad \text{II}\ \frac{A, s \vdash t \quad (IH)\ p\ (A, s)\ t}{p\ A\ (s \to t)} \qquad \text{IE}\ \frac{A \vdash s \to t \quad (IH)\ p\ A\ (s \to t) \qquad A \vdash s \quad (IH)\ p\ A\ s}{p\ A\ t}$$

$$\text{E}\ \frac{A \vdash \bot \quad (IH)\ p\ A\ \bot}{p\ A\ s}$$

In each case we must prove the conclusion using the premises (including the inductive hypotheses) and the side conditions as assumptions.

Our first such inductive proof will be monotonicity of $\vdash$: if $A \subseteq A'$ and $A \vdash s$, then $A' \vdash s$. In other words, we prove the rule weak in Figure 10.5 is admissible. The rule weak is often called the **weakening rule**.

Here the desired property of A and s we wish to prove by induction is $\forall A', A \subseteq A' \to A' \vdash s$. We prove by induction that A and s have this desired property whenever $A \vdash s$.

```
Lemma nd_weak A A' s :
  A ⊆ A' → nd A s → nd A' s.
```

Proof We argue by induction on the proof of $A \vdash s$ and consider each rule carefully. Future mathematical proofs will not be given at this level of detail.

Consider the assumption rule A:

$$\text{A}\ \frac{}{A \vdash s}\ s \in A$$

Note that in this case $s \in A$. We must prove $\forall A', A \subseteq A' \to A' \vdash s$. Assume $A \subseteq A'$. Hence $s \in A'$ and so $A' \vdash s$ by the assumption rule.

Consider the introduction rule II for implication:

$$\text{II}\ \frac{A, s \vdash t}{A \vdash s \to t}$$

We must prove $\forall A', A \subseteq A' \to A' \vdash s \to t$. The inductive hypothesis is $\forall A', A, s \subseteq A' \to A' \vdash t$. Assume $A \subseteq A'$. Clearly $A, s \subseteq A', s$. By the inductive hypothesis we know $A', s \vdash t$. Hence $A' \vdash s \to t$ by II.

Consider the elimination rule IE for implication:

$$\text{IE}\ \frac{A \vdash s \to t \qquad A \vdash s}{A \vdash t}$$

We must prove $\forall A', A \subseteq A' \to A' \vdash t$. Since there are two premises, there are two inductive hypotheses. The first inductive hypothesis is $\forall A', A \subseteq A' \to A' \vdash s \to t$. The second inductive hypothesis is $\forall A', A \subseteq A' \to A' \vdash s$. Assume $A \subseteq A'$. By the inductive hypotheses we know $A' \vdash s \to t$ and $A' \vdash s$. Hence $A' \vdash t$ by IE.

Consider the elimination rule E for $\bot$:

$$\text{E}\ \frac{A \vdash \bot}{A \vdash s}$$

We must prove $\forall A', A \subseteq A' \to A' \vdash s$. The inductive hypothesis is $\forall A', A \subseteq A' \to A' \vdash \bot$. Assume $A \subseteq A'$. By the inductive hypotheses we know $A' \vdash \bot$ and so $A' \vdash s$ by E. ∎

As an obvious corollary, we know that $A, t \vdash s$ whenever $A \vdash s$. That is, the following rule is admissible:

$$\text{W}\ \frac{A \vdash s}{A, t \vdash s}$$

We could also obtain corollaries which combine weakening with the defining rules of the calculus. We will only do so for IE: If $B \vdash s \to t$, $B \subseteq A$ and $A \vdash s$, then $A \vdash t$. That is, we have admissibility of the following rule:

$$\text{IEweak}\ \frac{B \vdash s \to t \qquad B \subseteq A \qquad A \vdash s}{A \vdash t}$$

Finally we can use II, app and W to prove that if $A \vdash s$, then $A \vdash \neg\neg s$. That is, the following rule is admissible. We leave the details to the reader.

$$\text{DN}\ \frac{A \vdash s}{A \vdash \neg\neg s}$$

It turns out that $A \vdash s$ is decidable, but we do not yet have the tools to prove this. The decidability proof will come later.

Exercise 10.5.2 Prove the following goals.

```
Goal ∀ A s, nd A (Imp s s).
Goal ∀ A s, nd A (Imp Fal s).
Goal ∀ A s t, nd A (Imp s (Imp t s)).
Goal ∀ A s t, nd A (Imp (Imp s t) (Imp (Not t) (Not s))).
```

Exercise 10.5.3 Prove that $\vdash$ respects substitution.

```
Lemma nd_subst A s σ : nd A s → nd (map (subst σ) A) (subst σ s).
```

Exercise 10.5.4 Prove the following soundness result for $\mathcal{N}$ relative to boolean semantic consequence: If $A \vdash s$, then $A \vDash s$. Use the result to conclude consistency of $\vdash$.

Exercise 10.5.5 Prove $\vdash$ has all the properties of entailment relations defined earlier.

```
Lemma nd_EntailRelAllProps : EntailRelAllProps nd.
```

Exercise 10.5.6 Prove *nd* is the least reflexive entailment relation satisfying cut and the characteristic properties of $\rightarrow$ and $\perp$.

```
Lemma nd_least_EntailRelAllProps (E : context → form → Prop) :
  Reflexivity E → Cut E → CharImp E → CharFal E → ∀ A s, nd A s → E A s.
```

Exercise 10.5.7 Extend the formulas and natural deduction system to include conjunction and disjunction.

Exercise 10.5.8 Prove the following two lemmas.

```
Lemma ndassert (A : context) (s u : form) :
 nd A s → nd (s::A) u → nd A u.
Lemma ndappbin (A : context) (s t u : form) :
 Imp s (Imp t u) ∈ A → nd A s → nd A t → nd A u.
```

10.6 Classical Natural Deduction

We now consider classical propositional logic. Classical propositional logic can prove formulas such as instances of double negation $\neg\neg s \rightarrow s$ and instances of Peirce's law $((s \rightarrow t) \rightarrow s) \rightarrow s$. The propositional formulas provable in classical propositional logic correspond to those which evaluate to true under every boolean assignment, if one interprets $\perp$ as *false* and interprets implication by truth tables (i.e., *implb* in the Coq library).

$$\text{A}\ \frac{}{A \vdash s}\ s \in A \qquad \text{II}\ \frac{A, s \vdash t}{A \vdash s \to t} \qquad \text{IE}\ \frac{A \vdash s \to t \qquad A \vdash s}{A \vdash t} \qquad \text{C}\ \frac{A, \neg s \vdash \bot}{A \vdash s}$$

Figure 10.3: Classical Natural Deduction Rules

The classical natural deduction system $\mathcal{N}_C$ is defined by the rules in Figure 10.3. Note that the difference from the previous system is that the elimination rule E for $\bot$ has been replaced by the **contradiction rule** C. In Coq, the classical natural deduction system can be defined as an inductive predicate *ndc* as usual.

All the rules from Figure 10.5 are admissible in the classical system $\mathcal{N}_C$. In each case admissibility can be proven using the same strategy as admissibility of the rule in $\mathcal{N}$. We leave the details to the reader.

Since we have omitted the elimination rule for $\bot$, a natural question is whether we can infer $A \vdash s$ from $A \vdash \bot$. That is, one may ask if the E rule (a defining rule for $\mathcal{N}$) is admissible in the system $\mathcal{N}_C$. We can prove admissibility of E in $\mathcal{N}_C$ easily using the contradiction rule and weakening.

$$\frac{\dfrac{A \vdash \bot}{A, \neg s \vdash \bot}}{A \vdash s}$$

Now we have enough information to know $A \vdash s$ in $\mathcal{N}$ implies $A \vdash s$ in $\mathcal{N}_C$. The proof is by a simple induction on the proof of $A \vdash s$ in $\mathcal{N}$.

Theorem 10.6.1 If $A \vdash s$ in $\mathcal{N}$, then $A \vdash s$ in $\mathcal{N}_C$.

Proof We argue by induction on $A \vdash s$. In every case except the explosion rule, we can directly use the corresponding rule in $\mathcal{N}_C$. In the case of the explosion rule, we can use the fact that the explosion rule is admissible in $\mathcal{N}_C$. ∎

Finally we prove $A \vdash s$ if and only if $A, \neg s \vdash \bot$. That is, in $\mathcal{N}_C$ it is enough to consider refutability of contexts.

Theorem 10.6.2 $A \vdash s$ in $\mathcal{N}_C$ if and only if $A, \neg s \vdash \bot$ in $\mathcal{N}_C$.

Proof Suppose $A \vdash s$. By weakening know $A, \neg s \vdash s$. By A we know $A, \neg s \vdash \neg s$. By IE we conclude $A, \neg s \vdash \bot$ as desired.

For the other direction, suppose $A, \neg s \vdash \bot$. We can conclude $A \vdash s$ simply using the contradiction rule (C). ∎

Exercise 10.6.3 Prove $A \vdash \neg\neg s \to s$.

```
Goal ∀ A s, ndc A (Imp (Not (Not s)) s).
```

Exercise 10.6.4 Prove the following lemmas for $\mathcal{N}_C$.

```
Lemma ndcA2 A s t :
 ndc (t :: s :: A) s.

Lemma ndcapp A s u :
  Imp s u ∈ A → ndc A s → ndc A u.

Lemma ndcapp1 A s u :
  ndc (Imp s u :: A) s → ndc (Imp s u :: A) u.

Lemma ndcapp2 A s t u :
  ndc (t :: Imp s u :: A) s → ndc (t :: Imp s u :: A) u.

Lemma ndcapp3 A s t u v :
  ndc (t :: v :: Imp s u :: A) s → ndc (t :: v :: Imp s u :: A) u.
```

Use the lemmas above to prove $A \vdash ((s \to t) \to s) \to s$. That is, prove Peirce's Law.

```
Goal ∀ A s t, ndc A (Imp (Imp (Imp s t) s) s).
```

Exercise 10.6.5 Prove $\vdash$ is closed under substitution.

```
Lemma ndc_subst A s σ : ndc A s → ndc (map (subst σ) A) (subst σ s).
```

Exercise 10.6.6 Prove the following result.

```
Lemma ndc_eval_xm_sound A s (e:form → Prop) :
 XM →
 ¬e Fal → (∀ t u, e (Imp t u) ↔ e t → e u) →
 ndc A s → (∀ t, t ∈ A → e t) → e s.
```

10.7 Glivenko's Theorem

Glivenko's Theorem states that a propositional formula s is classically provable if and only if its double negation is intuitionistically provable. The most interesting half of this equivalence is that $\neg\neg s$ is intuitionistically provable if s is classically provable. In particular, if $A \vdash s$, then $A \vdash \neg\neg s$. We prove this implication by induction on the proof of $A \vdash s$. We leave the converse implication as an exercise.

Theorem 10.7.1 (Glivenko) If $A \vdash s$ in $\mathcal{N}_C$, then $A \vdash \neg\neg s$ in $\mathcal{N}$.

Proof We prove this by induction on the proof of $A \vdash s$. We will use the admissible rules DN, app and IEweak from Figure 10.5 as well as the usual rules defining $\mathcal{N}$.

For the assumption rule we assume we have $s \in A$ and need to prove $A \vdash \neg\neg s$ in $\mathcal{N}$. We easily have this by the admissible rule DN and the assumption rule.

For implication introduction we know $A, s \vdash \neg\neg t$ in $\mathcal{N}$ by the inductive hypothesis. From this we can derive $A \vdash \neg\neg(s \to t)$ in $\mathcal{N}$ using the admissible rules app and IEweak (along with the usual $\mathcal{N}$ rules) as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\overset{IH}{A, s \vdash \neg\neg t} \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{A, \neg(s \to t), s, t, s \vdash t}}{A, \neg(s \to t), s, t \vdash s \to t}}{A, \neg(s \to t), s, t \vdash \bot}}{A, \neg(s \to t), s \vdash \neg t}}{A, \neg(s \to t), s \vdash \bot}}{A, \neg(s \to t), s \vdash t}}{A, \neg(s \to t) \vdash s \to t}}{A, \neg(s \to t) \vdash \bot}}{A \vdash \neg\neg(s \to t)}$$

For IE we know $A \vdash \neg\neg(s \to t)$ and $A \vdash \neg\neg s$ in $\mathcal{N}$ by the inductive hypotheses. From this we can derive $A \vdash \neg\neg t$ using the admissible rules app and IEweak (along with the usual $\mathcal{N}$ rules) as follows:

$$\dfrac{\dfrac{\overset{IH}{A \vdash \neg\neg s} \qquad \dfrac{\dfrac{\overset{IH}{A \vdash \neg\neg(s \to t)} \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{A, \neg t, s, s \to t \vdash s}}{A, \neg t, s, s \to t \vdash t}}{A, \neg t, s, s \to t \vdash \bot}}{A, \neg t, s \vdash \neg(s \to t)}}{A, \neg t, s \vdash \bot}}{A, \neg t \vdash \neg s}}{A, \neg t \vdash \bot}}{A \vdash \neg\neg t}$$

For the contradiction rule we know $A, \neg s \vdash \neg\neg\bot$ in $\mathcal{N}$ by the inductive hy-

pothesis. From this we can derive $A \vdash \neg\neg s$ in $\mathcal{N}$ as follows:

$$\dfrac{\overset{IH}{A, \neg s \vdash \neg\neg\bot} \qquad \dfrac{\dfrac{}{A, \neg s, \bot \vdash \bot}}{A, \neg s \vdash \neg\bot}}{\dfrac{A, \neg s \vdash \bot}{A \vdash \neg\neg s}}$$

As a consequence of Glivenko's theorem, we can prove that refutability in $\mathcal{N}$ is equivalent to refutability in $\mathcal{N}_C$.

Corollary 10.7.2 $A \vdash \bot$ in $\mathcal{N}$ if and only if $A \vdash \bot$ in $\mathcal{N}_C$.

Proof We know $A \vdash \bot$ in $\mathcal{N}$ implies $A \vdash \bot$ in $\mathcal{N}_C$ by Theorem 10.6.1. For the other direction, suppose $A \vdash \bot$ in $\mathcal{N}_C$. By Glivenko (Theorem 10.7.1) we know $A \vdash \neg\neg\bot$. However, from $A \vdash \neg\neg\bot$ it is easy to derive $A \vdash \bot$ as follows:

$$\dfrac{A \vdash \neg\neg\bot \qquad \dfrac{\dfrac{}{A, \bot \vdash \bot}}{A \vdash \neg\bot}}{A \vdash \bot}$$

A further consequence is that $A \vdash s$ in $\mathcal{N}_C$ if and only if $A, \neg s \vdash \bot$ in $\mathcal{N}$.

Corollary 10.7.3 $A \vdash s$ in $\mathcal{N}_C$ if and only if $A, \neg s \vdash \bot$ in $\mathcal{N}$.

Proof By Theorem 10.6.2 we know $A \vdash s$ in $\mathcal{N}_C$ if and only if $A, \neg s \vdash \bot$ in $\mathcal{N}_C$. By Corollary 10.7.2 we know $A, \neg s \vdash \bot$ in $\mathcal{N}_C$ if and only if $A, \neg s \vdash \bot$ in $\mathcal{N}$. Hence we have the desired equivalence. ∎

Exercise 10.7.4 Prove the easy half of Glivenko's theorem.

```
Lemma Glivenko_converse A s :
  nd A (Not (Not s)) → ndc A s.
```

Exercise 10.7.5 Prove the following consequence of Glivenko's theorem.

```
Goal ∀ A, ¬ ∃ s, ndc A (Not s) ∧ ¬ nd A (Not s).
```

Exercise 10.7.6 Prove consistency of $\vdash$.

```
Lemma ndc_con : ¬ ndc nil Fal.
```

Exercise 10.7.7 Prove *ndc* has all the properties of entailment relations defined earlier.

```
Lemma ndc_EntailRelAllProps : EntailRelAllProps ndc.
```

$$\text{A}\ \frac{}{s}\ s \in A \qquad \text{K}\ \frac{}{s \to t \to s} \qquad \text{S}\ \frac{}{(s \to t \to u) \to (s \to t) \to s \to u} \qquad \text{E}\ \frac{}{\bot \to u}$$

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

Figure 10.4: Hilbert Rules for Intuitionistic Propositional Logic

10.8 Hilbert System

The natural deduction systems require assumption management. In particular the implication introduction rule II changes the assumptions. It turns out that we can omit the implication introduction rule if we replace it with a number of **initial rules** – i.e., rules with no premises. One initial rule states that every formula of the form $s \to t \to s$ is provable. We call such a formula a K-formula. Another initial rule states that every formula $(s \to t \to u) \to (s \to t) \to s \to u$ is provable. Such formulas are called S-formulas. Doing this would yield a system in which only two rules have premises: a rule like IE and a rule like E. Indeed we can define a system in which the only rule with premises is a rule known as **modus ponens** which has the same form as IE since we can replace the E rule with an initial rule stating that every explosion formula (i.e., formula of the form $\bot \to s$) is provable. Such systems are called **Hilbert systems**. The rules in Figure 10.4 define our **Hilbert system for intuitionistic propositional logic**, which refer to by the name $\mathcal{H}$. In particular, we have $A \vdash s$ in system $\mathcal{H}$ when s is derivable from context A using the rules in Figure 10.4.

We can define this in Coq as an inductive predicate *hil* in the usual way.

We can easily prove by induction that if $A \vdash s$ in $\mathcal{H}$, then $A \vdash s$ in $\mathcal{N}$.

Lemma 10.8.1 If $A \vdash s$ in $\mathcal{H}$, then $A \vdash s$ in $\mathcal{N}$.

Proof We argue by induction on the proof of $A \vdash s$ in $\mathcal{H}$. We must argue a case for each rule in Figure 10.4. If $s \in A$, then we know $A \vdash s$ in $\mathcal{N}$ by A. The next three cases involve proving K-formulas, S-formulas and formulas of the form $\bot \to s$ in $\mathcal{N}$. Each of these cases is easy. Finally, we consider the modus ponens case. Assume $A \vdash s \to t$ and $A \vdash s$ in $\mathcal{H}$. The inductive hypotheses yield $A \vdash s \to t$ and $A \vdash s$ in $\mathcal{N}$. We conclude $A \vdash t$ in $\mathcal{N}$ using IE. ■

Note that in each case of the inductive proof, we have proven one of the defining rules of $\mathcal{H}$ is admissible in $\mathcal{N}$.

The converse also holds: If $A \vdash s$ in $\mathcal{N}$, then $A \vdash s$ in $\mathcal{H}$. In order to prove this, we first prove an important result called **the deduction theorem**. The deduction theorem states that if $A, s \vdash t$, then $A \vdash s \to t$. In other words, the II rule (a defining rule of the system $\mathcal{N}$) is admissible in $\mathcal{H}$. The proof is by induction on the proof of $A, s \vdash t$ using the results above.

Theorem 10.8.2 (Deduction Theorem) If $A, s \vdash t$, then $A \vdash s \to t$.

Proof We prove this by induction on the proof of $A, s \vdash t$. There are three possible cases to consider.

- Suppose $t \in A$, t is a K-formula, t is an S-formula or t is an explosion formula. In any of these cases $A \vdash t$ and $A \vdash t \to s \to t$. Hence $A \vdash s \to t$.
- Suppose t is s. We need to prove $A \vdash s \to s$. This follows from the fact that $(s \to (s \to s) \to s) \to (s \to s \to s) \to s \to s$ is an S-formula while $s \to (s \to s) \to s$ and $s \to s \to s$ are K-formulas.
- Suppose $A, s \vdash u \to t$ and $A, s \vdash u$. By the inductive hypothesis $A \vdash s \to u \to t$ and $A \vdash s \to u$. In order to see that $A \vdash s \to t$ it suffices to note that $(s \to u \to t) \to (s \to u) \to s \to t$ is an S-formula. ■

We can now prove $A \vdash s$ in $\mathcal{N}$ implies $A \vdash s$ in $\mathcal{H}$. The proof is by induction on the proof of $A \vdash s$ in $\mathcal{N}$. The deduction theorem is used for the II-case. The remaining cases are straightforward.

Lemma 10.8.3 If $A \vdash t$ in $\mathcal{N}$, then $A \vdash t$ in $\mathcal{H}$.

Proof We must argue a case for each rule in Figure 10.1. For the A case we must prove $A, s \vdash s$ in $\mathcal{H}$. We know this by A since $s \in A, s$. For the IE case the inductive hypotheses give $A \vdash s \to t$ and $A \vdash s$ in $\mathcal{H}$. We conclude $A \vdash t$ in $\mathcal{H}$ using MP. For the E case the inductive hypothesis gives $A \vdash \bot$ in $\mathcal{H}$. We conclude $A \vdash s$ in $\mathcal{H}$ using the following derivation.

$$\text{MP}\ \dfrac{\text{E}\ \dfrac{}{A \vdash \bot \to s} \qquad A \vdash \bot}{A \vdash s}$$

For the II case we use the deduction theorem (Theorem 10.8.2). ■

Combining the results we know $A \vdash s$ in $\mathcal{H}$ if and only if $A \vdash s$ in $\mathcal{N}$.

Theorem hil_iff_nd A s :
  hil A s ↔ nd A s.

Exercise 10.8.4 Prove the following form of weakening for the Hilbert calculus.

$$\text{A}\ \frac{}{A \vdash s}\ s \in A \qquad \text{II}\ \frac{A, s \vdash t}{A \vdash s \to t} \qquad \text{IE}\ \frac{A \vdash s \to t \qquad A \vdash s}{A \vdash t} \qquad \text{E}\ \frac{A \vdash \bot}{A \vdash s}$$

$$\text{WXM}\ \frac{A, \neg s \vdash t \qquad A, \neg\neg s \vdash t}{A \vdash t}$$

Figure 10.5: ND Rules for a logic with weak excluded middle

```
Lemma hilW A s t :
  hil A t → hil (s::A) t.
```

Exercise 10.8.5 Prove the following.

```
Lemma hilassert A s u :
  hil A s → hil (s::A) u → hil A u.
```

Exercise 10.8.6 Prove consistency of *hil* using the equivalence with *nd*.

```
Lemma hil_con : ¬ hil nil Fal.
```

Exercise 10.8.7 Use *nd_hil*, *hil_nd* and Exercise 10.4.3 to prove *hil* has all the properties of entailment relations defined earlier.

```
Lemma hil_EntailRelAllProps : EntailRelAllProps hil.
```

Exercise 10.8.8 Give a Hilbert calculus for classical propositional logic and define a corresponding inductive predicate *hilc* in Coq. Prove the deduction theorem for *hilc* and use the deduction theorem to prove the equivalence between *hilc* and *ndc*.

10.9 Intermediate Logics

An intermediate propositional logic is one that proves more than intuitionistic propositional logic but less than classical propositional logic. It is not obvious that such logics exist, but in fact they do. We will consider two examples.

Let $\vdash$ be the entailment relation defined by the rules in Figure 10.5. The formula $(\neg x \to y) \to (\neg\neg x \to y) \to y$ is not intuitionistically provable. On the other hand, the rules in Figure 10.5 are enough to prove $\vdash (\neg x \to y) \to (\neg\neg x \to y) \to y$. Furthermore, $\vdash$ is not full classical logic since we cannot prove double negation in general: $\nvdash \neg\neg x \to x$.

$$\text{A}\ \frac{}{A \vdash s}\ s \in A \qquad \text{II}\ \frac{A, s \vdash t}{A \vdash s \to t} \qquad \text{IE}\ \frac{A \vdash s \to t \qquad A \vdash s}{A \vdash t} \qquad \text{E}\ \frac{A \vdash \bot}{A \vdash s}$$

$$\text{GD}\ \frac{A, s \to t \vdash u \qquad A, t \to s \vdash u}{A \vdash u}$$

Figure 10.6: ND Rules for Gödel-Dummett Logic

Let $\vdash$ be the entailment relation defined by the rules in Figure 10.6. The formula $\vdash ((x \to y) \to z) \to ((y \to x) \to z) \to z$ is not intuitionistically provable. The rules in Figure 10.6 above are enough to prove $\vdash ((x \to y) \to z) \to ((y \to x) \to z) \to z$. Again, we cannot prove double negation in general: $\nvdash \neg\neg x \to x$. Hence $\vdash$ is again an entailment relation strictly between intuitionistic and classical logic.

10.10 Remarks

The first deduction systems developed by Frege in 1879 [3] were in the Hilbert style. (Hilbert studied and popularized such systems later.) Natural deduction systems were created independently by Gentzen [4] and Jaśkowski in the 1930s. The Glivenko result was published in 1929 [8] (before the invention of natural deduction).

11 Classical Tableau Method

In this chapter we show that a propositional formula is classically provable if and only if it is satisfied by all boolean assignments. We obtain this result with a method known as classical tableau method. Given a formula, the method decides whether the formula is satisfiable. If the formula is satisfiable, the method yields a satisfying assignment. If the formula is unsatisfiable, the method yields a proof of the negation of the formula in a classical calculus.

11.1 Boolean Evaluation and Satisfiability

A **boolean assignment** is a function that maps every variable to a boolean value. Given a boolean assignment, we can evaluate every propositional formula to a boolean value. We will use α and β as names for boolean assignments:

$$\alpha, \beta \;:\; assn \;:=\; var \to \mathbb{B}$$

We formalize the evaluation of propositional formulas with a function $eval$: $assn \to form \to \mathbb{B}$ defined by recursion on formulas:

$$\begin{aligned} eval\ \alpha\ x &:= \alpha x \\ eval\ \alpha\ (Imp\ s\ t) &:= \text{if } eval\ \alpha\ s \text{ then } eval\ \alpha\ t \text{ else } true \\ eval\ \alpha\ Fal &:= false \end{aligned}$$

Note that the evaluation of an implication is defined with a boolean conditional such that an implication $s \to t$ evaluates to *true* if and only if either both constituents s and t evaluate to *true* or s evaluates to *false*.

We say that an assignment **satisfies** a formula if the formula evaluates to *true* with the assignment. We say that an assignment **dissatisfies** a formula if it does not satisfy the formula. An assignment dissatisfies a formula if and only if the formula evaluates to *false* with the assignment. Moreover, an assignment dissatisfies a formula if and only if it satisfies the negation of the formula.

A propositional formula is **satisfiable** if there is a boolean assignment under which it evaluates to *true*. A formula is **unsatisfiable** if it is not satisfiable.

Formally, we will work with an alternative characterization of boolean satisfaction, which employs a function $\vDash$ mapping an assignment α and a formula s

to a proposition equivalent to $eval\ \alpha\ s = true$.

$$\begin{aligned} \alpha \vDash x &:= \text{if } \alpha x \text{ then } \top \text{ else } \bot \\ \alpha \vDash Imp\ s\ t &:= \alpha \vDash s \to \alpha \vDash t \\ \alpha \vDash Fal &:= \bot \end{aligned}$$

The symbol $\vDash$ is pronounced **double turnstile**. By induction on s it follows that the function $\vDash$ captures boolean satisfaction:

Fact 11.1.1 $\alpha \vDash s \ \leftrightarrow\ eval\ \alpha\ s = true$.

From the equivalence it follows that $\alpha \vDash s$ is decidable.

Fact 11.1.2 $\alpha \vDash s$ is decidable.

Here is the formal definition of satisfiability we will work with:

$$sat\ s := \exists \alpha.\ \alpha \vDash s$$

Exercise 11.1.3 Prove Facts 11.1.1 and 11.1.2.

Exercise 11.1.4 Prove that $\alpha \vDash \neg s$ and $\alpha \nvDash s$ are definitionally equal.

Exercise 11.1.5 Prove the following:

a) $\alpha \vDash s \to t \ \leftrightarrow\ \alpha \nvDash s \lor \alpha \vDash t$

b) $\alpha \vDash \neg(s \to t) \ \leftrightarrow\ \alpha \vDash s \land \alpha \nvDash t$

c) $\alpha \vDash \neg s \ \leftrightarrow\ \alpha \nvDash s$

d) $\alpha \vDash x \ \leftrightarrow\ \alpha x = true$

e) $\alpha \vDash \neg x \ \leftrightarrow\ \alpha x = false$

Exercise 11.1.6 The function $\vDash$ maps implications of formulas to implications of propositions. This yields the right meaning since $\alpha \vDash s$ is decidable. You can gain more insight into this phenomenon by proving the following equivalences for a decidable proposition X and an arbitrary proposition Y.

a) $X \to Y \ \leftrightarrow\ \text{if } \ulcorner X \urcorner \text{ then } Y \text{ else } \top$

b) $X \land Y \ \leftrightarrow\ \text{if } \ulcorner X \urcorner \text{ then } Y \text{ else } \bot$

c) $X \lor Y \ \leftrightarrow\ \text{if } \ulcorner X \urcorner \text{ then } \top \text{ else } Y$

Exercise 11.1.7 Extend the development to formulas with native conjunctions and disjunctions. Write a function that translates formulas to formulas without conjunctions and disjunctions such that a formula and its translation are satisfied by the same assignments.

11.2 Validity and Boolean Entailment

A formula is **valid** if it is satisfied by every assignment.

Fact 11.2.1 A formula is valid if and only if its negation is unsatisfiable.

The fact is negation happy. If we spell it out, we obtain

$$(\forall \alpha.\ \alpha \vDash s) \ \leftrightarrow\ \neg\exists\alpha.\ \alpha \vDash s$$

Now remember that $\alpha \vDash \neg s$ is definitionally equal to $\neg(\alpha \vDash s)$. With de Morgan the right hand side of the equivalence becomes $\forall \alpha.\ \neg\neg(\alpha \vDash s)$. Since $\alpha \vDash s$ is decidable, we can delete the double negation.

Fact 11.2.2 Every classically provable formula is valid.

Proof By induction on the formula using the classical Hilbert system. ∎

We will eventually show that every valid formula is classically provable.

Using boolean assignments, we can define an entailment predicate we call boolean entailment. It will turn out that boolean entailment agrees with classical entailment. An assignment **satisfies** a list of formulas if it satisfies every formula of the list:

$$\alpha \vDash A \ :=\ \forall s.\ s \in A \to \alpha \vDash s$$

We define **boolean entailment** as follows:

$$A \vDash s \ :=\ \forall \alpha.\ \alpha \vDash A \to \alpha \vDash s$$

Note that s is valid if and only if $nil \vDash s$.

Fact 11.2.3 If $A \vdash s$ in the classical ND calculus, then $A \vDash s$.

Proof By induction on the derivation $A \vdash s$. ∎

Exercise 11.2.4 Prove Facts 11.2.1, 11.2.2, and 11.2.3.

11.3 Signed Formulas and Clauses

The tableau method works with signed formulas. A **signed formula** is a pair of a sign and a formula, where a **sign** is either positive or negative.

```
Inductive sform : Type :=
| Pos : form → sform
| Neg : form → sform.
```

For a positively signed formula we write s^+ or simply s, and for a negatively signed formula we write s^-. We will speak of **positive** and **negative** formulas.

A **clause** is a list of signed formulas.

Definition clause := list sform.

An assignment **satisfies** a clause if it satisfies every positive formula of the clause and dissatisfies every negative formula of the clause. A clause is **satisfiable** if it is satisfied by at least one assignment.

By our definitions an assignment dissatisfies a formula if and only if it satisfies the negation of the formula. Thus a negative formula s^- is semantically equivalent to the negated formula $\neg s$. One may see a negative sign as an external negation. Similarly, one may see a clause as an external conjunction.

We use C, D, and E as names for clauses and S and T as names for signed formulas. Formally, we define satisfaction of clauses with a recursive function $\models$:

$$\begin{aligned} \alpha \models nil &:= \top \\ \alpha \models s^+ :: C &:= \alpha \models s \wedge \alpha \models C \\ \alpha \models s^- :: C &:= \alpha \not\models s \wedge \alpha \models C \end{aligned}$$

A clause C **entails** a clause D if every assignment satisfying C also satisfies D:

$$C \models D \;:=\; \forall \alpha.\, \alpha \models C \to \alpha \models D$$

Exercise 11.3.1 Prove the following facts about satisfiability and entailment of clauses:

a) $C \subseteq D \to \alpha \models D \to \alpha \models C$

b) $C \subseteq D \to sat\ D \to sat\ C$

c) $s^+ \in C \to s^- \in C \to \neg sat\ C$

d) If $C \subseteq D$, then $D \models C$.

e) If $C \models D$ and $D \models E$, then $D \models E$.

f) If $C \models D$ and C is satisfiable, then D is satisfiable.

11.4 Solved Clauses

A clause is **solved** if it contains only signed variables and no conflicting pair x^+ and x^-. A solved clause can be understood as a partial assignment that fixes the values of finitely many variables. A solved clause is satisfied by every assignment that respects the constraints imposed by the signed variables of the clause. Since the signed variables of a solved clause do not clash, every signed clause is satisfiable.

We define a function φ mapping clauses to assignments:

$$\varphi\, C\, x \;:=\; \text{if } \ulcorner x^+ \in C \urcorner \text{ then } \mathit{true} \text{ else } \mathit{false}$$

Fact 11.4.1 Let C be a solved clause. Then $\varphi\, C \vDash C$.

Exercise 11.4.2 Let C be a solved clause. Prove the following.

a) $\varphi\, C \vDash C$.

b) C is satisfiable.

c) If $x^- \notin C$, then $x^+ :: C$ is solved.

d) If $x^+ \notin C$, then $x^- :: C$ is solved.

11.5 Tableau Method

A clause is **clashed** if it contains either a positive occurrence of $\bot$ or a complementary pair s and s^-. Clearly, every clashed clause is unsatisfiable. A clause is **flat** if it contains no implication. Clearly, a flat clause is satisfiable if and only if it is not clashed.

The tableau method decides satisfiability of clauses by reducing clauses to flat clauses. To reduce a clause to flat clauses, implications appearing in the clause are eliminated one by one using the following equivalences:

$$\alpha \vDash s \to t^+ :: C \;\leftrightarrow\; \alpha \vDash s^- :: C \;\vee\; \alpha \vDash t^+ :: C$$
$$\alpha \vDash s \to t^- :: C \;\leftrightarrow\; \alpha \vDash s^+ :: t^- :: C$$

When we apply the tableau method by hand, we do the bookkeeping with a tree-structured table called a *tableau* (hence the name tableau method). Figure 11.1 shows a complete tableau for a clause consisting of a negative instance of Peirce's law. We start by writing the signed formulas of the initial clause in separate rows of the tableau. Then implications appearing in the tableau are eliminated by applying the above equivalences from left to right. When we eliminate an implication, we mark it with a number and extend the tableau with the additional signed formulas specified by the equivalence.[1] For negative implications we add two signed formulas:

$$\frac{s \to t^-}{\begin{array}{c} s \\ t^- \end{array}}$$

[1] Since eliminated implications stay on the tableau, it is maybe more appropriate to say that implications are decomposed.

$$\begin{array}{ll|ll}
\multicolumn{2}{c}{((x \to y) \to x) \to x^- \quad 1} \\
\multicolumn{2}{c}{(x \to y) \to x \qquad 2} \\
\multicolumn{2}{c}{x^-} \\
\hline
x \to y^- & 3 & x \\
x & & \otimes \\
y^- & & \\
\otimes & &
\end{array}$$

Figure 11.1: Complete tableau for the clause $[((x \to y) \to x) \to x^-]$

For positive implications we branch to represent the two clauses obtained with the equivalence:

$$\frac{s \to t}{s^- \mid t}$$

The expansion process yields a tree-structured table where each branch represents a clause.

We stop the exploration of a branch if it represents a clashed or a solved clause. If a tableau contains a solved branch, the initial clause is satisfiable since it is entailed by the solved clause represented by the branch. If all branches of a tableau are clashed, the initial clause is unsatisfiable.

A tableau is *complete* if every branch is either clashed or solved. An assignment satisfies the initial clause of a complete tableau if and only if the tableau contains a solved branch whose clause is satisfied by the assignment.

Figure 11.2 shows a complete tableau for the clause $[\neg\neg x \to \neg(x \to \neg y)^-]$. The tableau has 4 branches, three of them clashed and one of them solved. Thus an assignment satisfies the initial clause if and only if it satisfies the solved clause $[x, y^-]$ represented by the solved branch.

Exercise 11.5.1 For each of the following formulas s give a complete tableau for the clause $[s^-]$. Then say whether the formula is valid. If the formula is not valid, give a solved clause such that every assignment satisfying the solved clause dissatisfies the formula.

a) $x \to y \to x$

b) $(x \to y \to z) \to (x \to y) \to x \to z$

c) $(x \to \neg y \to \bot) \to \neg\neg(x \to y)$

d) $\neg\neg x \to \neg y \to \neg(x \to y)$

e) $(x \to y) \to (y \to x) \to z$

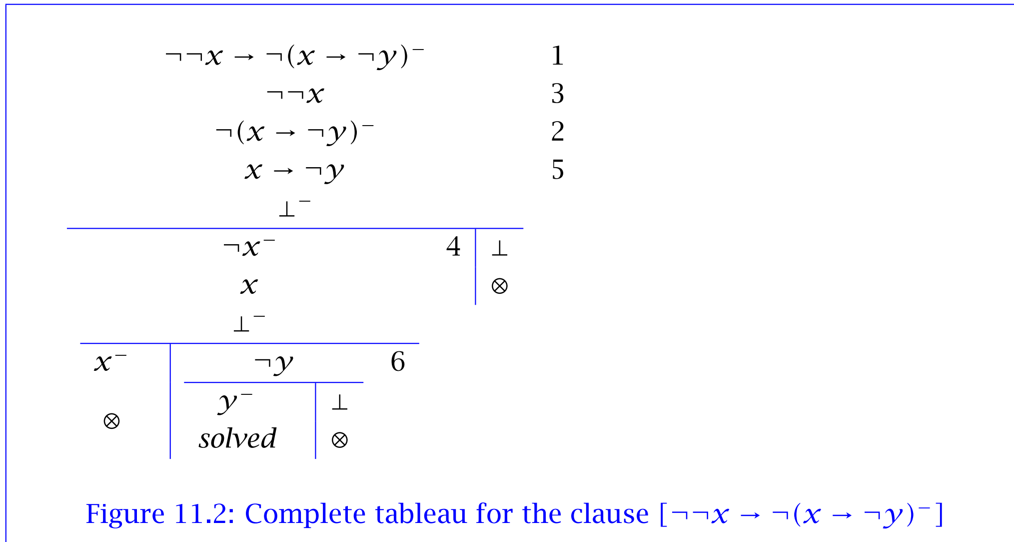

Figure 11.2: Complete tableau for the clause $[\neg\neg x \to \neg(x \to \neg y)^-]$

Hint: Recall Fact 11.2.1.

Exercise 11.5.2 The tableau procedure can be optimized by adding further elimination rules. For negations we may add the following rules:

$$\alpha \models \neg s^+ :: C \;\leftrightarrow\; \alpha \models s^- :: C$$
$$\alpha \models \neg s^- :: C \;\leftrightarrow\; \alpha \models s^+ :: C$$

Prove the correctness of the rules and redo some of the examples of Exercise 11.5.1 using the rules.

Exercise 11.5.3 Assume that the formulas are extended with conjunctions and disjunctions. Give equivalences providing for the elimination of conjunctions and disjunctions.

11.6 DNF Procedure

A list Δ of solved clauses is a **DNF** (disjunctive normal form) for a clause C if the following conditions are satisfied:

1. Ever assignment satisfying a clause in Δ satisfies C.
2. Every assignment satisfying C satisfies some clause in Δ.

Informally, we can say that a DNF for a clause represents the clause as a disjunction of solved clauses.

Lemma 11.6.1 Let Δ be a DNF for C. Then C is satisfiable if and only if Δ is nonempty.

$$
\begin{aligned}
dnf\ C\ nil &= [C] \\
dnf\ C\ (x^+ :: D) &= \text{if } x^- \in C \text{ then } nil \text{ else } dnf\ (x^+ :: C)\ D \\
dnf\ C\ (x^- :: D) &= \text{if } x^+ \in C \text{ then } nil \text{ else } dnf\ (x^- :: C)\ D \\
dnf\ C\ ((s \to t)^+ :: D) &= dnf\ C\ (s^- :: D) +\!\!\!+\ dnf\ C\ (t^+ :: D) \\
dnf\ C\ ((s \to t)^- :: D) &= dnf\ C\ (s^+ :: t^- :: D) \\
dnf\ C\ (\bot^+ :: D) &= nil \\
dnf\ C\ (\bot^- :: D) &= dnf\ C\ D
\end{aligned}
$$

Figure 11.3: Operational specification of the DNF procedure

We can compute a DNF for a clause C by computing a complete tableau for C and collecting the solved clauses obtained from the solved branches of the tableau. Note that a DNF obtained with a complete tableau for a clause C contains only subformulas of formulas in C. We speak of the **subformula property**.

We will now specify a recursive procedure that computes DNFs for clauses. We call the procedure **DNF procedure**. The procedure works with two clauses C and D called **accumulator** and **agenda**. When the procedure starts, the accumulator is empty and the agenda is the initial clause. The procedure uses the agenda as a stack, and in each step processes the topmost formula of the agenda. The procedure collects the signed variables it has seen so far in the accumulator, provided there is no clash. Hence the accumulator is always a solved clause. If a clash is discovered, the procedure yields the empty DNF. If the agenda is empty, the procedure yields the singleton DNF just consisting of the accumulator.

Here is the **declarative specification** of the DNF procedure dnf:

$$\forall C\ D.\ solved\ C \ \to\ dnf\ C\ D \text{ is a DNF for } C +\!\!\!+ D$$

The **operational specification** shown in Figure 11.3 specifies the DNF procedure algorithmically. Note that to every pair C, D of arguments exactly on equation applies. The recursion steps are such that the size of the agenda is decreased. Thus the procedure terminates. The size of the agenda is the sum of the sizes of the formulas on the agenda.

$$
\begin{aligned}
size\ x &:= 1 & size\ nil &:= 0 \\
size\ (s \to t) &:= 1 + size\ s + size\ t & size\ (s^+ :: C) &:= size\ s + size\ C \\
size\ \bot &:= 1 & size\ (s^- :: C) &:= size\ s + size\ C
\end{aligned}
$$

Given the operational specification, we can realize the DNF procedure in a programming language. A direct realization of the DNF procedure in Coq is not possible since the recursion is not structural. However, a bounded variant of the DNF procedure can be realized in Coq using the technique of bounded recursion we have used before for the unification procedure.

Exercise 11.6.2 (Challenge) Write a bounded version of the DNF procedure and prove the following in Coq:

$$\forall C\ D\ n.\ \ \mathit{solved}\ C\ \rightarrow\ \mathit{size}\ D < n \rightarrow \mathit{dnfN}\ n\ C\ D \text{ is a DNF for } C + D$$

11.7 Recursion Trees

We have considered the DNF procedure to better understand the computational aspects of the tableau method. Since the DNF procedure terminates for all clauses C and D, it gives us a recursion tree for all clauses C and D. The recursion tree for C and D represents the recursive calls the DNF procedure performs for C and D. The recursion tree for C and D can also be seen as a transcript of a process that derived a complete tableau for the initial clause $C + D$. It turns out that there is a straightforward formalization of recursion trees that is independent from the DNF procedure. The proof of our main result will be based on recursion trees.

We formalize recursion trees with an inductive type constructor

$$\mathit{rec} : \mathit{clause} \rightarrow \mathit{clause} \rightarrow \mathit{Type}$$

such that the elements of $\mathit{rec}\ C\ D$ are the recursion tree for C and D. The **recursion rules** in Figure 11.4 describe the value constructors for rec. At first view it may be easier to understand rec as an inductive predicate and view the derivations of $\mathit{rec}\ C\ D$ as recursion tree for C and D.

If you look at the recursion rules, it is clear that a type $\mathit{rec}\ C\ D$ has essentially a single member. Irrelevant differences may appear in the proofs of the side conditions for the variable rules.

Using size induction and a script, it is straightforward to construct a function

$$\mathit{provider}\ :\ \forall C\ D.\ \ \mathit{rec}\ C\ D$$

that yields a recursion tree for all clauses C and D.

$$\frac{}{rec\ C\ nil} \qquad \frac{}{rec\ C\ (\bot :: D)} \qquad \frac{rec\ C\ D}{rec\ C\ (\bot^- :: D)}$$

$$\frac{}{rec\ C\ (x :: D)}\ x^- \in C \qquad \frac{rec\ (x :: C)\ D}{rec\ C\ (x :: D)}\ x^- \notin C$$

$$\frac{}{rec\ C\ (x^- :: D)}\ x \in C \qquad \frac{rec\ (x^- :: C)\ D}{rec\ C\ (x^- :: D)}\ x \notin C$$

$$\frac{rec\ C\ (s^- :: D) \qquad rec\ C\ (t :: D)}{rec\ C\ (s \to t :: D)} \qquad \frac{rec\ C\ (s :: t^- :: D)}{rec\ C\ (s \to t^- :: D)}$$

Figure 11.4: Recursion rules

11.8 Assisted Decider for Satisfiability

We can now write functions that recurse on recursion trees. For instance, we can construct a function

$$rec_sat_dec\ :\ \forall C\ D.\ solved\ C \to rec\ C\ D \to\ dec\ (sat\ (C + D))$$

that decides the satisfiability of $C + D$ given a recursion tree for $C + D$. We construct this **assisted decider** with a script using induction on the recursion tree. We speak of an assisted decider since the function is given a recursion tree. We obtain an unassisted decider $\forall C.\ dec\ (sat\ C)$ by combining the assisted decider with the provider.

Lemma 11.8.1 Satisfiability of clauses is decidable.

11.9 Main Results

We shall use the notation $C \vdash s$ for the proposition saying that s is provable in the classical ND calculus in the context representing the clause C (obtained by erasing positive signs and replacing negative signs with negation).

With the assisted decider from the last section we can prove

$$\{sat\ C\} + \{\neg sat\ C\}$$

for every clause C. It turns out that the assisted decider can be modified such that we obtain a proof of

$$\{sat\ C\} + \{C \vdash \bot\} \qquad (11.1)$$

Since we already know

$$C \vdash \bot \;\rightarrow\; \neg sat\ C$$

we obtain the equivalence

$$C \vdash \bot \;\leftrightarrow\; \neg sat\ C$$

Since on both sides entailment can be equivalently expressed as refutation

$$C \vdash s \;\leftrightarrow\; \neg s :: C \vdash \bot$$
$$C \vDash s \;\leftrightarrow\; \neg sat\ (s^- :: C)$$

we obtain the equivalence

$$C \vdash s \leftrightarrow C \vDash s$$

which is a main result of this chapter. It also follows that boolean and classical entailment are decidable.

If you examine the above reasoning, you will see that the decidability of satisfiability is not needed. In fact, the decidability of satisfiability follows from the other facts. The key lemma of the reasoning is (11.1).

Lemma 11.9.1 Classical entailment agrees with boolean entailment.

Lemma 11.9.2 Classical entailment is decidable.

11.10 Refutation Lemma

The proof of the key lemma (11.1) hinges on an assisted decider

$$\forall C\ D.\ \ solved\ C \rightarrow rec\ C\ D \rightarrow\ \{sat\ (C +\!\!\!+ D)\} + \{C +\!\!\!+ D \vdash \bot\}$$

for classical refutability. We said before that this decider can be obtained by modifying the construction of the assisted decider for satisfiability:

$$\forall C\ D.\ \ solved\ C \rightarrow rec\ C\ D \rightarrow\ \{sat\ (C +\!\!\!+ D)\} + \{\neg sat\ (C +\!\!\!+ D)\}$$

It turns out that the construction underlying these deciders can be generalized so that one obtains a decider

$$\forall C\ D.\ \ solved\ C \rightarrow rec\ C\ D \rightarrow\ \{sat\ (C +\!\!\!+ D)\} + \{\rho\ (C +\!\!\!+ D)\}$$

for every abstract refutation predicate ρ satisfying certain properties. Unsatisfiability and classical refutability are two concreted examples for a refutation predicate.

A **refutation predicate** is a predicate ρ on clauses satisfying the **refutation properties** specified in Figure 11.5. It is not difficult to show that unsatisfiability and classical ND refutability are refutation predicates.

$\bot^+ \in C \to \rho\ C$	*ref_False*
$x^+ \in C \to x^- \in C \to \rho\ C$	*ref_clash*
$\rho\ (s^- :: C) \to \rho\ (t^+ :: C) \to \rho\ (s \to t^+ :: C)$	*ref_pos_imp*
$\rho\ (s^+ :: t^- :: C) \to \rho\ (s \to t^- :: C)$	*ref_neg_imp*
$C \subseteq D \to \rho\ C \to \rho\ D$	*ref_weak*
$\rho\ C \to \neg sat\ C$	*ref_sound*

Figure 11.5: Refutation properties

Lemma 11.10.1 $\lambda C.\ \neg sat\ C$ is a refutation predicate.

Lemma 11.10.2 $\lambda C.\ C \vdash \bot$ is a refutation predicate.

Lemma 11.10.3 (Refutation) Let ρ be a refutation predicate. Then we can construct a function $\forall C.\ \{sat\ C\} + \{\rho C\}$.

We remark that the soundness property is not needed for the proof of the refutation lemma. It is however essential for the following two lemmas.

Lemma 11.10.4 Every refutation predicate agress with unsatisfiability.

Lemma 11.10.5 Every refutation predicate is decidable.

In Coq we define a predicate

$$ref_pred : (clause \to Prop) \to Prop$$

characterizing refutation predicates. We use the record command to carry out the definition so that we obtain named projections for the various refutation properties.

12 Intuitionistic Gentzen System

In this chapter we prove two results about the intuitionistic entailment relation:

- Intuitionistic entailment is decidable.
- Intuitionistic entailment is weaker than classical entailment.

Both results are obtained with an analytic proof system. In an analytic proof system, derivations of a goal employ only subformulas of formulas appearing in the goal. Analytic proof systems were invented in the early 1930's by Gerhard Gentzen.

Recall that $\mathcal{N}$ refers to the intuitionistic natural deduction calculus defined in Section 10.5. In this chapter we will use $\vdash$ exclusively for the intuitionistic entailment relation given by $\mathcal{N}$.

12.1 Gentzen System GS

We define an inductive predicate $A \Rightarrow s$ with four proof rules:

$$\frac{}{A \Rightarrow x}\; x \in A \qquad \frac{}{A \Rightarrow u}\; \bot \in A \qquad \frac{A, s \Rightarrow t}{A \Rightarrow s \to t} \qquad \frac{A \Rightarrow s \qquad A, t \Rightarrow u}{A \Rightarrow u}\; s \to t \in A$$

We refer to the rules as **variable rule**, **explosion rule**, **right implication rule**, and **left implication rule**. We refer to the proof system given by the rules as **GS** or as **intuitionistic Gentzen system**. The main difference between $\mathcal{N}$ and GS are the variable rule and the left implication rule. The left implication rule applies implications appearing in the context of the conclusion. Such an application can be seen as a decomposition of an implication appearing in the context.

We will show that $\mathcal{N}$ and GS are equivalent. Proving that GS is sound for $\mathcal{N}$ is straightforward. Proving that GS is complete for $\mathcal{N}$ takes more effort and will be postponed to the next section.

Fact 12.1.1 (Soundness) If $A \Rightarrow s$, then $A \vdash s$.

Proof By induction on the derivation of $A \Rightarrow s$. ■

The most remarkable property of the proof system GS is the fact that it is analytic. A proof system is **analytic** if each of its rules is analytic, and a rule is

analytic if the premises of the rule contain only subformulas of formulas in the conclusion of the rule. One also speaks of the **subformula property**. Clearly, every rule of GS is analytic. In contrast, $\mathcal{N}$ is not analytic since the implication elimination rule of $\mathcal{N}$

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

violates the subformula property. The explosion rule of $\mathcal{N}$ also violates the subformula property.

If a proof system is analytic, then every derivation of a goal contains only subformulas of formulas appearing in the goal.

GS provides for systematic proof search. By proof search we mean a process that given a goal tries to construct in a backward fashion a derivation of the goal. If the variable rule or the explosion rule apply to the current goal, the search ends. Otherwise, the two implication rules of GS provide only finitely many possibilities for reducing the current goal to subgoals. That there are only finitely many possibilities for subgoal reduction is the key difference to $\mathcal{N}$, where the implication elimination rule offers possibly infinitely many possibilities for subgoal reduction (through the choice of the formula s).

Note that every rule of GS is **cumulative** in the sense that the contexts of the premises extend the context of the conclusion. Also note that the rules of GS express constraints on the context of the conclusion exclusively with membership. Both properties also hold for $\mathcal{N}$. Together, the two properties ensure that GS and $\mathcal{N}$ admit weakening.

Fact 12.1.2 (Weakening) If $A \Rightarrow s$ and $A \subseteq B$, then $B \Rightarrow s$.

Proof Follows by induction on the derivation of $A \Rightarrow s$. ∎

The implication rules of GS decompose implications into their constituents when applied backwards. While the right implication rule decomposes claimed implications, the left implication rule decomposes assumed implications. The two implication rules complement each other such that the variable rule suffices as assumption rule for GS. The fact that GS is defined with an assumption rule restricted to variables simplifies some of the proofs in this chapter.

Example 12.1.3 Since $\mathcal{N}$ is defined with an assumption rule, there is a one step derivation of $x \to y \vdash x \to y$. In contrast, there is no one step derivation of $x \to y \Rightarrow x \to y$ in GS. Consider the following derivation of $x \to y \Rightarrow x \to y$

using both implication rules and the variable rule.

$$\dfrac{\dfrac{\dfrac{}{x \to y, x \Rightarrow x} \qquad \dfrac{}{x \to y, x, y \Rightarrow y}}{x \to y, x \Rightarrow y}}{x \to y \Rightarrow x \to y}$$

The technique in Example 12.1.3 generalizes so that a general assumption rule is admissible.

Fact 12.1.4 (Assumption) If $s \in A$, then $A \Rightarrow s$.

Proof Follows by induction on s. ■

The consistency of GS can be easily verified by just looking at the four proof rules. This in contrast to $\mathcal{N}$, where such a verification fails for the non-analytic rules.

Fact 12.1.5 (Consistency) $\emptyset \not\Rightarrow \bot$.

We now show $\neg\neg x \not\Rightarrow x$. Once we have established the equivalence of GS and $\mathcal{N}$, we can conclude from this result that intuitionistic entailment is weaker than classical entailment. To show $\neg\neg x \not\Rightarrow x$, we analyse an assumed derivation of $\neg\neg x \Rightarrow x$ in backward direction and observe that the initial claim reoccurs up to weakening. The situation can be captured with the following lemma.

Lemma 12.1.6 If $A \subseteq [x, \neg\neg x]$ and $A \Rightarrow s$, then s is neither $\bot$ nor $\neg x$.

Proof By induction on the derivation of $A \Rightarrow s$. ■

Lemma 12.1.7 $\neg\neg x \not\Rightarrow x$.

Proof Let $\neg\neg x \Rightarrow x$. This judgment can only be obtained with the left implication rule. Hence $\neg\neg x \Rightarrow \neg x$. Contradiction by Lemma 12.1.6. ■

Exercise 12.1.8 Give a GS derivation of $\neg x \Rightarrow x \to y$ using only the four rules defining GS.

Exercise 12.1.9 Let A be $x \to y,\ y \to z$. Give a GS derivation of $A \Rightarrow x \to z$ using only the four rules defining GS.

Exercise 12.1.10 In this exercise we consider why the explosion rule is necessary for GS to be complete. Give an example of an A and s such that $A \Rightarrow s$ in GS and every derivation of $A \Rightarrow s$ makes use of the explosion rule. Hint: Consider a variant of GS defined without the explosion rule.

12.2 Completeness of GS

We now show that GS is complete for $\mathcal{N}$. For this we have to show that every rule of $\mathcal{N}$ is admissible for GS. We have already shown that the assumption rule is admissible for GS. The admissibility of the implication introduction rule is trivial since it agrees with the right implication rule of GS. The admissibility of the explosion rule and the implication elimination rule of $\mathcal{N}$ follows with the admissibility of cut and weakening for GS. We have already shown that weakening is admissible for GS. So it remains to show that the cut rule

$$\frac{A \Rightarrow s \qquad A, s \Rightarrow u}{A \Rightarrow u}$$

is admissible for GS. We call the formula s in such a cut rule the **cut formula**.

Showing the admissibility of the cut rule is the heart of the completeness proof for GS. Since a direct inductive proof of the admissibility of the cut rule does not go through, we follow Gentzen and generalize the cut rule as follows.

$$\frac{A \Rightarrow s \qquad B \Rightarrow u}{A +\!\!+ (B \setminus s) \Rightarrow u}$$

Recall that $A +\!\!+ (B \setminus s)$ is the concatenation of the lists A and $B \setminus s$, and that the list $B \setminus s$ is obtained from B by removing all occurrences of s. The cut rule can be obtained from the generalized cut rule with $B = A, s$ and weakening.

Lemma 12.2.1 (Generalized Cut) If $A \Rightarrow s$ and $B \Rightarrow u$, then $A +\!\!+ (B \setminus s) \Rightarrow u$.

Proof By induction on the cut formula s (first) and the derivation of $A \Rightarrow s$ (second). This yields two cases for $s = \bot$, three cases for $s = x$ and three cases for $s = s_1 \to s_2$. All the cases are easy except for the case with $s = s_1 \to s_2$ and the right implication rule. For the interesting case we have $A, s_1 \Rightarrow s_2$ and $B \Rightarrow u$ and we need to prove $A +\!\!+ (B \setminus s) \Rightarrow u$ (where $s = s_1 \to s_2$). We will use the following inductive hypotheses for s_1 and s_2:

- For all A, B and u, if $A \Rightarrow s_1$ and $B \Rightarrow u$, then $A, (B \setminus s_1) \Rightarrow u$.
- For all A, B and u, if $A \Rightarrow s_2$ and $B \Rightarrow u$, then $A, (B \setminus s_2) \Rightarrow u$.

We will not need the inductive hypothesis for $A, s_1 \Rightarrow s_2$.

We will prove for all B and u, if $B \Rightarrow u$, then $A +\!\!+ (B \setminus s) \Rightarrow u$ This yields one case for every rule of GS. All the cases are easy except for the case with the left implication rule. For the interesting case, we distinguish two subcases.

Suppose the left implication rule is of the form

$$\frac{B \Rightarrow t_1 \qquad B, t_2 \Rightarrow u}{B \Rightarrow u}$$

where $t_1 \to t_2 \in B$ is not s. The inductive hypotheses for $B \Rightarrow t_1$ and $B, t_2 \Rightarrow u$ give $A +(B \setminus s) \Rightarrow t_1$ and $A +((B, t_2) \setminus s) \Rightarrow u$. Since $t_1 \to t_2$ is not s, it is in $A +(B \setminus s)$. Hence we can complete derive $A +(B \setminus s) \Rightarrow u$ using the left implication rule and weakening as follows:

$$\dfrac{A +(B \setminus s) \Rightarrow t_1 \qquad \dfrac{A +((B, t_2) \setminus s) \Rightarrow u}{A +(B \setminus s), t_2 \Rightarrow u}}{A +(B \setminus s) \Rightarrow u}$$

Finally suppose $s_1 \to s_2 \in B$ and the left implication rule is of the form

$$\dfrac{B \Rightarrow s_1 \qquad B, s_2 \Rightarrow u}{B \Rightarrow u}$$

The inductive hypotheses for $B \Rightarrow s_1$ and $B, s_2 \Rightarrow u$ give $A +(B \setminus s) \Rightarrow s_1$ and $A +((B, s_2) \setminus s) \Rightarrow u$. Recall that $A, s_1 \Rightarrow s_2$. Using the inductive hypothesis for s_1 with $A +(B \setminus s) \Rightarrow s_1$ and $A, s_1 \Rightarrow s_2$ we obtain $(A +(B \setminus s)) +((A, s_1) \setminus s_1) \Rightarrow s_2$ and then $A +(B \setminus s) \Rightarrow s_2$ by weakening. Using the inductive hypothesis for s_2 with $A +(B \setminus s) \Rightarrow s_2$ and $A +((B, s_2) \setminus s) \Rightarrow u$ we obtain

$$(A +(B \setminus s)) +((A +((B, s_2) \setminus s)) \setminus s_2) \Rightarrow u.$$

A final application of weakening yields $A +(B \setminus s) \Rightarrow u$ as desired. ■

Lemma 12.2.2 (Cut) If $A \Rightarrow s$ and $A, s \Rightarrow u$, then $A \Rightarrow u$.

Proof Follows with Lemma 12.2.1. ■

Theorem 12.2.3 (Completeness) If $A \vdash s$, then $A \Rightarrow s$.

Proof By induction on the derivation of $A \vdash s$. The cases for the assumption rule and the right implication rule are straightforward. The case for the explosion rule follows with the cut rule. The case for the left implication rule is most interesting. We have to show that $A \Rightarrow t$ follows from $A \Rightarrow s \to t$ and $A \Rightarrow s$. Here is a derivation using cut with $s \to t$, left implication, weakening and assumption:

$$\dfrac{A \Rightarrow s \to t \qquad \dfrac{\dfrac{A \Rightarrow s}{A, s \to t \Rightarrow s} \qquad \dfrac{}{A, s \to t, t \Rightarrow t}}{A, s \to t \Rightarrow t}}{A \Rightarrow t}$$

Corollary 12.2.4

1. $A \vdash s$ if and only if $A \Rightarrow s$.
2. $\neg\neg x \nvdash x$.

Exercise 12.2.5 Suppose $A \Rightarrow \neg s$ and $A, \neg s \Rightarrow s$. Using only the cut rule and the four rules defining GS derive $A \Rightarrow u$.

Exercise 12.2.6 Determine which of the following rules are admissible for GS. Justify your answers.

$$\frac{A \Rightarrow s \qquad B \Rightarrow u}{A +\!\!+ (B \setminus t) \Rightarrow u} \qquad \frac{A \Rightarrow s \qquad A, t \Rightarrow u}{A, s \to t \Rightarrow u} \qquad \frac{A, s \Rightarrow u \qquad A, \neg s \Rightarrow u}{A \Rightarrow u}$$

$$\frac{A, s \Rightarrow u}{A \Rightarrow u}\ \neg\neg s \in A$$

12.3 Decidability

Every rule of GS has the property that the premises of the rule contain only subformulas of formulas in the conclusion of the rule. This property is known as the **subformula property**. The subformula property of GS has the consequence that every derivation of a judgement $A \Rightarrow s$ contains only formulas that are subformulas of formulas in $A \Rightarrow s$. Based on the subformula property we will show that every judgement $A \Rightarrow s$ is decidable.

A list U of formulas is **subformula-closed** if for every implication $s \to t \in U$ the constituents s and t are both in U. In the following U will always denote a subformula-closed list of formulas.

A **goal** is a pair (A, s). A goal (A, s) is derivable if the judgement $A \Rightarrow s$ is derivable. A goal (A, s) is a **U-goal** if $A \subseteq U$ and $s \in U$.

Fact 12.3.1 For every goal one can compute a subformula-closed list U such that the goal is a U-goal.

Let $\mathcal{P}U$ be the power list of U and $\Gamma := \mathcal{P}U \times U$. Every goal in Γ is a U-goal. We will write the list $\ulcorner A \urcorner^U$ without the subscript U.

Fact 12.3.2 Let (A, s) be a U-goal. Then:

1. $(\ulcorner A \urcorner, s) \in \Gamma$.
2. $A \Rightarrow s$ iff $\ulcorner A \urcorner \Rightarrow s$.

Now we construct with an iterative procedure a list $\Lambda \subseteq \Gamma$ containing all derivable goals in Γ. The procedure starts with $\Lambda = \emptyset$ and one by one adds goals from Γ to Λ following up to list equivalence with a single rule application from the goals already in Λ.

Lemma 12.3.3 One can construct a list $\Lambda \subseteq \Gamma$ such that:

1. If $(A, s) \in \Lambda$, then $A \Rightarrow s$.
2. Λ contains every goal $(A, u) \in \Gamma$ satisfying one of the following conditions:
   a) $\bot \in A$.
   b) $u \in A$ and u is a variable.
   c) There exist s and t such that $u = s \to t$ and $(\ulcorner A, s \urcorner, t) \in \Lambda$.
   d) There exists $s \to t \in A$ such that both (A, s) and $(\ulcorner A, t \urcorner, u)$ are in Λ.

Proof We construct Λ by iteration. We start with $\Lambda = \emptyset$ and add goals in Γ to Λ if this is required by (2). The so obtained list Λ satisfies (1) since every addition of a goal is justified by a rule of GS and weakening. The formalization and verification of this algorithm will be the subject of the next section. ■

Let $\Lambda \subseteq \Gamma$ be a list as specified by Lemma 12.3.3. We show that Λ contains the U-representations of all derivable U-goals.

Lemma 12.3.4 If (A, u) is a derivable U-goal, then $(\ulcorner A \urcorner, u) \in \Lambda$.

Proof By induction on the derivation of $A \Rightarrow u$. We consider the case where $A \Rightarrow u$ is obtained with the left implication rule. Here we are given an implication $s \to t \in A$ and smaller derivations for $A \Rightarrow s$ and $A, t \Rightarrow u$ and need to show the memberships $(\ulcorner A \urcorner, s) \in \Lambda$ and $(\ulcorner \ulcorner A \urcorner, t \urcorner, u) \in \Lambda$. By the inductive hypotheses we have $(\ulcorner A \urcorner, s) \in \Lambda$ and $(\ulcorner A, t \urcorner, u) \in \Lambda$. Thus we have the first membership. The second membership follows since $\ulcorner \ulcorner A \urcorner, t \urcorner = \ulcorner \ulcorner A \urcorner, t \urcorner$ since $\ulcorner A \urcorner, t \equiv A, t$. ■

Theorem 12.3.5 $A \vdash s$ is decidable.

Proof Let A and s be given. By Corollary 12.2.4 it suffices to show that $A \Rightarrow s$ is decidable. We construct a subformula-closed list U such that (A, s) is a U-goal. We also obtain Γ and Λ is described above. By weakening it suffices to show that $\ulcorner A \urcorner \Rightarrow s$ is decidable. Case analysis.

1. $(\ulcorner A \urcorner, s) \in \Lambda$. Then $\ulcorner A \urcorner \Rightarrow s$ by Lemma 12.3.3 (1).
2. $(\ulcorner A \urcorner, s) \notin \Lambda$. To show $\ulcorner A \urcorner \not\Rightarrow s$, we assume $\ulcorner A \urcorner \Rightarrow s$. By weakening we have $A \Rightarrow s$. By Lemma 12.3.4 we have $(\ulcorner A \urcorner, s) \in \Lambda$. Contradiction. ■

Exercise 12.3.6 Let x be a variable.

a) Give a subformula-closed U such that $\neg x \in U$.
b) Give a subformula-closed U such that $[\neg\neg x; x] \subseteq U$.

Exercise 12.3.7 Let x be a variable and U be $[x; x \to x]$.

a) Verify that U is subformula-closed.

b) Verify that the power list $\mathcal{P}U$ of U is $[nil; [x \to x]; [x]; [x; x \to x]]$.

c) Let Γ be $\mathcal{P}U \times U$ and write down the eight U-goals in Γ.

d) Construct the corresponding Λ as described in Lemma 12.3.3.

12.4 Finite Closure Iteration

The formalization and verification of the iteration algorithm underlying Lemma 12.3.3 takes considerable effort. We structure the effort with two reusable abstractions.

The first abstraction concerns functional fixed points that can be computed with functional iteration. Recall that a **fixed point** of a function f is an argument x such that $fx = x$.

Lemma 12.4.1 (Finite Fixed Point Iteration) Let $f : X \to X$ be a function. Then:

1. *Induction.* Let $p : X \to \mathit{Prop}$ and $x \in X$ such that px and $\forall z.\ pz \to p(fz)$. Then $p(f^n x)$ for every number n.
2. *Fixed Point.* Let $\sigma : X \to \mathbb{N}$ and $x \in X$ such that for every number n either $\sigma(f^n x) > \sigma(f^{n+1} x)$ or $f^n x$ is a fixed point of f. Then $f^{\sigma x} x$ is a fixed point of f.

Proof Claim (1) follows by induction on n. For claim (2) one shows by induction on n that either $\sigma x \geq n + \sigma(f^n x)$ or $f^n x$ is a fixed point of f. Thus $\sigma(f^{\sigma x} x) = 0$. By the assumption on σ and x it follows that $f^{\sigma x} x$ is a fixed point of f. ∎

The second abstraction concerns a predicate $\mathit{step} : \mathit{list}\ X \to X \to \mathit{Prop}$ and yields for every list V a minimal list $C \subseteq V$ closed under *step*.

Lemma 12.4.2 (Finite Closure Iteration) Let X be a type with decidable equality, $\mathit{step} : \mathit{list}\ X \to X \to \mathit{Prop}$ be a decidable predicate, and V be a list over X. Then one can construct a list $C \subseteq V$ such that:

1. *Closure.* If $\mathit{step}\ C\ x$ and $x \in V$, then $x \in C$.
2. *Induction.* Let $p : X \to \mathit{Prop}$ such that $\mathit{step}\ A\ x \to px$ for all $A \subseteq p$ and $x \in V$. Then $C \subseteq p$.

Proof We construct C by iteration. We start with $C = \emptyset$ and add $x \in V$ if $\mathit{step}\ C\ x$. Formally, we define a function $F : \mathit{list}\ X \to \mathit{list}\ X$ such that for every list A:

- If there exists an $x \in V$ such that $x \notin A$ and $\mathit{step}\ A\ x$, then $FA = A, x$ for one such x. Otherwise, $FA = A$.

We now define $C := F^v\,\emptyset$ where $v := card\ V$. We use Lemma 12.4.1 to prove the claims about C.

a) $F^n\,\emptyset \subseteq V$. Follows by Lemma 12.4.1 (1).

b) $C \subseteq V$. Follows with (a).

c) *Claim (2)*. Follows by Lemma 12.4.1 (1).

d) $FC = C$. Follows by Lemma 12.4.1 (2) with $\sigma A := card\ V - card\ A$ and (a).

e) *Claim (1)*. Follows with (d). ∎

We now come to the formal proof of Lemma 12.3.3. We obtain Λ with Lemma 12.4.2 where $V := \Gamma$ and *step* formalizes the closure conditions of claim (2) of Lemma 12.3.3. Claim (1) of Lemma 12.3.3 now follows with Lemma 12.4.2 (2). Claim (2) of Lemma 12.3.3 follows with Lemma 12.4.2 (1).

12.5 Realization in Coq

The most interesting issues concerning the Coq realization of this chapter are related to the abstraction for finite closure iteration. The abstraction is provided by the base library since it will be reused for other purposes.

Extensive use of base library

The Coq realization of this chapter uses almost every feature of the base library.

- The inversion tactic *inv* is essential for Fact 12.1.5 and lemmas 12.1.6 and 12.1.7.
- The type class-based automation for decidability is essential.
- The hint-based automation for list membership and list inclusion is essential.
- Setoid rewriting is used with list equivalences.
- List removal is essential for generalized cut lemma.
- Finite closure iteration is used with power lists.
- Finite closure iteration is realized with list cardinality.

Finite closure iteration abstraction realized with module

The finite closure iteration abstraction is defined in the base library using a module *FCI* providing a local name space. This way short names like C can be used in the module. The list Λ and the accompanying lemmas for this chapter are then established as follows:

```
Definition Λ : list goal :=
  FCI.C (step := step) Γ.

Lemma lambda_closure γ :
  γ ∈ Γ → step Λ γ → γ ∈ Λ.
```

```
Proof. apply FCI.closure. Qed.
Lemma lambda_ind (p : goal → Prop) :
  (∀ Delta γ, inclp Delta p → γ ∈ Γ → step Delta γ → p γ) → inclp Λ p.
Proof. apply FCI.ind. Qed.
```

The localized lemmas *lambda_closure* and *lambda_ind* provide short names for the instantiations of the corresponding lemmas in *FCI*. More importantly, the localized lemmas do not unfold the defined name Λ when they are applied.

Definition of step function for finite fixed point iteration

Based on the given predicate *step*, module *FCI* defines a step function *F* to be used with the fixed point iteration. The definition of *F* is based on a function *pick* defined as a Lemma.

```
Lemma pick (A : list X) :
  { x | x ∈ V ∧ step A x ∧ ¬ x ∈ A } + { ∀ x, x ∈ V → step A x → x ∈ A }.
Definition F (A : list X) : list X.
  destruct (pick A) as [[x _]|_]. exact (x::A). exact A.
Defined.
```

There is an interesting interplay between the functions *pick* and *F*. While *pick* comes with an informative type and an opaque definition, *F* comes with a transparent definition based on *pick*. The informative type of *pick* is designed to facilitate proofs about *F*. That the definition of *F* is carried out with a script is a matter of taste.

Step predicate defined with matches

The proof of Lemma 12.3.3 with finite closure iteration is interesting. The step predicate is defined such that decidability of the predicate is easy to prove and inversion of the predicate is feasible (for inversion inductive definitions are most convenient).

```
Definition step (Delta : list goal) (γ : goal) : Prop :=
  let (A,u) := γ in
  match u with
  | Var _ ⇒ u ∈ A
  | Imp s t ⇒ (rep (s::A) U, t) ∈ Delta
  | _ ⇒ ⊥
  end
  ∨
  ∃ v, v ∈ A ∧
  match v with
  | Fal ⇒ ⊤
  | Imp s t ⇒ (A, s) ∈ Delta ∧ (rep (t::A) U, u) ∈ Delta
  | _ ⇒ ⊥
  end
```

The matches avoid existential quantifications for the constituents of implications. Such quantifications would have to be guarded with membership in U to facilitate a decidability proof for *step*. A similar use of match is found in the definition of the predicate for subformula closedness.

```
Definition sf_closed (A : list form) : Prop :=
  ∀ u, u ∈ A → match u with
                | Imp s t ⇒ s ∈ A ∧ t ∈ A
                | _ ⇒ ⊤
              end.
```

12.6 Notes

Analytic proof systems were invented by Gerhard Gentzen [5, 6] in the early 1930's. Gentzen developed analytic proof systems for intuitionistic and classical logic covering the connectives $\wedge$, $\vee$, $\neg$, and $\rightarrow$ and the first-order quantifiers $\forall$ and $\exists$. Gentzen-style systems are called *Gentzen systems*, *sequent systems*, *analytic systems*, or *cut-free systems* in the literature. The word sequent is Kleene's [10] translation of the German word "Sequenz" Gentzen used for lists of formulas.

It is an interesting exercise to add proof rules for conjunctions and disjunctions to GS. This can be done such that all essential properties of GS are preserved. The completeness for a correspondingly extended $\mathcal{N}$ system can be shown by extending the proofs of the existing cut lemmas.

Based on his analytic proof system for intuitionistic logic, Gentzen [5, 6] gave the first proof of the decidability of intuitionist propositional entailment. Gentzen [5, 6] also shows non-derivability of excluded middle.

Many different variants of Gentzen systems are studied in the literature. Our system is closest to the system GKi of Troelstra and Schwichtenberg [19]. GKi is different from GS in that it employs multisets of formulas rather than lists of formulas.

Gentzen [5, 6] starts with a sequent calculus including the cut rule. Gentzen's celebrated *cut elimination theorem* then shows that the uses of the cut rule can be eliminated from the derivation, thus yielding a derivation in an analytic subsystem.

Gentzen's [5, 6] sequent systems come with structural and logical rules. Gentzen's structural rules include weakening and cut. Our system GS has no structural rules. The reason GS is complete without structural rules is the cumulative format of the rules and the use of membership constraints for contexts.

Reading Gentzen's [5, 6] very clear papers gives a good idea of how proof systems were studied in the 1930's. At this time the now standard notions of

lists, abstract syntax, and inductive definitions were unknown. Gentzen did inductions on derivations as inductions on the size of derivations.

Troelstra and Schwichtenberg's [19] text contains an extensive study of various Gentzen systems. Girard et al. [7] is an advanced text developing the connection between Gentzen systems and typed lambda calculi. Kleene's [10] introduction to logic and computation covers Gentzen systems and the decidability of intuitionistic entailment.

Bibliography

[1] F. Baader and W. Snyder. Unification theory. In J.A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume I, pages 447–533. Elsevier Science Publishers, 2001.

[2] E. Eder. Properties of substitutions and unifications. *Journal of Symbolic Computation*, 1(1):31–46, 1985.

[3] G. Frege. *Begriffschrift, eine der arithmetischen nachgebildete Formelsprache des reinen Denkens*. Nebert, Halle, 1879. Also in [9], pages 1–82.

[4] G. Gentzen. Untersuchungen über das logischen Schliessen I. *Mathematische Zeitschrift*, 39:176–210, 1935. Translation in: Collected papers of Gerhard Gentzen, ed. M. E. Szabo, North-Holland, Amsterdam [1969], pp. 68–131.

[5] Gerhard Gentzen. Untersuchungen über das logische Schließen I. *Mathematische Zeitschrift*, 39(1):176–210, 1935.

[6] Gerhard Gentzen. Untersuchungen über das logische Schließen II. *Mathematische Zeitschrift*, 39(1):405–431, 1935.

[7] Jean-Yves Girard, Yves Lafont, and Paul Taylor. *Proof and types*. Cambridge University Press, 1989.

[8] Valery Glivenko. Sur quelques points de la logique de M. Brouwer. *Bulletins de la classe des sciences*, 15:183–188, 1929.

[9] J. van Heijenoort, editor. *From Frege to Gödel: A Source Book in Mathematical Logic, 1879–1931*. Harvard University Press, Cambridge, Massachusetts, 1967.

[10] Stephen Cole Kleene. *Introduction to metamathematics*. Van Nostrand, 1952.

[11] D. Knuth and P. Bendix. Simple word problems in universal algebras simple word problems in universal algebras simple word problems in universal algebras. In J. Leech, editor, *Computational Problems in Abstract Algebra*, pages 263–297. Oxford, 1970.

[12] J-L. Lassez, M.J. Maher, and K. Marriott. Unification revisited. In Mauro Boscarol, Luigia Carlucci Aiello, and Giorgio Levi, editors, *Foundations of Logic and Functional Programming*, volume 306 of *Lecture Notes in Computer Science*, pages 67–113. Springer Berlin Heidelberg, 1988.

[13] Alberto Martelli and Ugo Montanari. An efficient unification algorithm. *ACM Trans. Program. Lang. Syst.*, 4(2):258–282, 1982.

[14] Robin Milner. A theory of type polymorphism in programming. *J. Comput. Syst. Sci.*, 17(3):348–375, 1978.

[15] Mike Paterson and Mark N. Wegman. Linear unification. *J. Comput. Syst. Sci.*, 16(2):158–167, 1978.

[16] Lawrence C. Paulson. Verifying the unification algorithm in LCF. *Sci. Comput. Program.*, 5(2):143–169, 1985.

[17] John Alan Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1):23–41, 1965.

[18] José-Luis Ruiz-Reina, Francisco-Jesús Martín-Mateos, José-Antonio Alonso, and María-José Hidalgo. Formal correctness of a quadratic unification algorithm. *J. Autom. Reasoning*, 37(1-2):67–92, 2006.

[19] A. S. Troelstra and H. Schwichtenberg. *Basic proof theory*. Cambridge University Press, New York, NY, USA, 2nd edition, 2000.