Introduction to Computational Logic

Lecture Notes SS 2011

July 15, 2011

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

Contents

1 Introduction 1

2 Types, Functions, and Equations 3

2.1 Booleans . . . 3
2.2 Proof by Case Analysis and Simplification . . . 5
2.3 Natural Numbers and Structural Recursion . . . 6
2.4 Proof by Structural Induction and Rewriting . . . 8
2.5 Pairs . . . 11
2.6 Iteration . . . 13
2.7 Factorials with Iteration . . . 15
2.8 Lists . . . 15
2.9 Linear List Reversal . . . 17
2.10 Options and Finite Types . . . 18
2.11 Simplifying Subterms . . . 20
2.12 Discussion and Remarks . . . 21
2.13 Tactics Summary . . . 22

3 Reduction and Typing of Terms 23

3.1 Terms . . . 23
3.2 Local Variables . . . 24
3.3 Beta Reduction . . . 25
3.4 Normal Forms and Convertibility . . . 27
3.5 Typing Rules . . . 28
3.6 A Type Checking Algorithm . . . 31
3.7 Plain Definitions and Local Definitions . . . 32
3.8 Inductive Definitions . . . 33
3.9 Matches . . . 35
3.10 Recursive Abstractions . . . 36
3.11 Canonical Form Theorem . . . 37
3.12 The Problem with Non-Terminating Recursion . . . 38
3.13 Universes . . . 39
3.14 A General Recursion Operator . . . 41

4 Propositions and Proofs **43**
4.1 Propositions as Types . . . 43
4.2 Falsity and Negation . . . 46
4.3 Lemmas and Proof Scripts . . . 48
4.4 Tricks of the Trade . . . 50
4.5 Conjunction and Disjunction . . . 52
4.6 Equivalence . . . 55
4.7 Leibniz Equality . . . 56
4.8 Coq's Equality . . . 59
4.9 Existential Quantification . . . 63
4.9.1 Russell's Paradox . . . 64
4.9.2 Cantor's Theorem . . . 65
4.10 Abstract Presentation of The Logical Operations . . . 66
4.11 Last But Not Least . . . 72

5 Dependent Matches and Induction **75**
5.1 Dependent Matches . . . 75
5.2 Boolean Case Analysis . . . 77
5.2.1 Bool and False Are Not Equal . . . 78
5.2.2 Kaminski's Equation . . . 79
5.3 Natural Induction . . . 80
5.4 Primitive Recursion . . . 82
5.5 Projections . . . 84
5.6 Surjections and Countability . . . 86
5.7 Abstract Presentation of the Naturals . . . 88
5.8 Natural Order . . . 89
5.9 Size Induction . . . 92
5.10 Type Constructors with Proper Arguments . . . 93
5.10.1 Inversion . . . 94
5.10.2 Recursion on Proof Terms . . . 96
5.11 Matches at eq . . . 97
5.12 Termination and Divergence . . . 99

6 Sum and Sigma Types **103**
6.1 Division by 2 as Certifying Function . . . 103
6.2 Bounded Search . . . 105
6.3 Least Number Search . . . 107

7 Programming with Dependent Types **111**
7.1 Cascaded Functions . . . 111
7.2 Length-Indexed Lists . . . 115

7.3 Finite Types 118

8 Boolean Logic 123
8.1 Syntax and Semantics of Boolean Logic 123
8.2 Decidability Results 127
8.3 Denotational Completeness 128

9 Quantified Boolean Logic 133
9.1 Syntax 133
9.2 Semantics 134
9.3 Tableaux for Quantified Boolean Formulas 136
9.4 Simulating Tableau in Coq 139
9.5 Soundness and Completeness 142
9.6 Other Interpretations 144

10 Mathematical Assumptions 145
10.1 Classical Assumptions 145
10.2 Extensional Assumptions 146
10.3 Proof Irrelevance 147

11 First-Order Logic 149
11.1 Syntax 149
11.2 Semantics 150

Coq Summary 153
A.1 Commands 153
A.2 Tactics 154

Contents

1 Introduction

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

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

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

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

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

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

1 Introduction

2 Types, Functions, and Equations

In this chapter we take a first look at Coq and its mathematical programming language. We define types and functions for basic data structures like booleans and natural numbers. Based on these definitions, we formulate equational theorems and construct their proofs in interaction with the Coq interpreter.

2.1 Booleans

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

$$\neg : bool \to bool \qquad \wedge : bool \to bool \to bool$$
$$\neg false = true \qquad false \wedge y = false$$
$$\neg true = false \qquad true \wedge y = y$$

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

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

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

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

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

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

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

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

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

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

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

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

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

2.2 Proof by Case Analysis and Simplification

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

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

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

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

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

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

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

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

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

$$\frac{}{\mathit{negb}\,(\mathit{negb}\,\mathit{true}) = \mathit{true}} \qquad \frac{}{\mathit{true} = \mathit{true}}$$

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

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

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

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

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

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

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

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

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

Exercise 2.2.2 (Disjunction) A boolean disjunction $x \vee y$ yields *false* if and only if both x and y are *false*.

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

b) Prove the de Morgan law $\neg(x \vee y) = \neg x \wedge \neg y$ in Coq.

2.3 Natural Numbers and Structural Recursion

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

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

The constructor O represents the number 0, and the constructor S yields the **successor** of a natural number (i.e., $Sn = n + 1$). Expressed with O and S, the natural numbers 0, 1, 2, 3, ... look as follows:

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

We say that the elements of *nat* are obtained by iterating the successor function S on the initial number O. This is a form of recursion. The recursion makes it possible to obtain infinitely many values from finitely many constructors.

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

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

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

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

$$+ : nat \to nat \to nat \qquad \cdot : nat \to nat \to nat$$
$$0 + y = y \qquad 0 \cdot y = O$$
$$Sx + y = S(x + y) \qquad Sx \cdot y = x \cdot y + y$$

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

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

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

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

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

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

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

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

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

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

Coq translates the short form automatically into the long form. One says that the short form is syntactic sugar for the long form. The underline character used in the short form serves as *wildcard pattern* that matches everything. The order of the rules in sugared matches is significant. Without the order sensitivity the second rule in the sugared match would be incorrect.

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

Exercise 2.3.1 Define functions as follows.

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

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

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

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

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

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

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

2.4 Proof by Structural Induction and Rewriting

Consider the proof goal

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

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

$$\frac{}{pO} \qquad \frac{x : nat \quad IHx : px}{p(Sx)}$$

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

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

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

```
Lemma plus_O (x : nat) :  plus x O = x.

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

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

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

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

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

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

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

Note that the proof scripts for the lemmas *plus_S* and *plus_O* are identical. When you run the script for each of the two lemmas, you see that they generate different proofs.

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

```
Lemma plus_com (x y : nat) :  plus x y = plus y x.

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

Note that the lemmas are applied with the rewrite tactic. Given that the definition of *plus* is not symmetric, the commutativity of *plus* is an interesting result. Next we prove that addition is associative.

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

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

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

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

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

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

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

Proof. rewrite plus_asso.  rewrite plus_asso. f_equal.
rewrite (plus_com _ (plus _ _)). rewrite plus_asso. f_equal.
rewrite plus_com. reflexivity. Qed.
```

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

Exercise 2.4.1 Prove Lemma *plus_com* by induction on y.

Exercise 2.4.2 Prove the following lemmas.

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

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

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

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

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

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

2.5 Pairs

Given two values x and y, we can form the ordered pair (x, y). Given two types X and Y, there is a product type $X \times Y$ that contains all pairs whose first component is in X and whose second component is in Y. This leads to the following Coq definition:

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

The function $prod : Type \to Type \to Type$ yields for two types X and Y the product type $X \times Y$. The constructor *pair* is a function that takes two types X and Y and two values $x : X$ and $y : Y$ and yields the pair (x, y). To obtain the pair $(O, true)$, we write *pair nat bool O true*. Here is a series of typings helping you to understand what is going on.

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

As is, we have to write the term *pair nat bool O true* for the pair $(O, true)$. This can be shortened by writing the underline character for the type arguments of *pair*.

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

The underline character leaves it to Coq to derive the type arguments of *pair* from the component arguments of *pair*. We can go one step further and declare the type arguments X and Y of *pair* as implicit. This way Coq always derives the type arguments of *pair* and we don't have to write the underlines.

```
Implicit Arguments pair [X Y].
Check pair O true.
% pair O true : prod nat bool
```

Sometimes it is necessary to suppress the type inference for implicit arguments. The implicit arguments of an identifier can be made explicit by writing @ in front of the identifier.

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

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

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

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

The curly braces around the type arguments declare X and Y as implicit arguments.

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

We prove the so-called eta law for pairs.

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

Proof. destruct p. reflexivity. Qed.
```

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

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

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

Exercise 2.5.1 Prove $swap(swap\ p) = p$ for all pairs p. Note that the tactic *simpl* fails to simplify the goal obtained with *destruct*. Use the tactic *cbv* instead.

Exercise 2.5.2 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 commonly used in mathematics. Define functions

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

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

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

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

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

2.6 Iteration

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

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

The Coq definition is now straightforward:

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

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

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

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

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

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

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

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

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

Definition encode : nat -> Nat := iter.

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

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

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

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

The proof uses the *unfold* tactic to simplify the applications of *encode* and *decode* since *simpl* only simplifies functions that involve a match.

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

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

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

Exercise 2.6.3 Prove the following lemma.

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

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

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

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

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

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

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

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

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

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

2.7 Factorials with Iteration

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

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

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

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

We realize the idea with two definitions.

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

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

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

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

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

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

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

Exercise 2.7.1 Prove the following lemmas.

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

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

2.8 Lists

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

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

All elements of a list must be taken from the same type.

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

The constructor *nil* represents the empty sequence, and the constructor *cons* represents nonempty sequences.

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

Here are functions defining the length, the concatenation, and the reversal of lists.

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

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

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

Using informal notation for lists, we have the following.

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

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

```
Lemma app_nil (X : Type) (xs : list X) : app xs nil = xs.
```

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

Exercise 2.8.1 Prove the following lemmas.

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

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

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

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

2.9 Linear List Reversal

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

If you are familiar with functional programming, you will know that the function *rev* takes quadratic time to reverse a list since each recursion step involves an application of the function *app*. One can write a tail-recursive function that reverses lists in linear time. The trick is to accumultate the elements of the main list in an extra argument.

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

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

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

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

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

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

Exercise 2.9.1 Prove the following lemma.

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

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

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

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

Proof the following lemmas.

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

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

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

2.10 Options and Finite Types

An empty type not having members can be defined as an inductive type with no constructors.

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

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

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

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

The proof is by case analysis on the assumed member *v* of *void*. To prove a claim by case analysis on a member of an inductive type, we need to prove the claim for every constructor of the type. Since *void* has no constructor, the claim follows

vacuously.[1] Every logical system comes with some form of vacuous reasoning. Typically, there is some proposition *False* such that from a proof of *False* one can obtain a proof of everything.

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

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

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

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

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

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

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

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

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

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

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

Exercise 2.10.2 Prove the following lemma.

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

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

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

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

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

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

2.11 Simplifying Subterms

Simplification can be tricky since full simplification of the claim may produce large terms that do not have the structure needed for rewriting. In such a case simplifying only a particular subterm may do the job. Moreover, it is usually a good idea to avoid nested matches since they do not go well with simplification.

As example, we consider two functions that test whether a number is even.

```
Fixpoint evenb (x : nat) : bool :=
match x with
| O => true
| S x' => negb (evenb x')
end.
```

Simplification will work well for *evenb* since there are no nested matches. This is not the case for the following function containing a nested match (hidden by syntactic sugar).

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

Lemma evenb'_negb (n : nat) :
evenb' (S n) = negb (evenb' n).
```

Proof. induction n. reflexivity.
simpl (evenb' (S(S n))). rewrite IHn. rewrite negb_negb. reflexivity. **Qed**.

What makes the proof go through is that we only simplify the subterm $evenb'(S(Sn))$. Try to simplify the entire claim and you will see where the difficulty is.

Exercise 2.11.1 Prove the following lemmas.

Lemma evenb_evenb' (n : nat) : evenb n = evenb' n.

Lemma evenb_SS (n : nat) : evenb (S (S n)) = evenb n.

Lemma evenb_negb (n : nat) : evenb n = negb (evenb (S n)).

Exercise 2.11.2 Identify the nested match in $evenb'$.

2.12 Discussion and Remarks

A basic feature of Coq's language are inductive types. We have introduced inductive types for booleans, natural numbers, pairs, and lists. The elements of inductive types are obtained with so-called constructors. Inductive types generalize the constructor representation of the natural numbers employed in the Peano axioms. Inductive types are also a basic feature of functional programming languagues (e.g., ML, Haskell).

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

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

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

Coq provides for the formulation and proof of theorems. So far we have seen equational theorems. As it comes to proof techniques, we have used simplification, case analysis, induction, and rewriting. Proofs are constructed by proof scripts, which are obtained with commands called tactics. A tactic either resolves a trivial proof goal or reduces a proof goal to one or several subgoals. Proof scripts are constructed in interaction with Coq, where Coq applies the proof rules and maintains the open subgoals.

Proof scripts are programs that construct proofs. To understand a proof, one steps with the Coq interpreter through the script constructing the proof and looks at the proof goals obtained with the tactics. Eventually, we will learn that Coq represents proofs as terms. You may type the command ***Print*** *L* to see the term serving as the proof of a lemma *L*.

For now we concentrate on the basic features of Coq and do not use notational devices such as infix operators. We also ignore Coq's extensive library that comes with theories for many structures (including booleans, natural numbers, pairs, lists, and options).

2.13 Tactics Summary

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

3 Reduction and Typing of Terms

In this chapter we take a look at the core language of constructive type theory, the logic implemented by Coq. This language consists of expressions called terms and comes with the notions of reduction and typing, which are fundamental for computational logic and the theory programming languages. This chapter will not be an easy read since it presents theoretical material involving many technical definitions. However, everything is implemented in Coq and you can freely experiment with examples.

3.1 Terms

We start with the definition of a class of terms. First we assume an infinite set of symbols called **variables**. Then we obtain the set of **terms** with the grammar

$$s,t \;::=\; x \mid \mathsf{T} \mid \forall x:s.t \mid \lambda x:s.t \mid s\,t$$

where the letter x ranges over variables. The grammar employs mathematical notation for terms, which translates to Coq's notation as follows:

$$\begin{array}{rcl} \mathsf{T} & \rightsquigarrow & \mathit{Type} \\ \forall x:s.t & \rightsquigarrow & \mathit{forall}\ x{:}s,\ t \\ \lambda x:s.t & \rightsquigarrow & \mathit{fun}\ x{:}s \Rightarrow t \end{array}$$

A term $\forall x:s.t$ represents a type whose members are functions that take an argument x of type s and yield a result of type t. The result type t may depend on the actual argument. A term $\lambda x:s.t$ descibes a function that takes an argument x of type s and yields the result t. The result t usually depends on the actual argument. A term $s\,t$ represents the application of the function described by s to the argument described by t. We adopt the following conventions.

- Terms of the form $\forall x:s.t$ are called **function types**.
- Terms of the form $\lambda x:s.t$ are called **lambda abstractions**.
- Terms of the form $s\,t$ are called **applications**.

We adopt the following notational conventions.

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

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

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

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

3.2 Local Variables

Terms of the form $\forall x{:}s.t$ or $\lambda x{:}s.t$ introduce a **local variable** x that is visible in the subterm t. We say that $\forall x$ and λx **bind** x **in** t and refer to the symbols $\forall$ and λ as **binders**. A variable is **free in a term** if it has an occurrence in the term that is not in the scope of a **binder**. Here are examples:

- x is not free in $\lambda x{:}y.x$.
- x is free in $(\lambda x{:}y.x)x$.
- x is free in $\lambda x{:}x.x$.
- x is not free in $\lambda x{:}\mathsf{T}.\lambda x{:}x.x$.
- x is free in $x \to x$.
- x is not free in $\forall x{:}y.x \to x$.

A term is **closed** if no variable is free in it. Examples of closed terms are T, $\mathsf{T} \to \mathsf{T}$, $\forall x{:}\mathsf{T}.x \to x$, and $\lambda X{:}\mathsf{T}.\lambda x{:}X.x$. Terms are called **open** if they are not closed.

It is a matter of notation which variables one chooses for the local variables of a term. Thus renaming of local variables does not change the term. Here are examples:

- $\lambda x:X.x$ and $\lambda y:X.y$ are identical.
- $\lambda x:X.\lambda y:X.fxy$ and $\lambda x:X.\lambda z:X.fxz$ are identical.
- $\lambda x:X.\lambda y:X.fxy$ and $\lambda y:X.\lambda x:X.fyx$ are identical.
- $\lambda x:X.\lambda y:X.fxy$ and $\lambda y:X.\lambda x:X.fxy$ are not identical.
- $\lambda X:\mathsf{T}.\lambda x:X.x$ and $\lambda x:\mathsf{T}.\lambda x:x.x$ are identical.

Renaming of local variables is known as **alpha renaming**. That alpha renaming does not change a term can be checked with Coq.

```
Goal (fun (X : Type) (x : X) => x) = (fun (Y : Type) (y : Y) => y).
Proof. reflexivity. Qed.
```

Coq's command *Goal* declares a throw-away lemma that is not given a name. When Coq prints a term, it may take the freedom to rename local variables.

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

Exercise 3.2.1 Check the examples we have given for alpha renaming with Coq. Try to come up with examples of your own.

Exercise 3.2.2 Decide for each pair whether the two terms are identical.

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

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

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

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

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

3.3 Beta Reduction

A term of the form $(\lambda x:s.t)\,u$ is called a **beta redex**. It represents the application of a function given as a lambda abstraction to an argument. A beta redex $(\lambda x:s.t)\,u$ can be simplified to the term t^x_u obtained from t by replacing every free occurrence of the variable x with the term u. We speak of a **beta reduction**. We write the **reduction rule** behind beta reductions as follows:

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

Here are examples of beta reductions:

- $(\lambda x:A.fxa)\,b \rightsquigarrow fba$
- $(\lambda xy:A.x)\,a \rightsquigarrow \lambda y:A.a$
- $(\lambda x:A.x)\,(fab) \rightsquigarrow fab$

- $(\lambda X{:}\mathsf{T}.\lambda x{:}X.x)\,A \rightsquigarrow \lambda x{:}A.x$
- $(\lambda X{:}\mathsf{T}.\lambda g{:}X \to X.\lambda x{:}X.g(gx))\,A \rightsquigarrow \lambda g{:}A \to A.\lambda x{:}A.g(gx)$
- $(\lambda g{:}A \to A.ga)(\lambda x{:}A.fxb) \rightsquigarrow (\lambda x{:}A.fxb)\,a \rightsquigarrow fab$

Beta reduction is a fundamental computation principle first investigated by Alonzo Church in the 1930's.

In Coq, we may use the command *Compute* to perform beta reductions. To mimic the above examples, we use Coq's section device to declare the free variables used in the examples.

```
Section Beta_Reduction.
Variable A : Type.
Variable a b : A.
Variable f : A -> A -> A.
Compute (fun x : A => f x a) b.
Compute (fun x y : A => x) a.
Compute (fun x : A => x) (f a b).
Compute (fun (X : Type) (x : X) => x) A.
Compute (fun (X : Type) (g : X -> X) (x : X) => g (g x)) A.
Compute (fun g : A -> A => g a) (fun x : A => f x x).
Compute fun x => (fun g => g x) (fun x => f x b).
End Beta_Reduction.
```

Step carefully through the commands with Coq to see what happens. For nested abstractions we have used Coq's sugared syntax. You may use the unsugared syntax if you like.

The **substitution** t^x_u performed by a beta reduction $(\lambda x{:}s.t)\,u \rightsquigarrow t^x_u$ replaces every free occurrence of the variable x in the term t with the term u. When doing a substitution by hand, one must avoid **capturing** of free variables in u by a binder in t. So $(\lambda y.x)^x_z = \lambda y.z$ is fine, but $(\lambda y.x)^x_y = \lambda y.y$ is not since y is captured by λy. Capturing can always be avoided by renaming local variables. For instance, we have $(\lambda y.x)^x_y = \lambda z.y$ for every variable $z \neq y$.

We use the opportunity to demonstrate Coq's type inference capabilities. If you omit the type of a variable in a binding construct, Coq will try to infer the type automatically. Run through the following section to see what happens.

```
Section Type_Inference.
Variable A : Type.
Variable a b : A.
Variable f : A -> A -> A.
Check (fun x  => f x a) b.
Check (fun x => x) (f a b).
Check (fun X (x : X) => x) A.
Check (fun X g (x : X) => g (g x)) A.
Check (fun g  => g a) (fun x => f x x).
```

```
Check (fun g => g a) (fun x => f x b).
End Type_Inference.
```

In type theory, a beta redex $(\lambda x:s.t)\,u$ and the term t^x_u obtained by beta reduction always denote the same value.

```
Example beta_equal (X Y : Type) (f : X -> Y) (x : X) :
(fun y => f y) x = f x.

Proof. reflexivity. Qed.

Example beta_equal' (A : Type) (a : A) (f : A -> A -> A) :
(fun g => g a) (fun x => f x) = (fun y h => h y) a f.

Proof. simpl. reflexivity. Qed.
```

The keyword *Example* is a synonym for *Lemma*.

Exercise 3.3.1 Beta reduce the term

```
Compute fun (X : Type) (f : X -> X -> X) (y : X) => (fun x y : X => f x y) y.
```

by hand and check your result with Coq.

Exercise 3.3.2 Give a beta redex where a local variable must be renamed to avoid capturing when the beta redex is reduced.

3.4 Normal Forms and Convertibility

Constructive type theory comes with beta reduction and further reduction rules taking care of definitions and inductive types. The reduction rules are the computation rules of type theory. We now discuss general properties of the reduction rules.

A term s **reduces** to a term t if t can be obtained from s by a finite (possibly empty) series of reduction steps. A term is **reducible** if a reduction step applies to it, and **normal** otherwise. A term t is a **normal form** of a term s if s reduces to t and t is normal.

Type theory is designed such that every well-typed term has a unique normal form. Coq's command *Compute* computes the normal form of a term. We say that a term **evaluates** to its normal form.

The existence of unique normal forms follows from two prominent properties defined as follows:

- **Strong normalization** The process of applying reduction steps to a well-typed term always terminates with a normal form.
- **Confluence** If a term s reduces to two terms t_1 and t_2, then there is always a term u such that both t_1 and t_2 reduce to u.

Strong normalization guarantees that one can compute a normal form for every term by just applying reduction steps as long as this is possible. Confluence ensures that a term has at most one normal form. Taken together, the two properties ensure that every term has a unique normal form.

Two terms are **convertible** if they have the same normal form. Convertible terms always describe the same value in type theory. Coq's tactic *reflexivity* will prove a goal $s = t$ if and only if s and t are convertible. We write $s \approx t$ to say that the terms s and t are convertible.

Strong normalization only holds for well-typed terms. We will spell out the rules for well-typedness in the next section. Note that Coq accepts only well-typed terms.

A precursor of type theory is the **untyped lambda calculus** studied by Church in the 1930s. The terms of the untyped lambda calculus are obtained with the grammar $s ::= x \mid \lambda x.s \mid ss$. Consider the term $\omega := \lambda x.xx$ and note that the term $\omega\omega$ reduces with one beta reduction to itself. This tells us that beta reduction does not terminate on some untyped or ill-typed terms.

Exercise 3.4.1 Compute the normal forms of the following terms.

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

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

Exercise 3.4.2 Explain why convertibility of well-typed terms is an equivalence relation.

3.5 Typing Rules

Only well-typed terms are meaningful in type theory. We will now present a system of *typing rules* that assigns types to terms and thereby defines the notion of well-typedness.

Types are represented as terms. The typing rules derive **typings** $s:t$ where s and t are terms. A term s is called a **type** if the typing $s:\mathsf{T}$ can be derived. Moreover, if a typing $s:t$ can be derived, we say that s **has type** t and that s **is a member of** t. A term s is **well-typed** if it has a type (i.e., there is a term t such that the typing $s:t$ can be derived).

To derive a typing $s:t$, we need assumptions providing the types of the free variables in s and t. Even if we start with closed terms, assumptions will be needed for the local variables of the terms. Things are arranged such that at any point in the derivation of a typing there is at most one assumption per variable.

There is one typing rule for every syntactic form identified by the grammar for terms: $s,t ::= x \mid \mathsf{T} \mid \forall x:s.t \mid \lambda x:s.t \mid s\,t$. The typing rule for variables

is straightforward: We can derive $x : t$ if we have the assumption $x : t$. We may write the typing rule for variables in symbolic notation:

$$\textbf{Var}\quad \frac{}{x:t \Rightarrow x:t}$$

The typing rule for T says that we can derive the typing $\mathsf{T}:\mathsf{T}$.

$$\textbf{Uni}\quad \frac{}{\mathsf{T}:\mathsf{T}}$$

The typing rule for function types looks as follows:

$$\textbf{Fun}\quad \frac{s:\mathsf{T} \qquad x:s \Rightarrow t:\mathsf{T}}{\forall x:s.t:\mathsf{T}}$$

The rule allows us to derive a typing $\forall x:s.t:\mathsf{T}$ from the typings $s:\mathsf{T}$ and $t:\mathsf{T}$, where we get the additional assumption $x:s$ for the derivation of $t:\mathsf{T}$. The rule can only be applied if there is no assumption for x so far. This is no problem since local variables can be renamed.

Here is the typing rule for lambda abstractions:

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

The rule has much in common with the rule for function types. In particular, Lam can only be applied if there is no assumption for x so far.

Finally, we have the typing rule for applications.

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

The rule says that an application $s\,t$ is well-typed if we can derive a function type $\forall x:u.v$ for s such that the argument type u can be derived for t. The type of the application is obtained with the substitution operation already used with the beta reduction rule. Note that $v_t^x \approx (\lambda x:u.v)t$. If the variable x does not occur in the term v, we have $v_t^x = v$ and the rule specializes to the ordinary application rule:

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

There is an additional rule that provides for conversion on the right hand side of a typing.

$$\textbf{Con}\quad \frac{s : t \qquad u : \mathsf{T}}{s : u}\quad t \text{ and } u \text{ are convertible}$$

The **conversion rule** Con allows us to derive a typing $s : u$ from a typing $s : t$ provided t and u are convertible and the typing $u : \mathsf{T}$ can be derived. Here is a contrived example where type checking involves a type conversion with the beta rule.

```
Check fun (X : Type) (f : X -> X) (x : (fun Y : Type => Y) X) => f x.
```

Beta reduction of the type of x is needed so that it agrees with the argument type of f.

Nontrivial type conversions are an essential feature of type theory. The typings involving the function *fin* in Section 2.10 provide interesting examples. In Chapter 4 we will see that nontrivial type conversions are crucial for proof checking.

Because of the conversion rule, every well-typed term has infinitely many types. However, all these types are convertible. In fact, our type theory is designed such that every well-typed term has a unique normal type.

Coq has terms of the form $s : t$ called **type ascriptions**. Type ascriptions give us the possibility to ascribe types to subterms. We can understand a type ascription as syntactic sugar for a beta redex.

$$s : t \quad \rightsquigarrow \quad (\lambda x : t.x)\, s$$

Thus a type ascription $s : t$ has type t and reduces to s. As it comes to type checking, a type ascription $s : t$ impose the obligation to chek that the typing $s : t$ is derivable. Here is an example.

```
Check @nil nat : list ((fun X : Type => X) nat).
```

Note that the conversion rule is needed to derive a type for this term. This way one can proceed from the type *list nat* of the term *@nil nat* to the type *list* $((\lambda X : \mathsf{T}.\, X)\ nat)$ required by the ascription.

Here are the most important properties of well-typed terms.

- **Propagation** If the typing $s : t$ is derivable, then the typing $t : \mathsf{T}$ is derivable.
- **Preservation** If the typing $s : t$ is derivable and s reduces to u, then the typing $u : t$ is derivable. Preservation is also known as *subject reduction*.
- **Unique type** If two typings $s : t$ and $s : u$ are derivable, then the types t and u are convertible. Thus the type of a term is unique up to conversion.

- **Strong normalization** The process of applying reduction steps to a well-typed term always terminates with a normal form.

Exercise 3.5.1 Suppose the typing $s : t$ is derivable and t reduces to u. Explain why the typing $s : u$ is derivable.

Exercise 3.5.2 You can experiment with the typing rules in Coq. Do the following examples by hand and check your results with Coq.

```
Check fun (s : Type)  (t : s −> Type) => forall x : s, t x.
Check fun (s  u : Type)  (t : s −> u) => fun x : s, t x.
Check fun (u : Type)  (v : u −> Type) (s : forall x : u, v x)  (t : u) => s t.
Check fun X : Type => X −> forall X : Type, X.
```

Exercise 3.5.3 Derive the typing

$$\lambda x : \mathsf{T}.\lambda x : x.x \;:\; \forall x : \mathsf{T}.\forall y : x.x$$

Note that a renaming of a local variable is needed so that the rule for lambda abstractions can be applied.

Exercise 3.5.4 Derive the following typings using the assumption $X : \mathsf{T}$.

a) $(\lambda Y : \mathsf{T}.Y)X : \mathsf{T}$

b) $(\lambda X : \mathsf{T}.X)X : \mathsf{T}$

Exercise 3.5.5 Give an ill-typed term that reduces to a well-typed term.

3.6 A Type Checking Algorithm

There is an algorithm that decides whether a typing $s : t$ is derivable. First note that convertibility of well-typed terms is decidable because of strong normalization. Since we also have the unique type property, it suffices to have a type checking algorithm that for a term s returns a type t whenever s has a type. Such a type checking algorithm is implemented by Coq's Check command. We present the type checking algorithm as a system of operational typing rules shown in Figure 3.1. The operational typing rules refine the typing rules such that conversion is only checked for well-typed terms (follows by Propagation).

Exercise 3.6.1 Determine normal types of the following terms and check your results with Coq.

a) $\forall x : \mathsf{T}.x$

b) $\lambda x : \mathsf{T}.\forall y : \mathsf{T}.x \to y$

$$\textbf{Var}\ \frac{}{x:t \Rightarrow x:t} \qquad \textbf{Uni}\ \frac{}{\mathsf{T}:\mathsf{T}}$$

$$\textbf{Funop}\ \frac{s:w_1 \qquad x:s \Rightarrow t:w_2}{\forall x:s.t:\mathsf{T}}\quad w_1 \approx \mathsf{T} \text{ and } w_2 \approx \mathsf{T}$$

$$\textbf{Lamop}\ \frac{s:w \qquad x:s \Rightarrow t:u}{\lambda x:s.t:\forall x:s.u}\quad w \approx \mathsf{T}$$

$$\textbf{Appop}\ \frac{s:w_1 \qquad t:w_2}{s\,t:v_t^x}\quad w_1 \to^* \forall x:u.v \text{ and } w_2 \approx u$$

$$s \approx t \ :\Leftrightarrow\ s \text{ and } t \text{ are convertible}$$
$$s \to^* t \ :\Leftrightarrow\ s \text{ reduces to } t$$

Figure 3.1: Operational typing rules

c) $\lambda f:\mathsf{T} \to \mathsf{T}.\forall x:\mathsf{T}.f\,x$

d) $\lambda xyz:\mathsf{T}.\lambda f:x \to y.\lambda g:y \to z.\lambda w:x.g\,(f\,w)$

Exercise 3.6.2 Convince yourself that the following terms are ill-typed.

a) $\forall x:\mathsf{T}.\forall y:x.y$

b) $\lambda f:\mathsf{T} \to \mathsf{T}.\forall x:f\,\mathsf{T}.x$

c) $\lambda xyz:\mathsf{T}.\lambda f:x \to y.\lambda g:y \to z.$
$\forall p:(x \to z) \to \mathsf{T}.\ p(\lambda w:x.g\,(f\,w)) \to p(\lambda w:x.w)$

3.7 Plain Definitions and Local Definitions

A **plain definition** declares a variable x, gives it a type t, and equates it with a term s.

Definition $x:t := s$.

When you enter a plain definition, Coq checks that the typing $s:t$ can be derived. Once the definition is in effect, the assumption $x:t$ is used for type checking. There is a reduction rule called **delta** that will replace a defined variable by the term it is equated to. Definitions are not recursive.

Coq comes with syntactic sugar for plain definitions. For instance,

```
Definition id (X : Type) (x : X) : X := x.
```

is syntactic sugar for

```
Definition id : forall X : Type, X -> X := fun (X : Type) => fun (x : X) => x.
```

Coq has so-called **let terms** providing for local definitions. A let term takes the fom $let\ x := s\ in\ t$ where x is a variable and s and t are terms. Here are the reduction rule and the typing rule for let terms.

$$let\ x := s\ in\ t \rightsquigarrow t_s^x \qquad \frac{s : u \qquad t_s^x : v}{let\ x := s\ in\ t : v}$$

It is tempting to understand a let term as syntactic sugar for a beta redex.

$$let\ x := s\ in\ t \rightsquigarrow (\lambda x : u.t)\ s \qquad \text{where } u \text{ is the type of } s$$

This understanding is correct as it comes to reduction. However, it is incorrect as it comes to type checking. While the term

```
Check let Id := fun X : Type => X in forall X : Id Type, X.
```

type checks as one would expect, its translation into a beta redex is ill-typed.

```
Check (fun Id : Type -> Type => forall X : Id Type, X) (fun X : Type => X).
% Type error: "X" has type "Id Type" which should be "Type"
```

The problem is in the left hand side of the beta redex where the body X of the function type is required by the typing rule Funop to be of a type that is convertible with T. However, the argument variable X has the type *Id Type* and there is no definition providing for the reduction of *Id Type* to *Type*. This problem does not occur with the let term since the typing rule for let terms expands the local definition in the body $\forall X : Id\,\mathsf{T}.X$ of the let term. This way the expanded body $\forall X : (\lambda X : \mathsf{T}.X)\mathsf{T}.X$ is type checked, which succeeds since $(\lambda X : \mathsf{T}.X)\mathsf{T}$ and T are convertible.

Exercise 3.7.1 Give plain definitions of *negb* and *andb* in Coq notation without using syntactic sugar.

3.8 Inductive Definitions

We now come to inductive definitions. An inductive definition declares a **type constructor** together with finitely many **member constructors**. For instance, the inductive definition

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

declares the type constructor

$$list \;:\; \mathsf{T} \to \mathsf{T}$$

and the member constructors

$$nil \;:\; \forall X{:}\mathsf{T}.\; list\; X$$
$$cons \;:\; \forall X{:}\mathsf{T}.\; X \to list\; X \to list\; X$$

The type constructor of an inductive definition yields types called **inductive types**. The member constructors of an inductive definition yield members of the inductive types obtained with the type constructor of the definition. For instance, the term *cons nat O* (*nil nat*) is a member of the inductive type *list nat*. We speak of the **member constructors of a type constructor** and the **type constructor of an inductive type**. For instance, *nil* and *cons* are the member constructors of the type constructor *list*, and *list* is the type constructor of the inductive type *list nat*.

A **construction** is a term $c\,s_1 \dots s_n$ where c is a constructor and $n \geq 0$. Since constructors are not reducible, a construction $c\,s_1 \dots s_n$ can only be reduced by reducing one of the subterms $s_1, \dots, s_n$.

Syntactically, constructors are variables. We call a term **ground** if each of its free variables is a constructor. The term *list nat*, for instance, is ground but not closed. Reduction of a ground term always yields a ground term.

There are no special typing rules for constructors. Everything is handled by the types an inductive definition assigns to its constructors. For every derivable typing $c\,s_1 \dots s_m : d\,t_1 \dots t_n$ where c and d are constructors, d must be a type constructor and c must be a member constructor of d.

The member constructors of a type constructor are **disjoint** and **injective**. Disjointness means that different member constructors always yield different values. Injectivity means that $c\,s_1 \dots s_n = c\,t_1 \dots t_n$ entails $s_i = t_i$ for all $i \in \{1, \dots, n\}$ if c is a member constructor.

We generalize the above definition of inductive types as follows: An **inductive type** is a type t that reduces to a type $c\,s_1 \dots s_n$ where c is a type constructor. In this case c is the uniquely determined type constructor of t.

We distinguish between the **parametric arguments** and the **proper arguments** of a constructor. The parametric arguments of a constructor appear as **parameters** in the head of the inductive definition declaring the constructor. In our example, *list* takes a parametric argument, *nil* takes a parametric argument, and *cons* takes one parametric and two proper arguments.

Exercise 3.8.1 Recall the inductive definition of pairs in Section 2.5. For each argument of the constructors *prod* and *pair* say whether it is parametric or proper.

3.9 Matches

Matches are terms providing for the case analysis that comes with the fact that every value of an inductive type is obtained by a member constructor of the type. As an example, consider a function that duplicates the first element of a list.

```
Definition duplicate : forall X : Type, list X -> list X :=
fun (X : Type) (xs : list X) =>
match xs with
| nil => xs
| cons x xr => cons x xs
end.
```

In general, a match takes the form

$$s,t,u \;::=\; \cdots \mid \mathit{match}\ s\ \mathit{return}\ t\ \mathit{with}\ R_1 \cdots R_m\ \mathit{end}$$
$$R \;::=\; c\,x_1 \cdots x_n \Rightarrow u$$

where s is the **discriminating term** of the match, t is the **return type** of the match, and $R_1,\ldots,R_m$ are the **rules** of the match. Every rule consists of a member constructor c, a possibly empty sequence of pairwise distinct **local variables** $x_1,\ldots,x_n$, and a **body** u. The clause *return* t specifying the return type of the match may be omitted in Coq. This leaves it to Coq to infer the return type of the match.

The **typing rule for matches** is complex and we will describe it informally. The discriminating term s of a match must have an inductive type, and for every member constructor of the inductive type the match must have exactly one rule. For the **return type** t of the match a typing $t:\mathsf{T}$ must be derivable. Finally, for every rule $c\,x_1 \ldots x_n \Rightarrow u$ of the match, the typing $u:t$ must be derivable, where the types of the proper arguments of c are assumed for the local variables $x_1,\ldots,x_n$.

We now come to the **reduction rule for matches**. A match reduces if the discriminating term s has the form $c\,s_1 \ldots s_n$ where c is a member constructor. In this case the corresponding rule $c\,x_m \cdots x_n \Rightarrow u$ of the match is selected and the match reduces to the term $u\,{}^{x_m \cdots x_n}_{s_m \cdots s_n}$ where $s_m,\ldots,s_n$ are the terms for the proper arguments of c.

Given a concrete type constructor, we can formally state the typing and reduction rules for matches for this type constructor. Here are the rules for *nat*.

$$\frac{s:\mathit{nat} \qquad u:t \qquad x:\mathit{nat} \Rightarrow v:t}{\mathit{match}\ s\ \mathit{return}\ t\ \mathit{with}\ O \Rightarrow u \mid S\,x \Rightarrow v\ \mathit{end}\ :\ t}$$

$$\mathit{match}\ O\ \mathit{return}\ t\ \mathit{with}\ O \Rightarrow u \mid S\,x \Rightarrow v\ \mathit{end} \quad \rightsquigarrow \quad u$$

$$\mathit{match}\ S\,s\ \mathit{return}\ t\ \mathit{with}\ O \Rightarrow u \mid S\,x \Rightarrow v\ \mathit{end} \quad \rightsquigarrow \quad v^x_s$$

Exercise 3.9.1 Assume the inductive definition of *nat* and compute the normal forms of the following terms. Check your results with Coq.

a) $\mathit{match}\ O\ \mathit{with}\ O \Rightarrow S\,O \mid S\,x \Rightarrow x\ \mathit{end}$

b) $\mathit{match}\ S\,O\ \mathit{with}\ O \Rightarrow S\,O \mid S\,x \Rightarrow x\ \mathit{end}$

c) $\mathit{match}\ S\,x\ \mathit{with}\ O \Rightarrow S \mid S\,y \Rightarrow \lambda x{:}\mathit{nat}.\mathit{add}\,y\,x\ \mathit{end}$

Exercise 3.9.2 Give the typing and reduction rules for matches for the type constructors *bool*, *void*, *option*, *prod*, and *list*.

3.10 Recursive Abstractions

Recursive abstractions are terms that describe structurally recursive functions taking an argument from an inductive type. As an example, consider a function computing the length of a list.

```
Definition length : forall X : Type, list X -> nat :=
fun (X : Type) =>
fix f (xs : list X) : nat :=
match xs with
| nil => O
| cons x xr => S (f xr)
end.
```

In general, a recursive abstraction takes the form

$$s,t,u ::= \cdots \mid \mathit{fix}\ f\,(x{:}s){:}\,t := u$$

where f and x are local variables. The variable f refers to the function being described and the variable x refers to the argument of the function being described. The **typing rule for recursive abstractions** is

$$\textbf{Fix}\ \frac{s:\mathsf{T} \qquad x:s \Rightarrow t:\mathsf{T} \qquad f:\forall x{:}s.t,\ x:s \Rightarrow u:t}{\mathit{fix}\ f\,(x{:}s){:}\,t := u\ :\ \forall x{:}s.t}\ s \text{ inductive type}$$

There is an additional **termination constraint** for the uses of f in u. The variable f may only be used in applications $f\,v$ where the term v is obtained from x by stripping off at least one constructor with a match. The **reduction rule for recursive abstractions** takes the form

$$(\mathit{fix}\ f\,(x{:}s){:}\,t := u)(c\,v_1 \ldots v_n) \rightsquigarrow u\,\substack{f \\ \mathit{fix}\ f\,(x:s):t:=u}\,\substack{x \\ c\,v_1 \ldots v_n}$$

where c must be a member constructor. The local variables f and x in the **body** u of the recursive abstraction are replaced with the recursive abstraction and the argument term.

Coq's **recursive definitions** are syntactic sugar for plain definitions and recursive abstractions. A recursive definition with a single argument[1]

Fixpoint $f\ (x : s) : t\ :=\ u.$

translates to a plain definition with a recursive abstraction:

Definition f : forall $x : s, t\ :=$ fix $f\ (x : s) : t := u.$

Coq supports recursive abstractions with more than one argument. In this case, the structural recursion must concern one of the arguments. Coq also supports mutual recursion. Most recursive abstractions with multiple arguments can be reduced to recursive abstractions with a single argument. However, there are advanced uses of recursive abstractions where more than one argument is essential (see Section 5.10.2).

Exercise 3.10.1 Complete the following definition so that it declares an addition function. Use a recursive abstraction with a single argument.

```
Definition addf : nat -> nat -> nat :=
```

Prove that *addf* agrees with the addition function *add* defined in Section 2.3.

Exercise 3.10.2 Assume that the following ground term t is given.

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

Compute the normal forms of the following terms. Write out each reduction step and say whether it is a beta, match, or fix reduction. Check your results with Coq.

a) $t\,O$

b) $t\,(S\,O)$

c) $\lambda z{:}nat.\,t\,z$

d) $\lambda z{:}nat.\,t\,(S\,z)$

3.11 Canonical Form Theorem

A **canonical term** is a term that is ground, well-typed, and normal. A **canonical typing** is a derivable typing $s : t$ where s and t are canonical terms.

Since reduction preserves typings and groundness, we know that the canonical terms are the normal forms of the well-typed ground terms. Thus the canonical terms are the values our type theory computes with. The following theorem formulates the type structure of the values of our type theory.

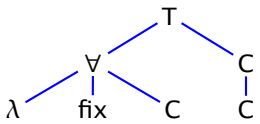
[1] The keyword *Fixpoint* pays tribute to a general theory of recursive functions where recursive functions appear as fixed points of non-recursive functions.

Canonical Form Theorem

Every canonical typing has exactly one of the following forms:

1. $\mathsf{T} : \mathsf{T}$.
2. $\forall x : s.t : \mathsf{T}$.
3. $\lambda x : s.t : \forall x : s.u$.
4. $cs_1 \ldots s_n : \mathsf{T}$ where c is a type constructor.
5. $cs_1 \ldots s_n : \forall x : u.v$ where c is a constructor.
6. $cs_1 \ldots s_m : dt_1 \ldots t_n$ where c is a member constructor of the type constructor d.
7. $fix\ f(x : s) : t := u : \forall x : s.v$.

Graphically, we can summarize the canonical form theorem as follows.



Here are some consequences of the theorem.

- Every canonical term is either T, a function type, an abstraction (either a lambda abstraction or a recursive abstraction), or a construction.
- Every canonical type is either T, a function type, or an inductive type that is a construction.
- Every canonical member of a function type is either an abstraction or a construction.
- Every canonical member of an inductive type is a construction headed by a member constructor of the constructor of the inductive type.

3.12 The Problem with Non-Terminating Recursion

Coq admits only structural recursion so that the application of recursive functions always terminates. This grave restriction on recursion is imposed since non-terminating recursion ruins the logical consistency of the type theory. The argument goes as follows. In a programming language with general recursion we can define a recursive function *push* as follows:

$$push : bool \to bool$$
$$push\ x = negb\ (push\ x)$$

Such a function cannot be defined in Coq. However, we can use Coq's section device to assume that we have such a function.

```
Section Push.
Variable push : bool -> bool.
Variable pusheq : forall x,  push x = negb (push x).
```

The variable *pusheq* is an assumed lemma providing the defining equation of the function *push*. Given the assumptions *push* and *pusheq*, we can now obtain a term $bogus : bool$ satisfying $negb\ bogus = bogus$. This is a logical inconsistency since there is no boolean that is identical with its negation.

```
Definition bogus : bool := push true.
Lemma contradiction :  negb bogus = bogus.
Proof. unfold bogus at 2. rewrite pusheq. unfold bogus. reflexivity. Qed.
End Push.
```

The problem is that *push* yields "meaningless" terms while the logic of type theory assumes that every term is "meaningful" (i.e., every term denotes a value of its type). Seen computationally, the term *bogus* is a non-terminating application of the recursive function *push*. Thus computation will go on forever and not deliver a value in *bool*.

Exercise 3.12.1 For the construction of a meaningless term *bogus*, we can also assume a function $push : nat \to nat$ satisfying $push\ x = S\ (push\ x)$ for all x. Write a section that assumes *push* and proves $S\ bogus = bogus$.

3.13 Universes

We have to say more about the type T. First of all, there is no single type T. Rather, there are infinitely many types T_0, T_1, T_2, ... called **universes** and the typing rules involving T are refined as follows.

$$\textbf{Uni}\ \frac{}{\mathsf{T}_n : \mathsf{T}_{n+1}} \qquad \textbf{Fun}\ \frac{s : \mathsf{T}_n \qquad x : s \Rightarrow t : \mathsf{T}_n}{\forall x : s.t : \mathsf{T}_n}$$

$$\textbf{Con}\ \frac{s : t \qquad u : \mathsf{T}_n}{s : u}\ \ t \text{ and } u \text{ are convertible}$$

Rule Uni arranges the universes into a hierarchy $\mathsf{T}_0 : \mathsf{T}_1 : \mathsf{T}_2 : \cdots$ and thus makes every universe a well-typed term. This way the cycle $\mathsf{T} : \mathsf{T}$ is avoided. Rule Fun populates the universes with function types. Note that every universe is closed under taking function types.

Coq hides the universe levels by printing all universes T_n as *Type*, except for the lowest universe T_0, which is printed as *Set*. As it comes to input, one either writes *Set* for the lowest universe or *Type* for a higher universe. When *Type*

appears in the input, Coq takes care that a universe level can be assigned. Here is an example of a term that does not type check since a universe level cannot be assigned consistently.

```
Definition T : Type := Type.
Check (fun X : T => X) T.
% Error : Universe Inconsistency
```

In principle, a single universe T with $\mathsf{T}:\mathsf{T}$ would be preferable over a hierarchy $\mathsf{T}_0:\mathsf{T}_1:\mathsf{T}_2:\cdots$ of universes since it makes more terms well-typed. However, in 1972 Jean-Yves Girard showed that a type theory with $\mathsf{T}:\mathsf{T}$ is not strongly normalizing.[2]

To come closer to the idea of $\top \in \top$, there are **cumulativity rules** that organize the universes into a cumulative hierarchy $\mathsf{T}_0 \subseteq \mathsf{T}_1 \subseteq \mathsf{T}_2 \subseteq \cdots$.

$$\textbf{CumT}\ \frac{}{\mathsf{T}_m \le \mathsf{T}_n}\ m \le n \qquad \textbf{CumF}\ \frac{u \le v}{\forall x{:}s.u \le \forall x{:}s.v} \qquad \textbf{Cum}\ \frac{s:u}{s:v}\ u \le v$$

With universe cumulativity the **unique type property** is adapted as follows: For every well-typed term s there exists a unique type t such that every derivable typing $s : u$ can be obtained from $s : t$ by conversion and cumulativity (i.e., an application of the typing rule Con followed by an application of the typing rule Cum).

It is not difficult to adapt the type checking algorithm in Figure 3.1 to the cumulative hierarchy of universes. The most significant change is in the Appop rule where $w_2 \approx u$ must be relaxed to $\mathsf{NF}\,w_2 \le \mathsf{NF}\,u$, where $\mathsf{NF}\,s$ stands for the normal form of s.

The following successful checks show that Coq does not work with a single type constructor $list:\mathsf{T}_m \to \mathsf{T}_n$.

```
Check (fun X : Set => list X) : Set -> Set.
Check (fun X : Type => list X) : Type -> Type.
```

For the first check a type constructor $list_0:\mathsf{T}_0 \to \mathsf{T}_0$ is needed. The second check will be succesful with every a type constructor $list_n:\mathsf{T}_n \to \mathsf{T}_n$ where $n > 0$. As it turns out, Coq automatically introduces type constructors $list_n:\mathsf{T}_n \to \mathsf{T}_n$ with associated member constructors nil_n and $cons_n$ for every $n \ge 0$ where such constructors are needed. This feature is known as **universe polymorphism**.

Let us summarize. Girard's paradox tells us that the typing rule $\mathsf{T}:\mathsf{T}$ results in a system that is not strongly normalizing. Thus one needs a less permissive system where all terms terminate. The solution is to replace T with an infinite

[2] Girard gave an involved construction of a term of type $\forall X:\mathsf{T}.X$ not having a normal form. Girard's result is known as **Girard's paradox**.

hierarchy of universes $\mathsf{T}_0 : \mathsf{T}_1 : \mathsf{T}_2 : \cdots$. The resulting system is strongly normalizing but quite restrictive. Thus one adds universe cumulativity and universe polymorphism to obtain a more permissive but still strongly normalizing system. The resulting system is quite complex. Coq hides this complexity by not showing universe levels and assigning universe levels automatically.

Exercise 3.13.1 For every universe T_{n+1} give a closed term s such that $s : \mathsf{T}_{n+1}$ is derivable but $s : \mathsf{T}_n$ is not derivable.

Exercise 3.13.2 Which of the following typings are derivable?

a) $\lambda x{:}\mathsf{T}_1.x \;:\; \forall x{:}\mathsf{T}_1.\mathsf{T}_2$

b) $\lambda x{:}\mathsf{T}_1.x \;:\; \forall x{:}\mathsf{T}_2.\mathsf{T}_3$

c) $\lambda x{:}\mathsf{T}_1.x \;:\; \forall x{:}(\lambda y{:}\mathsf{T}_4.y)\mathsf{T}_1.\mathsf{T}_2$

Exercise 3.13.3 For each of the partially given terms below, fill in the blanks so that the term is well-typed and closed, if this is possible. Give types for the well-typed terms you obtain.

a) $\lambda f : (\forall X{:}\mathsf{T}_1.X \to X).\lambda Y{:}\mathsf{T}_0.f \;\_\_\; Y$

b) $\lambda X{:}\mathsf{T}_0.\lambda c : (\forall Y : \mathsf{T}_1.(Y \to X) \to X).c \;\_\_\; (\lambda z : X.z)$

Exercise 3.13.4 Explain the following type error.

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

Hint: The universe inconsistency becomes apparent if you put universe levels in the definition of *Id* (impossible in Coq).

```
Definition Id : T_{n+1} := forall X : T_n, X -> X.
```

To see what is happening in Coq, activate the display option "display universe levels" and use the command *Print Id*.

Exercise 3.13.5 Is the following typing derivable?

```
(fun X : list Set => X) : list Set -> list Type
```

Use the typing rules to find the answer and check it with Coq.

3.14 A General Recursion Operator

In a functional programming language with general recursion we can define a recursive function *Fix* as follows:

$$Fix : \forall X\,Y{:}\mathsf{T}.\ ((X \to Y) \to X \to Y) \to X \to Y$$

$$Fix\ F\ x = F\ (Fix\ F)\ x$$

Fix is a recursion operator that generalize Coq's recursion operator in that it doesn't require structural recursion. To say more about *Fix*, we assume *Fix* in Coq.

```
Section Fix.
Variable Fix : forall X Y :Type, ((X -> Y) -> X -> Y) -> X -> Y.
Implicit Arguments Fix [X Y].
Variable Fixeq : forall (X Y : Type) (F : (X -> Y) -> X -> Y) (x : X), Fix F x = F (Fix F) x.
```

The assumed function *Fix* can be used to define all recursive functions. The trick is to represent a recursive function we want to define as a non-recursive function that may take the recursive function as an argument. We show this at the example of an addition function.

```
Definition Add : (nat -> nat -> nat) -> nat -> nat -> nat :=
fun f x y => match x with
| O => y
| S x' => S (f x' y)
end.
Lemma Add_fix (x y : nat) :
Fix Add x y = (fix f x := Add f x) x y.
Proof. induction x ; rewrite Fixeq ; simpl. reflexivity. rewrite IHx. reflexivity. Qed.
End Fix.
```

Lemma *Add_fix* states that the assumed recursion operator *Fix* and Coq's recursion operator *fix* yield equivalent functions for the non-recursive function *Add*. We cannot prove this result for a general function F since Coq must see the function F to verify that the recursion obtained with *fix* is structural.

We can now say more about the relationship between recursion and fixpoints, in case you are curious. Consider the defining equation $(Fix\ F)\ x = F(Fix\ F)x$ of *Fix*. It says that the functions $Fix\ F$ and $F(Fix\ F)$ agree on all arguments. In other words, the application of F to $Fix\ F$ does not yield a new function. One says that $Fix\ F$ is a fixpoint of F and that *Fix* is a fixpoint operator (since it yields a fixpoint of F when applied to F). Thus the recursion operator *Fix* can be seen as a fixpoint operator.

Exercise 3.14.1 Assume *Fix* and *Fixeq*. Find a term $bogus : nat$ for which you can prove $S\ bogus = bogus$.

4 Propositions and Proofs

Propositions and proofs are familiar notions from mathematics. A proposition is a statement that may be true or false, and a proof is a rigorous argumentation that a statement is true. The development and study of formal systems of propositions and proofs is a central concern of logic. In this chapter we introduce the formal proof system that comes with Coq's constructive type theory.

4.1 Propositions as Types

In constructive type theory, propositions and proofs are represented as terms. More specifically, propositions are represented as types, and proofs are represented as members of propositions. The representation is such that a term s is a proof of a proposition t if and only if the typing $s:t$ is derivable. Thus type checking provides for proof checking.

To accomodate propositions, Coq's type theory comes with a special universe *Prop*. A term is considered a **proposition** if and only if it has type *Prop*. There are three typing rules for *Prop*.

$$\textbf{PropU}\ \frac{}{\mathit{Prop}:\mathsf{T}_1} \qquad \textbf{PropS}\ \frac{}{\mathit{Prop}\le\mathsf{T}_0} \qquad \textbf{PropQ}\ \frac{s:\mathsf{T}_n \qquad x:s\Rightarrow t:\mathit{Prop}}{\forall x:s.t:\mathit{Prop}}$$

The first rule makes *Prop* a member of the universe T_1. Without this rule the term *Prop* would not be well-typed. The second rule makes *Prop* a subuniverse of T_0. Hence every proposition is a type. The first and the second rule do not give members to *Prop*. This is done by the third rule, which makes a function type a member of *Prop* if the result type of the function type is a member of *Prop*.

Logically, we see a function type $\forall x:s.t$ in *Prop* as a **universal quantification** and more specialized as an **implication** $s\to t$ if s is a proposition and x is not free in t. Seen from this perspective, the typing rule *PropQ* closes the universe of propositions under taking universal quantifications and implications.

A **proof** is a term s such that there is a proposition t such that s has type t. In this case we say that s **proves** t or that s **is a proof of** t. A proposition is **provable** if it has a proof.

So far, every canonical proposition is a universal quantification, and every

canonical proof is a lambda abstraction. Here are examples of canonical propositions and canonical proofs.

```
Definition p1 : forall X : Prop, X -> X
:= fun (X : Prop) (x : X) => x.
Definition p2 : forall X Y : Prop, X -> Y -> X
:= fun (X Y : Prop) (x : X) (y : Y) => x.
Definition p3 : forall X Y Z: Prop, (X -> Y -> Z) -> Y -> X -> Z
:= fun (X Y Z: Prop) (f : X -> Y -> Z) (y : Y) (x : X) => f x y.
```

Here is a non-canonical proposition with a canonical proof.

```
Definition p4 : forall p : bool -> Prop, p true -> p (negb false)
:= fun (p : bool -> Prop) (x : p true) => x.
```

Why can we trust type theoretic proofs? To answer this question we must look at the typing rules (see Section 3.5) that associate proofs with propositions. These are the rules *Lam*, *App*, and *Con*. If we look closely at these rules, we notice that they express generally accepted logical laws. We start with the rule *Lam*, which we depict in a simplified form omitting proof terms.

$$\frac{s : \mathsf{T}_n \qquad x : s \Rightarrow t}{\forall x : s.t}$$

Read logically, *Lam* says that a quantification $\forall x : s.t$ is provable if t is provable under the assumption that x is a member of s. There is the side condition that s is a type. For the special case of an implication, we may write *Lam* as follows:

$$\frac{s : Prop \qquad x : s \Rightarrow t}{s \to t}$$

This version of *Lam* says that an implication $s \to t$ is provable if t is provable under the assumption that there is a proof x of s.

Next we look at the logical reading of the typing rule *App*. Omitting proof terms, we depict *App* as follows.

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

Read logically, *App* says the following: If a quantification $\forall x : s.t$ is provable and u is a member of s, then the proposition t^x_u is provable. For the special case of an implication, we may write *App* as follows:

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

This version of *App* reads as follows: If both an implication $s \to t$ and its **premise** s are provable, then the **conclusion** t of the implication is provable. This logical law is known as **modus ponens** and was already identified by the ancient Greeks.

Finally, we look at the logical reading of the typing rule *Con*. Omitting proof terms, we obtain

$$\frac{s \qquad t : \mathit{Prop}}{t} \quad s \approx t$$

Read logically, *Con* says that if a proposition s is provable, then every convertible proposition t is provable. *Con* is logically sound since reduction preserves the logical meaning of well-typed terms.

There is an important difference between the rule

$$\textbf{Fun} \quad \frac{s : \mathsf{T}_n \qquad x : s \Rightarrow t : \mathsf{T}_n}{\forall x : s.t : \mathsf{T}_n}$$

populating the ordinary universes T_n and the rule

$$\textbf{PropQ} \quad \frac{s : \mathsf{T}_n \qquad x : s \Rightarrow t : \mathit{Prop}}{\forall x : s.t : \mathit{Prop}}$$

populating the universe *Prop*. With *Fun*, quantification over a universe yields a term in a higher universe (e.g., $\forall x : \mathsf{T}_n . x : \mathsf{T}_{n+1}$). With *PropQ*, quantification of a proposition over some universe still yields a proposition (e.g., $\forall x : \mathit{Prop}. x : \mathit{Prop}$). This feature is known as **impredicativity**. Adding the impredicative universe *Prop* to the system of Section 3.13 preserves strong normalization as well as all other properties discussed in Chapter 2.

Exercise 4.1.1 Find proofs for the following propositions. Check your answers with Coq.

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

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

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

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

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

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

4.2 Falsity and Negation

Prop is a universe of types that can be extended with inductive types. Inductive types that are members of *Prop* are called **inductive propositions**. We first introduce an inductive proposition *False* that has no ground proof.

```
Inductive False : Prop := .
```

Technically, *False* is an inductive type in the universe *Prop* that has no member constructor.[1] We prove that from a proof of *False* we can obtain a proof of every proposition.[2]

```
Definition False_vacuous : False -> forall X : Prop, X
:= fun f => match f with end.
```

The vacuous reasoning underlying this proof was discussed in Section 2.10.

With *False* and implication we can define **logical negation**.

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

We prove the characteristic property of negation: If both *not X* and *X* are provable, then *False* is provable. Since *False* is not provable, this means that a proof of *not X* tells us that there is no proof of *X*.

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

Type checking the definition of *not_elim* is interesting. Type inference assigns the type *not X* to the argument variable f. This yields the type *not X* → *not X* for the abstraction. Using the conversion rule we switch to the type *not X* → *X* → *False*. The conversion unfolds the definition of *not* with a delta and a beta reduction.

Here is a proof showing that the proposition $\forall X : Prop.X$ has no proof.

```
Definition pure_False : not (forall X : Prop, X)
:= fun f => f False.
```

The proof tells us that we do not need an inductive definition to obtain a proposition that has no proof.

Why can we be sure that *False* does not have a ground proof? The canonical form theorem tells us that *False* does not have a canonical proof. Thus strong normalization and preservation exclude the existence of a ground proof. The existence of non-ground proofs of *False* is no problem since they rely on assumptions. It is not surprising that with contradicting assumptions one can obtain a proof of *False* (see the function *not_elim* above).

[1] We have already seen an inductive type *void* not having member constructors in Section 2.10.

[2] We rely on Coq's type inference and omit the argument types of the proof. Use the command *Print False_vacuous* to see the types Coq derives for the argument variables of the proof.

Girard's paradox (mentioned in a footnote in Section 3.13) tells us that a type theory with $\mathsf{T}:\mathsf{T}$ has a closed term s of type $\forall X:\mathsf{T}.\, X$ that has no normal form. Now suppose t is a proposition. Then the typing $s\,t:t$ is derivable. Hence s is a spoiler function that yields a proof for every proposition. Thus Girard's paradox tells us that a type theory with $\mathsf{T}:\mathsf{T}$ is logically useless.

We call a proposition **functional** if it is a function type. Since the result type of a functional proposition is a proposition (due to the typing rule *PropQ*), proofs of functional propositions are proof-constructing functions. This operational understanding of proofs can be conveyed by depicting functional propositions as **proof rules**

$$\frac{\mathit{Premise}_1 \quad \ldots \quad \mathit{Premise}_n}{\mathit{Conclusion}}$$

where the **premises** are the propositional argument types of the proposition and the **conclusion** is the result type of the proposition. For instance, the propositions

$\forall X:\mathit{Prop}.\ \mathit{False} \to X$

$\forall X:\mathit{Prop}.\ \mathit{not}\,X \to X \to \mathit{False}$

can be depicted as the proof rules

$$\frac{\bot}{s} \qquad\qquad \frac{\neg s \qquad s}{\bot}$$

where $\bot$ and $\neg$ are the mathematical notations for *False* and *not*. Note that the two proof rules depict the types of the proofs *False_vacuous* and *not_elim* defined above. Given a proof rule and a proof of the proposition the rule depicts, the proof is a function that maps proofs of the premises to a proof of the conclusion. Thus a proof represents a proof rule together with its justification.

False and *not* are predefined in Coq's standard library. In the following we use the definitions from Coq's library, so make sure that the preliminary definitions of this section are not active. For Coq's negation function *not* the prefix operator ~ is defined, so we can write ~~~*False* for *not*(*not*(*not False*)).

```
Check ~~~False.
% ~~~ False : Prop
```

Deactivate the display of notations in the Coq interpreter to see *not* in place of the prefix operator ~. Sometimes the predefined proof

False_ind : $\forall X:\mathit{Prop}.\ \mathit{False} \to X$

corresponding to our proof *False_vacuous* is useful.

We call a function a **predicate** if it yields a proposition if supplied with enough arguments. Negation is our first example of a predicate. We will see many more predicates.

Exercise 4.2.1 Depict each of the following propositions as a proof rule. Also find proofs for the propositions and check your answers with Coq.

a) $\neg\neg\neg\bot$

b) $\forall X : Prop.\ X \to \neg\neg X$

c) $\forall X : Prop.\ \neg\neg\neg X \to \neg X$

d) $\forall X : Prop.\ \neg X \to (\neg X \to X) \to \bot$

Exercise 4.2.2 Prove that the proposition $\neg\neg\bot$ is not provable.

Exercise 4.2.3 Give the typing and reduction rules for matches for the type constructor *False*.

4.3 Lemmas and Proof Scripts

In Chapter 2 we have stated lemmas with Coq's lemma device and obtained their proofs with proof scripts. We will now see that a lemma and its proof amount to a plain definition that equates the name of the lemma to the proof.

Suppose we want to establish a lemma that provides a proof of the proposition $\forall X\,Y : Prop.\ (X \to Y) \to \neg Y \to \neg X$. We choose a name for the lemma and enter the following command.

```
Lemma modus_tollens (X Y : Prop) :
(X -> Y) -> ~Y -> ~X.
```

In response, Coq checks that the term given is a type (not necessarily a proposition) and switches to proof mode. We can now enter a proof script that establishes the lemma.

```
Proof. intros f g x. exact (g (f x)). Qed.
```

Step through the proof script to understand. The *intros* tactic introduces assumptions. The *exact* tactic gives the proof term for the remaining claim. The definition of the negation function *not* is used tacitly. We use the *Print* command to look at Coq's representation of the lemma.

```
Print modus_tollens.
% modus_tollens =
% fun (X Y : Prop) (f : X → Y) (g : ~Y) (x : X) ⇒ g(fx)
% : forall X Y : Prop, (X → Y) → ~Y → ~X
```

We now see that the lemma is in fact represented as a plain definition where the name of the lemma is the defined name, the proposition of the lemma is the type of the defined name, and the proof is the term the defined name is equated to.

Here is our second example of a lemma.

```
Lemma triple_negation (X : Prop) :
~~~X -> ~X.
Proof. intros f x. apply f. intros g. exact (g x). Qed.
Print triple_negation.
% triple_negation =
% fun (X:Prop) (f:~~~X) (x:X) ⇒ f (fun g:~X ⇒ g x)
% : forall X:Prop, ~~~X → ~X
```

Step through the proof script and look at the proof term to understand what is happening. The tactic *apply* takes a function (i.e., a term whose type is a function type up to conversion) and reduces the goal to subgoals for the argument types of the function. The above proof script applies a function f of type $\sim\sim X \to \mathit{False}$ to a goal with the claim *False* and thus obtains a subgoal with the claim $\sim\sim X$.

Recall the proof *False_vacuous* from the previous section. With Coq's lemma device, we can obtain this proof interactively.

```
Lemma False_vacuous' :
False -> forall X : Prop, X.
Proof. intros f. destruct f. Qed.
```

Destructuring a proof of *False* always proves the current goal. For readability one should use the tactic *contradiction* in place of *destruct*.

```
Lemma False_vacuous'' :
False -> forall X : Prop, X.
Proof. intros f. contradiction f. Qed.
```

Like *exact*, the tactic *contradiction* fails if it cannot prove the goal it is applied to.

Type checking (see Section 3.6) and proof construction with a proof script have in common that they derive a typing by applying the typing rules backwards. In both cases one of the two sides of the typing is synthesized:

- When we type check a term s, a typing $s : t$ is derived.
- When we construct a proof for a proposition t, a typing $s : t$ is derived.

A proof script can end with either *Qed* or *Defined*. In both cases the lemma and its proof amount to a plain definition, but in case the proof script ends with *Qed*, there is the extra provision that the name of the lemma does not delta reduce. Try the lemma *modus_tollens*:

```
Compute modus_tollens.
% modus_tollens : forall X Y : Prop, (X → Y) → ~Y → ~X
```

We speak of **opaque** (*Qed*) and **transparent** (*Defined*) lemmas. Usually we work with opaque lemmas since having a name for the proof suffices for the application of the lemma.

When the commands *Qed* and *Defined* are executed, Coq calls its type checker to ensure that the proof term constructed by the proof script interpreter is in fact a proof of the proposition specified by the lemma. This security check relieves the correctness burden on the proof script interpreter. There are rare cases where Coq's proof script interpreter fails to synthesize a proper proof term.

From a theoretical point of view, lemmas and proof scripts are just a convience. Once a lemma is established with *Qed* or *Defined*, Coq represents it as a plain definition. Thus every proposition that can be proven with Coq can be proven with just plain definitions.

From a practical point of view, lemmas and proof scripts are essential. Writing large proof terms by hand is tedious and unrewarding. In contrast, constructing proof terms interactively with the proof script interpreter is fun. The interpreter maintains and displays the open subgoals and automatically takes care of many details. Every open subgoal specifies the claim and the assumptions for a still to be synthesized subterm of the proof term. Moreover, the way the tactics (i.e., the commands of a proof script) reduce a goal to subgoals is in natural correspondence with ordinary mathematical reasoning.

Exercise 4.3.1 State the propositions of Exercises 4.1.1 and 4.2.1 as lemmas and prove the lemmas in interaction with Coq.

Exercise 4.3.2 Prove the following lemma.

```
Lemma ex432 (X Y : Prop) : ~X -> ((X -> Y) -> X) -> False.
```

Hint. Use *contradiction*.

Exercise 4.3.3 Prove the following lemmas.

```
Lemma ex433a (X : Type) (a : X) (r : X -> X -> Prop) :
(forall x y : X, r x y) -> r a a.
Lemma ex433b (X : Type) (a b : X) (p : X -> Prop) :
p a -> (forall q : X -> Prop, q a -> q b) -> p b.
```

4.4 Tricks of the Trade

Since proofs are functions, writing proofs is like writing functions in a functional programming language. In a programming language, one uses local definitions and auxiliary functions to structure a program and to avoid code duplication. The same ideas apply to proving. Consider the following proof.

```
Lemma circuit (X : Prop) :
(X -> ~X) -> (~X -> X) -> False.
Proof. intros f g. apply f.
apply g. intros x. exact (f x x).
apply g. intros x. exact (f x x). Qed.
```

The proof is ugly since it proves the claim X twice. This can be avoided by synthesizing a proof term with a let term that binds the proof of X to a local variable x. We use the tactic *assert* to synthesize the let term.

```
Lemma circuit1 (X : Prop) :
(X -> ~X) -> (~X -> X) -> False.
Proof. intros f g. assert (x : X). apply g. intros x. exact (f x x).
exact (f x x). Qed.
Print circuit1.
% circuit1 =
% fun (X:Prop) (f:X → not X) (g:not X → X) ⇒ let x := g (fun x:X ⇒ f x x) in f x x
% : forall X:Prop, (X → not X) → (not X → X) → False
```

When you step through the proof script, you will see that *assert* creates two subgoals, one for the proposition X, and one for the previous claim where the assumption $x : X$ is added.

Here is another proof for the proposition of *circuit* using an auxiliary lemma.

```
Lemma R {X Y : Prop} :
(X -> X -> Y) -> X -> Y.
Proof. intros f x. exact (f x x). Qed.
Lemma circuit2 (X : Prop) : (X -> ~X) -> (~X -> X) -> False.
Proof. intros f g. exact (R f (g (R f ))). Qed.
```

Note that the proof of *circuit2* applies the auxiliary lemma R twice. Since the arguments X and Y of Lemma R are declared with curly braces, Coq treats them as implicit arguments (see Section 2.5). Below is another proof script that constructs the same proof term with smaller steps.

```
Lemma circuit3 (X : Prop) : (X -> ~X) -> (~X -> X) -> False.
Proof. intros f g. apply (R f). apply g. exact (R f). Qed.
```

Step through the script to see how the apply tactic transforms the proof goal.

Finding a proof for a proposition may be difficult. Consider the following example.

```
Lemma looks_difficult (X Y : Prop) :
X -> (forall p : Prop -> Prop, p Y -> p X) -> Y.
```

We start the proof with the command *intros x f*. This leaves us with the goal

$$\frac{\begin{array}{l} x : X \\ f : \forall p : Prop \to Prop.\ \ pY \to pX \end{array}}{Y}$$

Our only hope for proving Y is to find a function p such that $f\,p$ yields a proof of Y. We choose $p = \lambda Z : Prop.\ Z \to Y$ which yields $f\,p : (Y \to Y) \to X \to Y$. We now have a proof of Y since a proof of $Y \to Y$ is straightforward and we have $x : X$ as an assumption.

```
Proof. intros x f. apply (f (fun Z => Z -> Y)). intros y. exact y. exact x. Qed.
Print looks_difficult.
% looks_difficult =
% fun (X Y : Prop) (x : X) (f : forall p : Prop → Prop, p Y → p X) ⇒
%       f (fun Z : Prop ⇒ Z → Y) (fun y : Y ⇒ y) x
% : forall X Y : Prop, X → (forall p : Prop → Prop, p Y → p X) → Y
```

Exercise 4.4.1 Suppose the proofs of the lemmas *R* and *circuit2* would end with *Defined*. What term would you get with *Compute circuit2*? Check your answer with Coq.

Exercise 4.4.2 Establish the lemmas *circuit*, *R*, and *circuit2* with transparent proofs (i.e., *Defined* in place of *Qed*) and prove the following lemma.

```
Lemma test : circuit = circuit2.
```

Exercise 4.4.3 Prove the following lemma.

```
Lemma ex443 (X : Type) (a b : X) (p : X -> Prop) :
p a -> (forall q : X -> Prop, q b -> q a) -> p b.
```

Exercise 4.4.4 Here are some challenging lemmas.

```
Lemma ex4441 (X : Prop) : ~~(~~X -> X).
Lemma ex4442 (X Y : Prop) : ~~((~Y -> ~X) -> X -> Y).
Lemma ex4443 (X  Y : Prop) : ~~(((X -> Y) -> X) -> X).
```

Hint. Try to introduce as many assumptions as possible. When you get stuck, assert the assumption $F : False$ or apply the predefined lemma *False_ind*. The type of *False_ind* is $\forall X : Prop.\ False \to X$.

4.5 Conjunction and Disjunction

Given two propositions s and t, we can form their **conjunction** $s \wedge t$ (read s and t) and their **disjunction** $s \vee t$ (read s or t). The meaning of conjunctions and disjunctions can be specified as follows:

- $s \wedge t$ is provable if and only if both s and t are provable.
- $s \vee t$ is provable if and only if either s or t is provable.

Our informal account of conjunctions and disjunctions can be formalized with two inductive definitions.

```
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.
```

The type constructors *and* and *or* represent the **logical operations** that come with conjunctions and disjunctions.[3] The member constructors formalize the provability semantics of conjunctions and disjunctions. An **inductive predicate** is a type constructor that is a predicate. The type constructors *and* and *or* are our first examples of inductive predicates.

We prove that conjunction is commutative.

```
Lemma and_commutative (X Y : Prop) :
and X Y -> and Y X.
Proof. intros C. destruct C as [x y]. apply conj. exact y. exact x. Qed.
Print and_commutative.
% and_commutative =
% fun (X Y : Prop) (C : and X Y) ⇒
% match C with
% |conj x y ⇒ conj Y X y x
% end
% : forall X Y : Prop, and X Y → and Y X
```

Note that the destruct tactic is used with a **destructuring pattern** $[x\ y]$ specifying the proper argument variables for the constructor *conj*. Step through the script and look at the proof term to understand. The proof script can be simplified.

```
Lemma and_commutative' (X Y : Prop) :
and X Y -> and Y X.
Proof. intros [x y]. split. exact y. exact x. Qed.
```

This script synthesizes the same proof term the script for the lemma *and_commutative* synthesizes (check with the *Print* command). The match of the proof term is now synthesized by the intros tactic since the destructuring pattern $[x\ y]$ appears here. The tactic *split* applies to all inductive claims with a single member constructor *c* and has the same effect as *apply c*.

Next we prove that disjunction is commutative.

[3] Negation, implication, quantification (universal and existential), and equality also count as logical operations.

```
Lemma or_commutative (X Y : Prop) :
or X Y -> or Y X.
Proof. intros D. destruct D as [x|y].
apply or_intror. exact x. apply or_introl. exact y. Qed.
```

Note that the destruct tactic is used with the destructuring pattern $[x|y]$ specifying the proper argument variables for the constructors *or_introl* and *or_intror*. Once more the proof script can be simplified.

```
Lemma or_commutative' (X Y : Prop) :
or X Y -> or Y X.
Proof. intros [x|y]. right. exact x. left. exact y. Qed.
```

The tactics *left* and *right* apply the first and the second member constructor of an inductive claim with two member constructors.

Next we prove that conjunction is associative.

```
Lemma and_associative (X Y Z : Prop) :
and (and X Y) Z -> and X (and Y Z).
Proof. intros [[x y] z]. split. exact x. split. exact y. exact z. Qed.
```

Note that the proof script uses a nested destructuring pattern.

We observe a surprising coincidence between conjunctions and pairs as defined in Section 2.5:

1. The proposition constructor *and* corresponds to the type constructor *prod*.
2. The proof constructor *conj* corresponds to the pair constructor *pair*.
3. The proof *and_commutative* corresponds to the function *swap*.

Conjunction and disjunction are predefined in Coq's standard library. We will use the predefined versions from now on, so make sure the above trial versions are not active. The predefined versions come with the infix operators $\wedge$ and $\vee$ for *and* and *or*. Coq adopts the notational convention that $\vee$ takes its arguments before $\to$ and after $\wedge$. Thus $s \vee t \wedge u \to s$ is read as $(s \vee (t \wedge u)) \to s$.

Exercise 4.5.1 Prove the following propositions in interaction with Coq.

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

b) $\forall X : Prop.\ \neg(X \wedge \neg X)$

c) $\forall X\, Y : Prop.\ \neg X \vee \neg Y \to \neg(X \wedge Y)$

d) $\forall X\, Y : Prop.\ \neg X \wedge \neg Y \to \neg(X \vee Y)$

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

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

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

h) $\forall X\, Y\, Z : Prop.\ (X \vee Y) \to (X \to Z) \to (Y \to Z) \to Z$

Exercise 4.5.2 Give the typing and reduction rules for the matches for the inductive predicates *and* and *or*.

Exercise 4.5.3 Here are two challenging lemmas.

```
Lemma ex4531 (X Y : Prop) : ~~(X \/ ~X).
Lemma ex4532 (X Y : Prop) : ~~(~(X /\ Y) -> ~X \/ ~Y).
```

4.6 Equivalence

With conjunction and implication we can define a logical operation known as **equivalence**.

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

We write $s \leftrightarrow t$ for a proposition *iff* s t. By definition an equivalence $s \leftrightarrow t$ is provable if and only if s is provable if and only if t is provable (parentheses around the right if and only if). We call two propositions s and t **provably equivalent** if the equivalence $s \leftrightarrow t$ is provable.

The equivalence function *iff* is predefined in Coq's standard library. Coq provides the infix operator $\leftrightarrow$ for *iff*, where $\leftrightarrow$ takes its arguments after the infix operators $\rightarrow$, $\vee$, and $\wedge$. Since equivalence is introduced with a plain definition, the definition of equivalence is handled tacitly by the conversion rule in proofs (analogous to negation). Here is an example.

```
Definition iff_circuit (X : Prop) :
~(X <-> ~X).
Proof. intros [f g]. assert (x : X). apply g. intros x. exact (f x x). exact (f x x). Qed.
```

We introduce the predefined proposition *True* to give a further example for a proof of an equivalence.

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

Lemma True_False : True <-> ~False.
Proof. split. intros _ f. exact f. intros _. exact I. Qed.
```

Exercise 4.6.1 Give the typing and reduction rules for the matches for the type constructor *True*.

Exercise 4.6.2 Prove the following equivalences in interaction with Coq.

a) $\forall X\, Y : Prop.\ \neg(X \vee Y) \leftrightarrow \neg X \wedge \neg Y$

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

c) $\forall X\, Y\, Z : Prop.\ (X \vee Y \rightarrow Z) \leftrightarrow (X \rightarrow Z) \wedge (Y \rightarrow Z)$

Exercise 4.6.3 Prove the following lemmas. The lemmas show that falsity, conjunction, and disjunction can be expressed without constructors.

```
Lemma iff_False :
False <-> forall Z : Prop, Z.
Lemma iff_and (X Y : Prop) :
X /\ Y <-> forall Z : Prop, (X -> Y -> Z) -> Z.
Lemma iff_or (X Y : Prop) :
X \/ Y <-> forall Z : Prop, (X -> Z) -> (Y -> Z) -> Z.
```

Exercise 4.6.4 Find a closed term *dis* for which you can prove the proposition $\forall X\, Y : Prop.\ X \vee Y \leftrightarrow dis\ X\ Y$. Note that the term *dis* cannot contain constructors since it is required to be closed.

4.7 Leibniz Equality

An **equation** is a proposition of the form $s = t$ where s and t are members of the same type. The two tactics we have used for equations in Chapter 2 can be described as follows:

- *reflexivity* proves an equation $s = t$ if s and t are convertible.
- *rewrite e* where e is a proof of an equation $s = t$ rewrites a claim us to ut.

We have seen more complex application of the *rewrite* tactic in Chapter 2 but they all reduce to the basic form specified above. Suppose we have a proof e of an equation $x = y$, the claim $pxxx$, and would like to replace the second occurrence of x with y. Then we can proceed as follows:

$$pxxx \approx (\lambda z.\, pxzx)\, x \rightsquigarrow (\lambda z.\, pxzx)\, y \approx pxyx$$

So the trick is to use the conversion rule before and after the basic rewrite step. We can use the **pattern tactic** to see this course of action in Coq.

```
Lemma demo_basic_rewrite (X : Type) (p : X -> X -> X -> Prop) (x y : X) :
x= y -> p x y x -> p x x x.
intros e A. pattern x at 2. rewrite e. exact A. Qed.
```

We now define equality as follows:

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

With this definition we can write equations $s = t$ as propositions *eql s t* and simulate the tactics *reflexivity* and *rewrite*. To rewrite a claim with an equation, we simply apply a proof of the equation. Here are examples.

```
Lemma eql_ref {X : Type} {x : X} :
eql x x.
Proof. intros p A. exact A. Qed.

Lemma eql_sym {X : Type} {x y : X} :
eql x y -> eql y x.
Proof. intros e. pattern x.  apply e. exact eql_ref. Qed.

Lemma eql_trans {X : Type} {x y z : X} :
eql x y -> eql y z -> eql x z.
Proof. intros e f.  pattern x. apply e. exact f. Qed.

Lemma eql_feq {X Y : Type} (f : X -> Y) {x y : X} :
eql x y -> eql (f x) (f y).
Proof. intros e. pattern x. apply e. exact eql_ref. Qed.
```

Step carefully through the proofs and look at the proof terms. The proof script of *eql_feq* passes through the claims

$$eql\ (f\,x)\ (f\,y) \quad \underset{\text{pattern x}}{\rightsquigarrow} \quad (fun\ z \Rightarrow eql\,(f\,z)\,(f\,y))\,x \quad \underset{\text{apply e}}{\leftarrow} \quad e : x = y$$

$$\rightsquigarrow\ (fun\ z \Rightarrow eql\,(f\,z)\,(f\,y))\,y \quad \underset{\text{exact eql\_ref}}{\approx} \quad eql\ (f\,y)\ (f\,y)$$

and obtains the following proof term:

$$fun\ (X\ Y : Type)\ (f : X \to Y)\ (x\ y : X)\ (e : @eql\ X\ x\ y) \Rightarrow \\ e\ (fun\ z : X \Rightarrow @eql\ Y\ (f\,z)\ (f\,y))\ (@eql\_ref\ Y\ (f\,y))$$

Note that the *pattern* tactic synthesizes the predicate needed for the application of the proof of the equation. As it turns out, the *apply* tactic can often synthesize the predicates for the application of equations by itself. In fact, the *pattern* commands in the above proof scripts can all be dropped. Incidentally, under a **predicate** we understand a function that yields a proposition if applied to sufficiently many arguments.

The defined equality *eql* is known as **Leibniz equality**. This acknowledges the contribution of Gottfried Wilhelm Leibniz who observed that two objects are equal if and only if they have the same properties.

We now know that our type theory can represent equations and rewriting. The key idea is to represent the proof of an equation $s = t$ as a function e of type $\forall p.\, pt \to ps$. Given e and a claim us, we can rewrite the claim to ut by applying the function eu. Beta reduction and the conversion rule generalize this basic form of rewriting to rewriting of inner subterms.

We can prove many interesting facts about equality. We first show that *False* and *True* are not equal.

```
Lemma neql_False_True :
~ eql False True.
Proof. intros e. apply e. exact I. Qed.
```

Next we prove that the boolean values *false* and *true* are not equal. We use an auxiliary function *boolp*[4] and the **conversion tactic change**.

```
Definition boolp (x : bool) : Prop := if x then True else False.
Lemma neql_false_true :
~ eql false true.
Proof. intros e. change (boolp false). apply e. exact I. Qed.
```

A command *change s* converts a claim *t* to *s* provided *s* is a type and *s* and *t* are convertible. In the proof above *change* converts the claim *False* to *boolp false*. This claim can be rewritten to *boolp true* with the equation *false* = *true*. Since *boolp true* evaluates to *True*, the proof can be finished with *exact I*.

Next we prove that the constructor *S* is injective. The trick is to use the predecessor function *pred* to undo *S*.

```
Lemma eql_S_injective (x y : nat) :
eql (S x) (S y) -> eql x y.
Proof. intros e. change (eql (pred(S x)) y). apply e. exact eql_ref. Qed.
```

Here is another proof using the lemma *eql_feq*.

```
Lemma eql_S_injective' (x y : nat) :
eql (S x) (S y) -> eql x y.
Proof. intros e. exact (eql_feq pred e). Qed.
```

Finally we prove that *Sn* and *O* are different for all numbers *n*. To do so, we employ an auxiliary function *iszero* that yields *True* if its argument is *O* and *False* otherwise. We use the **pose tactic** to define *iszero* locally as part of the proof. This local definition appears as a let term in the proof term.

```
Lemma neql_S_O (n : nat) :
~ eql (S n) O.
Proof. intros e.  pose (iszero x := match x with O => True | S _ => False end).
change (iszero (S n)). apply e. exact I. Qed.
```

Note that the local definition of *iszero* must be expanded for the type checking of the proof term.

Exercise 4.7.1 Suppose the terms *s* and *t* are convertible. Give a proof term for the equation *@eql u s t*.

[4] The if-then-else notation appearing in the definition of *boolp* is syntactic sugar for a match discriminating on a boolean value.

Exercise 4.7.2 Find a proof term for the following lemma.

```
Lemma ex472 (X Y : Type) (f g : X -> Y) (x : X) :
eql f g -> eql (f x) (g x).
```

Check your result with Coq.

Exercise 4.7.3 Prove the following lemmas.

```
Lemma ex4731 (X Y : Prop) :
eql X Y -> (X <-> Y).
Lemma ex4732 (X : Type) (x y : X) :
eql x y <-> forall r : X -> X -> Prop, (forall z, r z z) -> r x y.
```

Exercise 4.7.4 Prove the following lemmas.

```
Lemma ex4741 (X : Type) (p : X -> Prop) (x y : X) :
p x -> ~ p y -> ~ eql x y.
Lemma ex4742 (X Y : Type) (f g : X -> Y) (x : X) :
~ eql (f x) (g x) -> ~ eql f g.
```

Exercise 4.7.5 Prove the following lemmas.

```
Lemma ex4751 (X : Type) (x : X) :
~ eql (Some x) None.
Lemma ex4752 (X : Type) (x y : X) :
eql (Some x) (Some y) -> eql x y.
```

4.8 Coq's Equality

Coq's predefined equality is obtained as follows.

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

This gives us two constructors:

$$\begin{aligned} \mathit{eq} &: \forall X : \mathit{Type}.\ X \to X \to \mathit{Prop} \\ \mathit{eq\_refl} &: \forall X : \mathit{Type}.\ X \to \mathit{eq}\ X\ x\ x \end{aligned}$$

The constructor *eq* provides us with propositions $\mathit{eq}\ u\ s\ t$ that represent equations $s = t$ where s and t are members of u. The constructor *eq_refl* provides proofs for all equations $\mathit{eq}\ u\ s\ s$ where s is a member of u. By the conversion rule we thus obtain proofs for all equations $\mathit{eq}\ u\ s\ t$ where s and t are convertible members of u.

The argument X of the constructors *eq* and *eq_refl* is declared implicit so that we can write *eq s t* and *eq_refl s* for equations and proofs. In addition, the infix operator = is provided for the constructor *eq* so that we can write $s = t$ for *eq s t*.

The tactic *reflexivity* used in Chaper 2 establishes a claim by applying the constructor *eq_refl*. We test this information with a lemma.

```
Lemma eq_reflexivity (X : Type) (x : X) :  x = x.
Proof. reflexivity.  Qed.
Print eq_reflexivity.
% eq_reflexivity =
% fun (X : Type) (x : X) ⇒ @eq_refl X x
% : forall (X : Type) (x : X), @eq X x x
```

Next we prove a lemma that justifies the *rewrite* tactic.

```
Lemma eq_rewrite {X : Type} {p : X −> Prop} {x y : X} :
x = y −> p y −> p x.
Proof. intros e. destruct e. intros A. exact A. Qed.
```

The proof matches on the proof of the equation. Since there is only a single proof constructor

$$eq\_refl \;:\; \forall X : Type.\; X \to eq\ X\ x\ x$$

for equations, x and y must be convertible under the given assumptions. We obtain the following transformation of the proof goal:

$$\frac{\begin{array}{l} x : X \\ y : X \\ e : @eq\ X\ x\ y \end{array}}{p\,y \to p\,x} \quad \rightsquigarrow \quad \frac{x : X}{p\,x \to p\,x}$$

destruct e

Now it is straightforward to finish the proof. When we look at the proof term,

```
Print eq_rewrite.
% eq_rewrite =
% fun (X : Type) (p : X → Prop) (x y : X) (e : @eq X x y) ⇒
% match e in (@eq _ _ z) return (p z → p x) with
% | eq_refl ⇒ fun A : p x ⇒ A
% end
% : forall (X : Type) (p : X → Prop) (x y : X), @eq X x y → p y → p x
```

we see that that the match for *eq* takes a form we have have not seen before. We postpone the discussion of the match for *eq* to Section 5.11.

We can now prove that Leibniz equality agrees with Coq's equality.

```
Lemma eql_eq (X : Type) (x y : X) :
eql x y <-> x = y.
Proof. split.
intros e. pattern x. apply e. reflexivity.
intros e. rewrite e. exact eql_ref. Qed.
```

When we look at the proof term, we see that Coq uses a lemma *eq_ind_r* to realize the rewrite step. This lemma is a variant of our lemma *eq_rewrite*.

In simple equational proofs there is not need for the *rewrite* tactic. Instead one can use the basic tactic *destruct e*, which eliminates the right hand side of the equation established by *e*. Here are examples.

```
Lemma eq_sym {X : Type} {x y : X} :
x = y -> y = x.
Proof. intros e. destruct e. reflexivity. Qed.

Lemma eq_trans {X : Type} {x y z : X} :
x = y -> y = z -> x = z.
Proof. intros e f. destruct f. exact e. Qed.

Lemma f_equal {X Y : Type} (f : X -> Y) {x y : X} :
x = y -> f x = f y.
Proof. intros e. destruct e. reflexivity. Qed.
```

The lemmas *eq_sym*, *eq_trans*, and *f_equal* are predefined in Coq. There are also corresponding tactics *symmetry*, *transitivity*, and *f_equal*.

Coq comes with the notation $s <> t$ for negated equations $\sim(s = t)$. Here are three lemmas involving disequations.

```
Lemma neq_False_True :
False <> True.
Proof. intros e. rewrite e. exact I. Qed.

Lemma bool_dis :
false <> true.
Proof. pose (foo (x : bool) := if x then True else False).
intros e. change (foo false). rewrite e. exact I. Qed.

Lemma S_injective (x y : nat) :
S x = S y -> x = y.
Proof. intros e. exact (f_equal pred e). Qed.
```

Exercise 4.8.1 Prove the following lemmas.

```
Lemma ex4811 (X Y : Type) (f g : X -> Y) (x : X) :
f = g -> f x = g x.
Lemma ex4812 (X Y : Prop) :
X = Y -> (X <-> Y).
```

Exercise 4.8.2 Prove the following lemmas showing that equality can be expressed without constructors.

```
Lemma 4821 (X : Type) (x y : X) :
x = y <-> forall p : X -> Prop, p x -> p y.

Lemma 4822 (X : Type) (x y : X) :
x = y <-> forall r : X -> X -> Prop, (forall z, r z z) -> r x y.
```

Exercise 4.8.3 Prove the following lemmas.

```
Lemma ex4831 (X Y : Type) (f g : X -> Y) (x : X) :
f x <> g x -> f <> g.

Lemma ex4832 (X Y : Type) (f : X -> Y) (x y : X) :
f x <> f y -> x <> y.

Lemma ex4833 (X : Type) (p : X -> Prop) (x y : X) :
p x -> ~p y -> x <> y.

Lemma ex4834 (X Y : Prop) :
~(X <-> Y) -> X <> Y.
```

Exercise 4.8.4 Find a proof term for $\forall x\colon nat.\ pred(S\,x) = x$.

Exercise 4.8.5 Different member constructors of an inductive type yield different values provided the inductive type is not a proposition. Lemma *bool_dis* proves this fact for *bool*. Prove analogous results for the member constructors of *nat*, *option*, and *list*.

```
Lemma nat_dis (x : nat) :  S x <> O.
Lemma option_dis (X : Type) (x : X) :  Some x <> None.
Lemma list_dis (X : Type) (x : X) (xs : list X) :   cons x xs <> nil.
```

First do the proofs analogous to the proof of *bool_dis* using auxiliary functions and the *change* tactic. Then do the proofs again using Coq's tactic *discriminate e*, which establishes the current claim if this is justified by disjointness of member constructors and the equation asserted by *e*.

Exercise 4.8.6 The member constructors of an inductive type are always injective provided the inductive type is not a proposition. Lemma *S_injective* proves this fact for *nat*. Prove analogous results for the member constructors of *option*, *pair*, and *list*.

```
Lemma Some_injective (X : Type) (x y : X) :
Some x = Some y -> x = y.
Lemma pair_injective (X Y : Type) (x x' : X) (y y' : Y) :
pair x y = pair x' y'  -> x = x' /\ y = y'.
```

```
Lemma cons_injective (X : Type) (x x' : X) (xs xs' : list X) :
cons x xs = cons x' xs' -> x = x' /\ xs = xs'.
Lemma SS_injective (x y : nat) :
S (S x) = S (S y) -> x = y.
```

First do the the proofs analogous to the proof of *S_injective* using undo functions and the lemma *f_equal*. Then do the proofs again using Coq's tactic *injection e*, which obtains the equations that follow by injectivity of member constructors from the equation asserted by the proof term *e*.

4.9 Existential Quantification

Given a proposition t that type checks under the assumption $x:s$, we can form the **existential quantification** $\exists x:s.\,t$. The meaning of an existential quantification $\exists x:s.\,t$ can be specified as follows:

- $\exists x:s.\,t$ is provable if and only if there exists a member u of s such that t^x_u is provable.

We will represent an existential quantification $\exists x:s.\,t$ as a term $ex\ s\ (\lambda x:s.\,t)$ where *ex* is an inductive predicate of type $\forall\, X:\mathsf{T}.\ (X \to Prop) \to Prop$. This leads to the following inductive definition, which is predefined in Coq.

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

Note that the parameter X is specified in curly braces. This has the effect that the parametric argument X is declared implicit for the constructors *ex* and *ex_intro*. Coq provides a convenient notation for existential quantification:

$$\begin{aligned} exists\ x:s,\ t &\rightsquigarrow ex\ (fun\ x:s \Rightarrow t)\\ exists\ x,\ t &\rightsquigarrow ex\ (fun\ x \Rightarrow t)\end{aligned}$$

We now prove the De Morgan Law for existential quantification.

```
Lemma ex_DeMorgan (X : Type) (p : X -> Prop) :
~ (exists x, p x) <-> forall x, ~ p x.
Proof. split. intros f x A. apply f. apply (ex_intro _ x). exact A.
intros f E. destruct E as [x A]. exact (f x A). Qed.
```

The proof script can be simplified by using the **exists tactic** for the application of the constructor *ex_intro*, and by moving the destructuring pattern for the proof of the existential quantification to the *intros* tactic.

```
Lemma ex_DeMorgan' (X : Type) (p : X -> Prop) :
~ (exists x, p x) <-> forall x, ~ p x.
Proof. split. intros f x A. apply f. exists x. exact A.
intros f [x A]. exact (f x A). Qed.
```

The next lemma shows that it does not matter in which order two variables are existentially quantified.

```
Lemma ex_exchange (X Y : Type) (p : X -> Y -> Prop) :
(exists x, exists y, p x y)  -> exists y, exists x, p x y.
Proof. intros [x [y C]]. exists y. exists x. exact C. Qed.
```

Exercise 4.9.1 Prove the following lemma. It shows that existential quantification generalizes conjunction.

```
Lemma ex_and (X Y : Prop) :
X /\ Y <-> exists x : X, Y.
```

Exercise 4.9.2 Prove that the proposition $\exists X : Prop.\ X \wedge \neg X$ is not provable.

Exercise 4.9.3 Prove the following lemma. It shows that existential quantification can be expressed without constructors.

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

Exercise 4.9.4 Assume a section with $X\ Y : Type$, $p : X \to Prop$, and $r : X \to Y \to Prop$. Prove the following propositions.

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

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

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

Exercise 4.9.5 Explain why the following term does not type check.

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

4.9.1 Russell's Paradox

The next lemma is known as **Russell's paradox**.

```
Lemma Russell (X : Type) (p : X -> X -> Prop) :
~ exists x : X, forall y : X,  p x y <-> ~p y y.
Proof. intros [x A]. exact (iff_circuit _ (A x)). Qed.
```

Russell's paradox has many interesting applications.

1. *Halting problem.* There is no Turing machine that halts on the coding of a Turing machine if and only if this machine does not halt on its own coding. Take for X the set of Turing machines and for pxy the relation "x halts on the coding of y".

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

Exercise 4.9.6 Prove $\forall X\!:\!Type\ \forall p\!:\!X\!\to\!X\!\to\!Prop\ \forall x\!:\!X\ \exists y\!:\!X.\ \neg(pxy \leftrightarrow \neg pyy)$.

Exercise 4.9.7 Prove the following lemma, which says that a barber who shaves exactly those males who don't shave themseves cannot be male.

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

4.9.2 Cantor's Theorem

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

```
Lemma Cantor (X : Type) (f : X -> X -> Prop) :
exists g : X -> Prop, forall x : X, f x <> g.
Proof. exists (fun x => ~ f x x). intros x e. apply (iff_circuit (f x x)).
pattern (f x) at 1. rewrite e. split ; intros A ; exact A. Qed.
```

Note that the witness the proof gives for the predicate g represents the *diagonal set* $\{\, x \in X \mid x \notin fx \,\}$ used in informal proofs of Cantor's Theorem.

Exercise 4.9.8 Prove the following lemma.

```
Definition has_neg (X : Type) : Prop :=
exists g : X -> X, forall x : X, g x <> x.
Definition surjective {X Y : Type} (f : X -> Y) : Prop :=
forall y : Y, exists x : X, f x = y.
Lemma Generalized_Cantor (X Y : Type) :
has_neg Y -> ~ exists f : X -> X -> Y, surjective f.
```

4.10 Abstract Presentation of The Logical Operations

In this chapter we are concerned with propositions and proofs. We have seen the following **logical operations**.

- *Truth* $\top$
- *Falsity* $\bot$
- *Negation* $\neg s$
- *Conjunction* $s \wedge t$
- *Disjunction* $s \vee t$
- *Implication* $s \to t$
- *Equivalence* $s \leftrightarrow t$
- *Equality* $s = t$
- *Universal quantification* $\forall x : s.\, t$
- *Existential quantification* $\exists x : s.\, t$

We started from a type theory where propositions, proofs, implication, and universal quantification are native. We learned that the remaining logical operations can be defined. For the definitions we can in each case either use a plain definition or an inductive definition. The two definitions yield provably equivalent results.

There is a presentation of the logical operations that abstracts away from the particular form of definition. This abstract presentation characterizes every logical operation with a set of rules. Here are the rules for conjunction.

$$\frac{s : \mathit{Prop} \qquad t : \mathit{Prop}}{s \wedge t : \mathit{Prop}} \qquad \frac{s \qquad t}{s \wedge t} \qquad \frac{s \wedge t \qquad s, t \Rightarrow u}{u}$$

The first rule is a **formation rule** that desribes conjunctions syntactically. The second and third rule are proof rules that derive provability of the conclusion from provability of the premises. The rule in the middle is an **introduction rule** saying how conjunctions can be proved. It says that the provability of a conjunction $s \wedge t$ follows from the provability of its constituents s and t. The rule at the right is an **elimination rule** saying how the provability of a conjunction can be used for establishing the provability of further propositions. It says that given the provability of a conjunction $s \wedge t$, the provability of a proposition u can be established under the assumptions that s is provable and that t is provable. We observe that Coq realizes the introduction rule with the tactic *split* and the elimination rule with the tactic *destruct*.

The abstract presentation with formation, introduction, and elimination rules works for all logical operations. Figure 4.1 shows the complete set of introduction and elimination rules together with a conversion rule for propositions.

The conversion rule is essential so that the elimination rules for equations and universal quantifications and the introduction rule for existential quantifications can work as expected. Note that most logical operations have exactly one introduction rule and exactly one elimination rule. The exceptions are truth (no elimination rule), falsity (no introduction rule), and disjunction (two introduction rules).

The abstract presentation of the logical operations with formation, introduction, and elimination rules can be represented in type theory as shown Figure 4.2. For every rule an **abstract logical operation** is assumed whose type expresses the rule. Following the classification of the rules, there are formation, introduction, and elimination operations. Every abstract operation is identified with a name (i.e., a variable). Note that the types of the formation operations are closed, and that the types of the introduction and elimination operations have the name of the formation operation of the respective logical operation as their single free variable. This means that every logical operation is specified independently of the others. Incidentally, the rules in Figure 4.1 violate this modularity in that the introduction rule for negation employ falsity and the elimination rule for equivalence employs implication.

We can prove that the abstract logic operations are equivalent to Coq's predefined logical operations. Here is the proof for existential quantification.

```
Section Existential_Quantification.
Variable ex : forall X : Type, (X -> Prop) -> Prop.
Variable ex_I : forall (X : Type) (p : X -> Prop) (x : X), p x -> ex X p.
Variable ex_E : forall (X : Type) (p : X -> Prop) (Z : Prop),
                    ex X p -> (forall x : X, p x -> Z) -> Z.
Lemma ex_agrees (X : Type) (p : X -> Prop) :
ex X p <-> exists x, p x.
Proof. split.
intros A. apply (ex_E _ _ _ A). intros x B. exists x. exact B.
intros [x A]. exact (ex_I X p x A). Qed.
End Existential_Quantification.
```

We can see the typings in Figure 4.2 as a complete specification of the logical operations. The specification gives us the syntax and the proof rules for the logical operations. So we could accommodate propositions and proofs in type theory by assuming the abstract logical operations in Figure 4.2. Of course, working with assumed logical operations comes with the risk that the assumptions are inconsistent in that they provide for a proof of falsity. This risk can be eliminated by showing that the abstract logical operations can be defined in the underlying type theory.

In Coq's type theory, each logical operation can be defined either with an inductive definition or a plain definition.

$$\frac{}{\top} \qquad \frac{\bot}{u}$$

$$\frac{s \Rightarrow \bot}{\neg s} \qquad \frac{\neg s \quad s}{u}$$

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

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

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

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

$$\frac{}{s = s} \qquad \frac{s = t \quad u\, t}{u\, s}$$

$$\frac{x : s \Rightarrow t}{\forall x : s.\, t} \qquad \frac{\forall x : s.\, t}{(\lambda x : s.\, t)\, u}$$

$$\frac{(\lambda x : s.\, t)\, u}{\exists x : s.\, t} \qquad \frac{\exists x : s.\, t \quad x : s,\, t \Rightarrow u}{u}$$

$$\frac{s}{t} \quad s \approx t$$

Figure 4.1: Introduction and elimination rules

$$
\begin{array}{rcl}
true & : & Prop \\
true\_I & : & true \\
\\
false & : & Prop \\
false\_E & : & \forall Z : Prop.\ false \to Z \\
\\
not & : & Prop \to Prop \\
not\_I & : & \forall X : Prop.\ (\forall Z : Prop.\ X \to Z) \to not\ X \\
not\_E & : & \forall X Z : Prop.\ not\ X \to X \to Z \\
\\
and & : & Prop \to Prop \to Prop \\
and\_I & : & \forall X Y : Prop.\ X \to Y \to and\ X\ Y \\
and\_E & : & \forall X Y Z : Prop.\ and\ X\ Y \to (X \to Y \to Z) \to Z \\
\\
or & : & Prop \to Prop \to Prop \\
or\_IL & : & \forall X Y : Prop.\ X \to or\ X\ Y \\
or\_IR & : & \forall X Y : Prop.\ Y \to or\ X\ Y \\
or\_E & : & \forall X Y Z : Prop.\ or\ X\ Y \to (X \to Z) \to (Y \to Z) \to Z \\
\\
imp & : & Prop \to Prop \to Prop \\
imp\_I & : & \forall X Y : Prop.\ (X \to Y) \to imp\ X\ Y \\
imp\_E & : & \forall X Y : Prop.\ imp\ X\ Y \to X \to Y \\
\\
iff & : & Prop \to Prop \to Prop \\
iff\_I & : & \forall X Y : Prop.\ (X \to Y) \to (Y \to X) \to iff\ X\ Y \\
iff\_E & : & \forall X Y Z : Prop.\ iff\ X\ Y \to ((X \to Y) \to (Y \to X) \to Z) \to Z \\
\\
eq & : & \forall X : Type.\ X \to X \to Prop \\
eq\_I & : & \forall (X : Type)(x : X).\ eq\ X\ x\ x \\
eq\_E & : & \forall (X : Type)(p : X \to Prop)(x y : X).\ eq\ X\ x\ y \to p\,y \to p\,x \\
\\
all & : & \forall X : Type.\ (X \to Prop) \to Prop \\
all\_I & : & \forall (X : Type)(p : X \to Prop).\ (\forall x : X.\ p\,x) \to all\ X\ p \\
all\_E & : & \forall (X : Type)(p : X \to Prop)(x : X).\ all\ X\ p \to p\,x \\
\\
ex & : & \forall X : Type.\ (X \to Prop) \to Prop \\
ex\_I & : & \forall (X : Type)(p : X \to Prop)(x : X).\ p\,x \to ex\ X\ p \\
ex\_E & : & \forall (X : Type)(p : X \to Prop)(Z : Prop).\ ex\ X\ p \to (\forall x : X.\ p\,x \to Z) \to Z
\end{array}
$$

Figure 4.2: Abstract logical operations

When we define a logical operation inductively, the formation operation appears as an inductive predicate and the introduction operations appear as member constructors. The elimination operation is then established as a lemma, which is proved using the match coming with the inductive definition. This means that the formation and the introduction operations of a logical operation carry all necessary information about the operation. Here is an example.

```
Inductive and (X Y : Prop) : Prop :=
| and_I : X −> Y−> and X Y.
Lemma and_E (X Y Z : Prop) :
and X Y −> (X −> Y −> Z) −> Z.
Proof. intros [x y] f. exact (f x y). Qed.
```

When we define a logical operation with a plain definition, we look at the formation and the elimination rule for the operation. The trick is to define a logical operation as a function proving the proposition stated by its elimination operation. We conclude that the formation and the elimination operation of a logical operation carry all necessary information about the operation. Here is an example.

```
Definition and (X Y : Prop) : Prop :=
forall Z : Prop, (X −> Y −> Z) −> Z.
Lemma and_I (X Y : Prop) :
X −> Y−> and X Y.
Proof. intros x y Z f. exact (f x y). Qed.
Lemma and_E (X Y Z : Prop) :
and X Y −> (X −> Y −> Z) −> Z.
Proof. intros A. exact (A Z). Qed.
```

Historically, logicians first analysed proof systems where the underlying type theory was implicit. There was just the requirement that propositions be well-formed. The design and investigation of formal proof systems started in 1879 with Gottlob Frege's Begriffsschrift. The notion of type was invented around 1905 by Bertrand Russell as a means of avoiding the inconsistencies in Frege's Begriffsschrift and Cantor's set theory. Lambda abstractions and beta reduction where studied by Alonzo Church in the 1930's.

For a long time, research concentrated on *first-order proof systems* disallowing quantification over propositions and functions. First-order proof systems come without lambda abstractions and conversion. In a first-order system, the elimation rules for equality and universal quantification and the introduction rule for existential quantification may be formulated as follows.

$$\frac{s = t \qquad u^x_t}{u^x_s} \qquad \frac{\forall x{:}s.t \qquad u : s}{t^x_u} \qquad \frac{t^x_u \qquad u : s}{\exists x{:}s.t}$$

Proof systems accommodating the logical operations with introduction and elimination rules as shown in Figure 4.1 are known as **natural deduction systems**. The first natural deduction system was devised by Gerhard Gentzen in 1935. Gentzen's system is a first-order system. In the 1960's, Dag Prawitz realized that a natural deduction system with implication and universal quantification can express the remaining logical operations provided propositions can quantify over propositions and predicates:

$$
\begin{array}{rcll}
\top & := & \forall X : \mathit{Prop}.\ X \to X & \\
\bot & := & \forall X : \mathit{Prop}.\ X & \\
\neg s & := & s \to \bot & \\
s \wedge t & := & \forall X : \mathit{Prop}.\ (s \to t \to X) \to X & \\
s \vee t & := & \forall X : \mathit{Prop}.\ (s \to X) \to (t \to X) \to X & \\
s \leftrightarrow t & := & (s \to t) \wedge (t \to s) & \\
\exists x : s.\, t & := & \forall X : \mathit{Prop}.\ (\forall x : s.\, t \to X) \to X & \\
s = t & := & \forall p : u \to \mathit{Prop}.\ pt \to ps & \text{where } s, t : u
\end{array}
$$

That propositions can quantify over propositions and predicates requires that the universe of propositions is impredicative. Definitions that exploits the impredicativity of *Prop* are called **impredicative**. The Prawitz definitions shown above are all impredicative.

The first modern proof system based on the propositions as types principle and an impredicative universe of propositions appeared in Jean-Yves Girard's dissertation in 1972.

Exercise 4.10.1 Prove that the abstract characterization of disjunction agrees with Coq's predefined notion of disjunction. Find similiar proofs for negation, equality, and universal quantification.

Exercise 4.10.2 Realize the abstract operations for negation with an inductive definition and a lemma for the elimination operation. Do not use *False*. Do similar proofs for equivalence and universal quantification.

Exercise 4.10.3 Realize the abstract operations for negation with a plain definition and lemmas for the introduction and elimination operation. Do not use *False*. Do similar proofs for equivalence and universal quantification.

Exercise 4.10.4 Assuming abstract logical operations comes with the risk of inconsistency. Prove the inconsistency of the following assumptions for a logical operation *bad*.

```
Section Bad.
Variable bad : Prop -> Prop -> Prop.
Variable badI : forall X Y : Prop, X -> bad X Y.
Variable badE : forall X Y Z : Prop, bad X Y -> (X -> Y -> Z) -> Z.
Lemma inconsistent : False.
Proof. Your proof goes here. Qed.
End Bad.
Check inconsistent.
```

4.11 Last But Not Least

In Mathematics, one assumes that every proposition is either false or true. Consequently, if X is proposition, the propositions $X \vee \neg X$ and $\neg\neg X \to X$ must be true. This leads to the question whether the proposition

```
Definition XM : Prop := forall X : Prop, X \/ ~X.
```

known as **excluded middle** is provable. The answer is no. In fact, neither *XM* nor its negation $\neg XM$ are provable.

Type theory is designed such that every provable proposition is true. From the unprovability of excluded middle we learn that there are true propositions that are unprovable. Here is another true but unprovable proposition known as **double negation**.

```
Definition DN : Prop := forall X : Prop, ~~X -> X.
```

We can show that *XM* and *DN* are provably equivalent. Thus *XM* is unprovable if and only if *DN* is unprovable.

```
Lemma XP_DN : XM <-> DN.
Proof. split ; intros A X.
intros f. destruct (A X) as [B|C]. exact B. contradiction (f C).
apply A. intros f. apply f. right. intros x. apply f. left. exact x. Qed.
```

Coq's type theory is designed such that assuming basic mathematical facts such as *XM* does not lead to inconsistency (i.e., a proof of *False*). This comes at the price that matches discriminating on proofs can only return proofs (so-called **elim restriction**). For instance, we cannot write the following function since the match returns booleans but discriminates on proofs.

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

Coq comes with several automation tactics that can find proofs automatically. For now we mention the automation tactic *tauto*, which will find every proof that involves only the following rules:

1. Introduction and elimination rules for $\top$, $\bot$, $\neg$, $\wedge$, $\vee$, $\rightarrow$, $\leftrightarrow$.
2. Introduction rule for $=$.
3. Conversion rule.

Here is an example for the use of *tauto*.

```
Lemma XP_DN' : XM <-> DN.
Proof. split ; intros A X. generalize (A X). tauto. apply A. tauto. Qed.
```

Note the use of the tactic *generalize*, which given a proof $t : u$ reduces a claim s to the claim $u \rightarrow s$. The tactic *generalize* is useful for introducing a consequence of the assumptions.

Exercise 4.11.1 Here are some challenging lemmas. Try to prove the lemmas with *tauto* and *generalize*. Once you have found a proof with *tauto*, you may fill in by hand the parts found by *tauto*.

```
Lemma 41111 : DN <-> forall X Y: Prop, (~Y -> ~X) -> X -> Y.
Lemma 41112 : XM <-> forall X Y: Prop, (X -> Y) -> (~X -> Y) -> Y.
Lemma 41113 : DN <-> forall X Y: Prop, ~(~X /\ ~Y) -> X \/ Y.
Lemma 41114 : XM <-> forall X Y: Prop, (X -> Y) -> ~X \/ Y.
Lemma 41115 : XM <-> forall X Y: Prop, X \/ Y -> X \/ ~X /\ Y.
Lemma 41116 : DN <-> forall X Y: Prop, ((X -> Y) ->X) -> X.
```

5 Dependent Matches and Induction

In Chapter 2 we did many proofs by case analysis and induction. When you look at the synthesized proof terms, you will see that they employ matches taking a different form from the matches we have explained so far. In fact, the matches we have seen so far are special cases of a more general form of matches providing for dependent type checking. We will now see that dependent matches are essential for proving and computing with inductive types.

5.1 Dependent Matches

The matches we have explained so far (Section 3.9) are a special case of a more general form of matches. The special case requires that all rules of a match

match s return t with $R_1 \cdots R_n$ end

return a value of a fixed return type t. A **dependent match**

match s as z return t with $R_1 \cdots R_n$ end

for an inductive type v is more general in that it works with a return type function $\lambda z : v.t$. The typing of a dependent match is arranged as follows.

1. The dependent match has the type $(\lambda z : v.t)s$.
2. For every rule $p \Rightarrow u$ of the match the typing $u : (\lambda z : v.t)\,p$ must be derivable.

The matches explained so far are the special case where the argument variable z does not occur in t. In this case we have $(\lambda z : v.t)s \approx t$ and $(\lambda z : v.t)\,p \approx t$. For dependent matches at *bool* and *nat* we obtain the following typing rules.

$$\frac{s : bool \qquad z : bool \Rightarrow t : \mathsf{T}_n \qquad u : t^z_{true} \qquad v : t^z_{false}}{\text{match } s \text{ as } z \text{ return } t \text{ with } true \Rightarrow u \mid false \Rightarrow v \text{ end} : t^z_s}$$

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

Dependent matches where z occurs in t are needed to obtain the proof terms justifying the case analyses at *bool* and *nat* we did with *destruct* in Chapter 2.

```
Lemma negb_negb (x : bool) :
negb (negb x) = x.
Proof. destruct x ; reflexivity. Qed.
Print negb_negb.
% fun x : bool ⇒
% match x as z return negb(negb z) = z with
% | true ⇒ eq_refl true
% | false ⇒ eq_refl false
% end
```

```
Lemma nat_match_id (x : nat) :
match x with O => O | S x' => S x' end = x.
Proof. destruct x ; reflexivity. Qed.
Print nat_match_id.
% fun x : nat ⇒
% match x as z return match z with O ⇒ O | S x' ⇒ S x' end = z with
% | O ⇒ eq_refl O
% | S x' ⇒ eq_refl (S x')
% end
```

The typing rule for dependent matches at pair types looks as follows.

$$\frac{s : pair\ s_1\ s_2 \qquad z : pair\ s_1\ s_2 \Rightarrow t : \mathsf{T}_n \qquad x : s_1,\ y : s_2 \Rightarrow u : t^z_{pair\ s_1\ s_2}}{\text{match } s \text{ as } z \text{ return } t \text{ with } pair\ x\ y \Rightarrow u \text{ end} : t^z_s}$$

The proof of the eta law for pairs requires a dependent match at a pair type.

```
Lemma pair_eta (X Y : Type) (p : prod X Y) :
pair (fst p) (snd p) = p.
Proof. destruct p. simpl. reflexivity. Qed.
Print pair_eta.
% fun (X Y : Type) (p : prod X Y) ⇒
% match p as z return pair (fst z) (snd z) = z with
% pair x y ⇒ eq_refl (pair x y)
% end
```

The typing rules for dependent matches can be represented as types, and the representing types can be derived by Coq's type inference. The following commands yield the types representing the typing rules for dependent matches discriminating on booleans, natural numbers, and pairs.

```
Check fun f u v (s : bool) =>
match s as z return f z with true => u | false => v end.
% ∀f : bool → Type, f true → f false → ∀s : bool, f s
```

```
Check fun f u v (s : nat) =>
match s as z return f z with O => u | S x => v x end.
% ∀f : nat → Type, f 0 → (∀x : nat, f (Sx)) → ∀s : nat, f s
```

```
Check fun X Y f u (s : prod X Y) =>
match s as z return f z with pair x y => u x y end.
% ∀ (X Y : Type) (f : prod X Y → Type),
%       (∀ (x : X) (y : Y), f (pair x y)) → ∀ s : prod X Y, f s
```

Exercise 5.1.1 Give the typing rules for dependent matches discriminating on options and list. Use the *Check* command to find out whether your rules are correct.

5.2 Boolean Case Analysis

We represent dependent matches at *bool* with a single function.

```
Definition bool_E :=
fun f u v s =>
match s as x return f x with
| true => u
| false => v
end.

Check bool_E.
% forall f : bool → Type, f true → f false → forall x : bool, f x

Compute bool_E (fun _ => nat) 1 0.
% = fun x : bool ⇒ if x then 1 else 0
% : bool → nat
```

To justify boolean case analysis in proofs, the following lemma suffices.

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

Proof. exact (bool_E p). Qed.
```

Note that the term *bool_E p* type checks since $p : bool \to Type$ can be obtained from $p : bool \to Prop$ by cumulativity (i.e., an application of the typing rule Cum).

```
Lemma negb_negb' (x : bool) :
negb (negb x) = x.
Proof. pattern x. apply bool_Ep ; reflexivity. Qed.

Print negb_negb'.
% negb_negb' = fun x : bool ⇒
% bool_Ep (fun b : bool ⇒ negb (negb b) = b) (eq_refl true) (eq_refl false) x
```

Form the definition of *bool_Ep* it is clear that *bool_Ep* is a type restricted version of *bool_E*. An application *bool_Ep s* can always be replaced with the application *bool_E s*.

Exercise 5.2.1 Prove the following lemmas.

```
Lemma ex5231 (x : bool) :  negb x <> x.
Lemma ex5232 (x : bool) :  x=true \/ x=false.
Lemma ex5233 (x y : bool) :  (if x then negb y else y) = (if y then negb x else x).
```

Exercise 5.2.2 Prove the following lemmas.

```
Definition boolp (x : bool) : Prop := if x then True else False.
Lemma boolp_injective (x y : bool) :  boolp x = boolp y -> x = y.
Lemma ex5241 (x : bool) :  boolp x = True -> x = true.
```

Exercise 5.2.3 Find ground terms that have the following types.

a) $\forall f : bool \to Type.\ f\ true \to f\ false \to \forall x : bool.\ f\ x$

b) $\forall p : bool \to Prop.\ p\ true \to p\ false \to \forall x : bool.\ p\ x$

c) $\forall p : bool \to Prop\ \forall x : bool.\ (x = true \to p\ true) \to (x = false \to p\ false) \to p\ x$

Exercise 5.2.4 Prove the following lemma. Do not use other lemmas.

```
Lemma Boolean_Cantor (X : Type) :
~ exists  f : X -> X -> bool, forall g : X -> bool, exists x : X, f x = g.
```

5.2.1 Bool and False Are Not Equal

We call a type t **inhabited** if there is a term s such that the typing $s : t$ is derivable. We can define inhabitation within Coq.

```
Definition inhabited (X : Type) : Prop := exists x : X, True.
```

We now prove that the types *bool* and *False* are not equal. The trick is to find a **discriminating predicate** $p : Type \to Prop$ such that one can prove both $p\ bool$ and $\neg\ p\ False$. Clearly, *inhabited* is such a predicate.

```
Lemma bool_neq_False :
bool <> False.
Proof. intros e. cut (inhabited bool).
rewrite e. intros [f _]. exact f. exists true. exact I. Qed.
```

Note the use of the tactic *cut*. Given a claim u, the command *cut s* reduces the claim to $s \to u$ and also introduces a new goal with the claim s to justify the reduction.

Exercise 5.2.5 Prove $\neg\ inhabited\ False$.

Exercise 5.2.6 Prove $bool \neq nat$. Hint: Use as discriminating predicate a predicate saying that given three members of a type at least two of them must be equal.

Exercise 5.2.7 Find a short proof of $bool \neq False$ that uses neither *inhabited* nor *cut*. First observe that $bool \to False$ is a proposition and that $bool \to False$ is provably equivalent to $\neg$ *inhabited bool*. Then replace the use of *cut* with the command *generalize true*.

5.2.2 Kaminski's Equation

Kaminski's equation takes the form $f(f(f\,x)) = f\,x$ and holds for every function $f : bool \to bool$ and every $x : bool$. The proof proceeds by repeated boolean case analysis: First on x and then on either *f true* or *f false*. If we do the proof with Coq and *destruct*, we face the problem that *destruct* (*f true*) does not provide the equations *f true* = *true* and *f true* = *false* coming with the case analysis. Fortunately, there is variant of *destruct* called *case_eq* providing the equations.

```
Lemma Kaminski (f : bool -> bool) (x : bool) :
f (f (f x)) = f x.
Proof. destruct x ; case_eq (f true) ; case_eq (f false) ; intros a b;
repeat (exact a || exact b || rewrite a || rewrite b). Qed.
```

The proof script uses the **tactical** *repeat* to solve the eight subgoals introduced by the cascaded boolean case analysis. Replace the semicolon before *repeat* with a period and solve the eight subgoals by hand to understand.

Tacticals compose tactics into more powerful tactics. Here are the specifications of four helpful tacticals.

- $s\,;t$ Applies tactic s, then applies tactic t to every subgoal created by s.
- $s\,||\,t$ Applies tactic s. If application of s fails, tactic t is applied.
- *repeat t* Applies tactic t until it either fails or leaves the goal unchanged.
- *try t* Applies tactic t. If t fails, *try t* leaves the goal unchanged and succeeds.

The use of *repeat* in the proof script for *Kaminski* may be replaced by the tactic *congruence*, which tries to solve a goal by applying unquantified equations appearing as assumptions. This makes the proof of *Kaminski* a one-liner.

```
Lemma Kaminski' (f : bool -> bool) (x : bool) : f (f (f x)) = f x.
Proof. destruct x ; case_eq (f true) ; case_eq (f false) ; congruence. Qed.
```

Exercise 5.2.8 Prove the following lemma justifying the tactic *case_eq* for boolean case analyses. Then prove Kaminski's equation with *destruct* and the lemma *case_eq_bool*.

```
Lemma case_eq_bool (p : bool -> Prop) (x : bool) :
(x = true -> p true) -> (x = false -> p false) -> p x.
```

Exercise 5.2.9 Give two proofs of the following variant of Kaminski's equation.

```
Lemma Kaminski2 (f g : bool −> bool) (x : bool) :
f (f (f (g x))) = f (g (g (g x))).
```

a) Do the proof with boolean case analysis. You get 32 cases.

b) Do the proof without case analysis by applying *Kaminski*.

5.3 Natural Induction

The **elimination function for natural numbers** combines a recursion at *nat* with a dependent match at *nat*:

```
Definition nat_E :=
fun p u v => fix f x :=
match x as y return p y with
| O => u
| S x' => v x' (f x')
end.
```

```
Check nat_E.
% forall f : nat → Type, f O → (forall x : nat, p x → p(S x)) → forall x : nat, f x
```

The elimination function *nat_E* can replace many recursive abstractions at *nat*. If we look at the type of *nat_E*, we discover that *nat_E* justifies induction on natural numbers. In fact, the following specialization of *nat_E* suffices.

```
Definition nat_Ep (p : nat −> Prop) :
p O −> (forall x, p x −> p (S x)) −> forall x, p x
:= nat_E p.
```

Here is an example.

```
Lemma plus_O (x : nat) :
x + O = x.
Proof. revert x. apply nat_Ep. reflexivity.
intros x IHx. simpl. f_equal. exact IHx. Qed.
```

All uses of the tactic *induction* at *nat* can be justified by the proof function *nat_Ep*. In fact, an application of the tactic *induction* at *nat* inserts a proof term that applies a predefined function *nat_ind* that has the same type as our function *nat_Ep*. The predefined *nat_ind* is defined with a predefined function *nat_rect* that is equivalent to our elimination function *nat_E*.

There are a few cases where the induction scheme *nat_Ep* underlying the tactic *induction* is not general enough. There is a tactic *fix* providing for the construction of recursive abstractions. Here is an example.

```
Fixpoint evenb (n : nat) : bool := match n with
| 0 => true
```

```
| 1 => false
| S (S n) => evenb n
end.

Lemma evenb_negb (n : nat) :
evenb n = negb (evenb (S n)).

Proof. revert n. fix f 1.
destruct n as [|n'].  reflexivity.
destruct n' as [|n''].  reflexivity.
exact (f n'').  Qed.
```

A command *fix f n* introduces a recursive abstraction with the local name f and n arguments. The recursion must structurally decrease the nth argument.

Exercise 5.3.1 Give three inductive proofs of $\forall x : nat.\ Sx \neq x$:

a) With the tactic *induction*.

b) With the function *nat_Ep*.

c) With the tactic *fix*.

Exercise 5.3.2 Prove the following lemma.

```
Lemma plus_injective x y z :
x + y = x + z -> y = z.
```

Exercise 5.3.3 Prove the following lemma.

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

Exercise 5.3.4 Prove $\forall n.\ evenb\ n = negb\,(evenb\,(S\ n))$ with the tactic *induction*. You will need a lemma. The proof is difficult since the recursion of *evenb* takes off two applications of the constructor S while *induction* takes off only one application of S.

Exercise 5.3.5 Prove the following lemma.

```
Inductive crazy : Type := Crazy : crazy -> crazy.
Lemma crazy_False :  crazy -> False.
```

Recall that $crazy \to False$ is Coq's abbreviation for $\forall x \in crazy.\ False$.

5.4 Primitive Recursion

We define a recursion scheme known as **primitive recursion**.

```
Definition nat_pr {X : Type} : X -> (nat -> X -> X) -> nat -> X :=
fun u v =>
fix f x := match x with
| O => u
| S x' => v x' (f x')
end.
```

With primitive recursion we define an addition function.

```
Definition add := nat_pr (fun y => y) (fun _ r y => S (r y)).

Compute add 3 4.
% 7
```

The function *add* may be hard to understand at first. Our confidence in *add* increases once we look at its normal form.[1]

```
Compute add.
% fix f (x : nat) : nat -> nat :=
%  match x with
%  | 0 => fun y : nat => y
%  | S x' => fun y : nat => S (f x' y)
%  end
```

Primitive recursion obtains a recursive function from two terms u and v. We can specify the function obtained by primitive recursion declaratively.

```
Definition spec {X : Type} (u : X) (v : nat -> X -> X) (f : nat -> X) : Prop :=
forall n,
f O = u /\
f (S n) = v n (f n).

Lemma pr_correct (X : Type) (u : X) (v : nat -> X -> X) :
spec u v (nat_pr u v).
Proof. split ; reflexivity. Qed.
```

The specification of primitive recursion is canonical in that all functions satisfying the specification are equivalent.

```
Lemma spec_canonical (X : Type) (u : X) (v : nat -> X -> X) :
forall f g, spec u v f -> spec u v g -> forall n, f n = g n.
Proof. intros f g F G n. induction n.
destruct (G 0) as [A _]. rewrite A. apply (F 0).
destruct (G n) as [_ A]. rewrite A. rewrite <- IHn. apply (F n). Qed.
```

[1] The output you see can actually be obtained with the command *Eval cbv in add*. Computing the normal form with *Eval* is less efficient but preserves the names of the local variables.

To prove that the function *add* defined above is equivalent with the predefined addition function *plus*, it now suffices to show that *plus* satisfies the specification of *add*.

```
Lemma add_agrees :
forall n, add n = plus n.
Proof. apply (spec_canonical (fun y => y) (fun _ r y => S (r y))).
apply pr_correct. split ; reflexivity. Qed.
```

The following predicate specifies a function designed by Ackermann in 1928.

```
Definition ackermann (f : nat -> nat -> nat) : Prop :=
forall m n,
f O n = S n /\
f (S m) O = f m 1 /\
f (S m) (S n) = f m (f (S m) n).
```

The specification of Ackermann's function poses two questions:

1. Can we define with primitive recursion a function that satisfies the predicate *ackermann*?
2. Can we prove that there is at most one function that satisfies the predicate *ackermann*?

We will answer both questions positively.

Ackermann argued the existence and uniqueness of his function as follows. Since for any two arguments exactly one of the equations applies, f exists and is unique if application of the equations terminates. The equations terminate since either the first argument is decreased, or the first argument stays the same and the second argument is decreased.

Ackermann's termination argument is outside the scope of Coq's termination checker. Coq will insist that for every fix there is a single argument that is structurally decreased by every recursive application. The problem can be resolved by formulating Ackermann's function with two nested recursions.

```
Definition ack : nat -> nat -> nat :=
fix f m := match m with
| O => S
| S m' => fix g n := match n with
            | O => f m' 1
            | S n' => f m' (g n')
            end
end.
```

Note that *ack* is defined as a recursive function that returns a recursive function. Each of the two recursions is structural on its single argument. The correctness proof for *ack* is straightforward.

```
Lemma ack_correct : ackermann ack.
Proof. repeat split ; reflexivity. Qed.
```

Here is a definition of Ackermann's function using primitive recursion.

```
Definition ack' := nat_pr S (fun _ g => nat_pr (g 1) (fun _ => g)).
```

The correctness proof is again straightforward.

```
Lemma ack'_correct : ackermann ack'.
Proof. repeat split ; reflexivity. Qed.
```

We can also show that *ack* and *ack'* are equivalent.

```
Lemma ack_ack' (m n : nat) : ack m n = ack' m n.
Proof. revert n. induction m. reflexivity.
destruct n. reflexivity. apply IHm. Qed.
```

Exercise 5.4.1 Show that the specification of Ackermann's function is canonical.

```
Lemma ackermann_canonical (f g : nat -> nat -> nat) :
ackermann f -> ackermann g -> forall m n, f m n = g m n.
```

Hint: Follow the proof *spec_canonical* and use a nested induction for n.

Exercise 5.4.2 Prove $\forall X : Type.\ @nat\_pr\ X = nat\_E\ (\lambda\_.X)$.

Exercise 5.4.3 Prove the following correctness lemma for a subtraction function defined with primitive recursion.

```
Definition sub : nat -> nat -> nat :=
nat_pr (fun _ => O) (fun x f => nat_pr (S x) (fun y _ => f y)).
Lemma sub_correct (x y : nat) :
sub x y = x - y.
```

Exercise 5.4.4 Write functions that compute products, powers, and factorials with primitive recursion. Prove the correctness of your functions.

Exercise 5.4.5 Express the predecessor function with *prim_rec*. Do not use *match*. Prove the correctness of your function.

5.5 Projections

Consider the following equations:

$$\begin{aligned} K\ c\ O &= c \\ K\ c\ (S\ n)\ x &= K\ c\ n \\ P\ O\ k &= k \\ P\ (S\ n)\ O\ x &= K\ x\ n \\ P\ (S\ n)\ (S\ k)\ x &= P\ n\ k \end{aligned}$$

We are looking for functions K and P that satisfy the equations for all natural numbers c, n, k and x. The puzzle has a unique solution in set theory. There is also a solution in constructive type theory with dependent matches.

We call a function **arithmetic** if it has a type $nat \to \cdots \to nat$ with $n \geq 1$ arrows. An arithmetic function is **constant** if it always returns the same value. Finally, we call an arithmetic function a **projection** if it always returns one of its arguments. We now realize that Kcn is a constant function that takes n arguments and returns c, and that Pnk is a projection that takes n arguments and returns its k-th argument provided $0 \leq k < n$ and the argument numbering starts with 0.

To define K and P in type theory, we first define a recursive function that given a number n yields the type $nat \to \cdots \to nat$ with n arrows.

```
Fixpoint AF (n : nat) : Type :=
match n with
| O => nat
| S n' => nat -> AF n'
end.

Compute AF 4
% nat → nat → nat → nat
```

We are now able to give types for K and P such that the above equations are well-typed.

$$K : nat \to \forall n : nat.\ AF\ n$$
$$P : \forall n : nat.\ nat \to AF\ n$$

Note that type checking terms like $K\,c\,O$ crucially relies on the conversion rule for unwinding the definition of AF. We are now ready for the definition of K.

```
Fixpoint K (c n : nat) : AF n :=
match n as z return AF z with
| O => c
| S n' => fun _ => K c n'
end.

Compute K 9 3.
% fun _ _ _ : nat ⇒ 9
```

Note that the definition of K really requires a dependent match. Moreover, the return type of K and the dependent match can only be expressed with the recursive function AF.

Exercise 5.5.1 Prove the K satisfies the defining equations given above.

Exercise 5.5.2 (Projections) Define a function $P : \forall n{:}nat.\ nat \to AF\ n$ satisfying the defining equations given above. Prove that your function satisfies the defining equations. Also check that the term $P\ 4\ 2$ reduces to $fun\ \_\ \_\ x\ \_ : nat \Rightarrow x$.

Exercise 5.5.3 Write a function K' that obtains constant functions with *nat_E* and show that K' agrees with K.

```
Definition K' (c : nat) := nat_E ...
Lemma K_agrees c n : K c n = K' c n.
```

Explain why K' cannot be obtained with primitive recursion.

5.6 Surjections and Countability

A **surjection** is a surjective function. We say that a type X is **countable** if there exists a surjectiion from *nat* to X.[2]

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

Definition sur (X Y : Type) : Prop :=
exists f : X -> Y, surjective f.

Definition countable (X : Type) : Prop :=
sur nat X.
```

We show that *bool* is countable and that $nat \to bool$ is uncountable.

```
Lemma countable_bool :
countable bool.
Proof. exists (fun n => match n with O => false | _ => true end).
intros [|]. exists 1. reflexivity. exists 0. reflexivity. Qed.

Lemma uncountable_nat_bool :
~ countable (nat -> bool).
Proof. intros [f A]. pose (g n := negb (f n n)).
destruct (A g) as [n B]. absurd (f n n = g n).
unfold g. destruct (f n n) ; discriminate.
rewrite B. reflexivity. Qed.
```

Note the use of the tactic *absurd* in the uncountability proof. A command *absurd s* applies to any goal and replaces it with goals for $\neg s$ and s.

The notion of countability was first investigated by Cantor. Cantor's Theorem (Section 4.9.2) says that, given a type X, there is no surjection from X to $X \to Prop$. Hence it follows from Cantor's Theorem that $nat \to Prop$ is not countable.

[2] Note that according to our definition uninhabited types are not countable. We deviate from the standard notion of countability for simplicity.

One can prove a variant of Cantor's Theorem saying that there is no surjection from X to $X \to \mathit{bool}$ (Exercise 5.2.4). The proofs of both results exploit there is a negation function (on either *Prop* or *bool*). Here is generalization of Cantor's Theorem giving us both results.

```
Definition strong (X : Type) : Prop :=
exists f : X -> X, forall x, f x <> x.

Lemma Cantor_generalized (X Y : Type) :
strong Y -> ~sur X (X -> Y).
Proof. intros [neg N] [f A]. pose (g x := neg (f x x)).
destruct (A g) as [x B]. absurd (g x = f x x).
unfold g. exact (N _). rewrite B. reflexivity. Qed.
```

Exercise 5.6.1 Prove that uninhabited types are not countable.

```
Lemma uncountable_uninhabited (X : Type) : ~ inhabited X -> ~ countable X.
```

Exercise 5.6.2 Prove the following lemma justifying the tactic *absurd*.

```
Lemma absurd (X Y : Prop) : ~ X -> X -> Y.
```

Exercise 5.6.3 Prove that $\mathit{nat} \to \mathit{nat}$ is uncountable. First do a direct proof in the style of *uncountable_nat_bool*. Then prove the claim with *Cantor_generalized*.

Exercise 5.6.4 Prove that $\mathit{bool} * \mathit{nat}$ is countable. Doing the proof in Coq takes some effort since there is little support for arithmetic. Use the following definitions and lemma.

```
Fixpoint div2 (x : nat) : bool * nat :=
match x with 0 => (true, 0) | 1 => (false, 0)
| S (S x') => let (b, d) := div2 x' in (b, S d) end.
Fixpoint double (x : nat) : nat :=
match x with O => O | S x' => S (S (double x')) end.
Lemma if_eta (b : bool) (X Y : Type) (f : X -> Y) (x y : X) :
(if b then f x else f y) = f (if b then x else y).
```

Exercise 5.6.5 Cantor also proved that there is no injective function from the power set of a set back to the set. To come up with the right spoiler function for this proof is much harder than for the proof of the surjective Cantor Theorem. Complete the following proof.

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

Lemma Cantor_injective (X : Type) (f : (X -> Prop) -> X) : ~ injective f.

Proof. intros A. pose (g x := exists h, f h = x /\ ~ h x). absurd (~ g (f g)).
  ... Qed.
```

```
nat   : Type
O     : nat
S     : nat -> nat
natE  : forall f : nat -> Type, f O -> (forall x, f x -> f (S x)) -> forall x, f x
natEO : forall f u v,   natE f u v O = u
natES : forall f u v x,   natE f u v (S x) = v x (natE f u v x)
```

Figure 5.1: Abstract presentation of the natural numbers

```
nat  : Type
O    : nat
S    : nat -> nat
Pdis : forall x,   S x <> O
Pinj : forall x y, S x = S y -> x = y
Pind : forall p : nat -> Prop, p O -> (forall x, p x -> p (S x)) -> forall x, p x
```

Figure 5.2: Peano axioms

5.7 Abstract Presentation of the Naturals

Figure 5.1 shows an abstract representation of the naturals numbers in Coq's type theory. The abstract presentation assumes the existence of the constructors *nat*, *O*, and *S* and the eliminator *natE*. For the eliminator two quantified equations are assumed. Given the abstract representation, one can for instance define addition and show that it is commutative.

In contrast to the inductive definition of the naturals, the abstract representation of the naturals comes without computation. This makes proofs cumbersome since the lack of computation must be compensated by explicit rewrite steps with *natEO* and *natES*.

At the end of the 19th century the Peano axioms for the natural numbers emerged. Figure 5.2 shows the Peano axioms formulated in Coq's type theory. The Peano axioms assume the existence of the constructors *nat*, *O*, and *S* and the provability of three propositions. The propositions say that the constructors *O* and *S* are disjoint, that the constructor *S* is injective, and that natural induction is an admissible proof technique.

In Coq's type theory, the Peano axioms are weaker than the abstract presentation of the naturals. Starting from the abstract presentation, it is straightforward to give the proofs assumed by the Peano axioms. Given the Peano axioms, there seems no way to obtain the elimination function *natE*.

Given the Peano axioms, one can define total and functional relations mimicking the functions for addition and multiplication. Since in set theory total and functional relations are functions, the Peano axioms are strong enough to define addition and multiplication in set theory.

Exercise 5.7.1 Give an abstract presentation of the booleans.

Exercise 5.7.2 Introduce the abstract representation of the naturals in a section and do the following.

a) Prove $\forall x : nat.\ Sx \neq O$.

b) Prove that S is injective.

c) Prove the induction axiom as postulated by the Peano axioms.

d) Define an addition function *plus*.

e) Prove $\forall y : nat.\ plus\ (S\,O)\ y = y$.

f) Show that *plus* is commutative. Follow the proof in Section 2.4.

5.8 Natural Order

We represent the ordering of the natural numbers as a boolean function.

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

We use Coq's *Notation* command to define the usual notations for comparisons.

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

Note that $7 > 5$ is a boolean term that evaluates to *true*.

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

We define the canonical embedding $bool \to Prop$ and tell Coq with the *Coercion* command to insert an application of the embedding function whenever we write a term of type *bool* in a context that expects a proposition.

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

```
Compute ~ 7 > 5.
% True → False
```

Functions that are automatically inserted due to a *Coercion* command are called **coercions**. In regular mode the applications of coercions are not displayed, but you can tell the Coq interpreter to display them.

We define a new tactic with Coq's *Ltac* command.

```
Ltac stauto := (simpl ; tauto).
```

The command defines a tactic *stauto* that first applies the tactic *simpl* followed by the tactic *tauto*. This provides for short proofs.

```
Lemma le_refl x :
x <= x.
Proof. induction x ; stauto. Qed.
```

The next two proofs are tricky. Try to do them yourself to understand.

```
Lemma le_trans {x} y {z} :
x<=y -> y<=z -> x<=z.
Proof. revert y z. induction x. stauto.
destruct y. stauto. destruct z. stauto.
intros A B. apply (IHx y z) ; stauto. Qed.

Lemma lt_trans' {x} y {z} :
x < y -> y < z -> S x < z.
Proof. intros A B. apply (le_trans (S y)) ; assumption. Qed.
```

The tactic *assumption* used above tries to prove the current goal by applying an assumption and fails if this is not possible.

The following lemmas show that "<" is a strict linear order.

```
Lemma le_strict x :
~ x < x.
Proof. induction x ; stauto. Qed.

Lemma le_or {x y} :
x<=y -> x<y \/ x=y.
Proof. revert y. induction x ; destruct y ; try stauto.
intros A. destruct (IHx y A) as [B|B]. stauto.
rewrite B. stauto. Qed.

Lemma le_lin x y :
x<y \/ y<=x.
Proof. revert y. induction x ; destruct y ; try stauto. exact (IHx y). Qed.

Lemma le_neg x y :
negb (x <= y) = (x > y).
Proof. revert y. induction x ; destruct y ; try stauto. exact (IHx y). Qed.
```

Note the use of the tactical *try*. A command *try s* applies the script *s* and succeeds even if *s* fails.

Exercise 5.8.1 Prove the following lemmas.

```
Lemma bool2Prop_not (b : bool) : ~b <-> negb b.
Lemma bool2Prop_and (a b : bool) : a /\ b <-> andb a b.
Lemma bool2Prop_or (a b : bool) : a \/ b <-> orb a b.
Lemma bool2Prop_eq_true (b : bool) : b <-> b = true.
Lemma bool2Prop_eq_false (b : bool) : ~b <-> b = false.
```

Exercise 5.8.2 Prove the following lemmas.

```
Lemma le_O {x} : x <= O -> x = O.
Lemma le_S x : x <= S x.
Lemma le_wk {x y} : x < y -> x <= y.
Lemma le_rwk {x y} : x <= y -> x <= S y.
Lemma le_neg' x y : negb (x < y) = (x >= y).
Lemma le_anti x y : x<=y -> y<=x -> x=y.
Lemma le_equal x y : x=y <-> x<=y /\ y<=x.
```

Exercise 5.8.3 Prove the following variants of *le_trans*.

```
Lemma le_lt_trans {x} y {z} : x <= y -> y < z -> x < z.
Lemma lt_le_trans {x} y {z} : x < y -> y <= z -> x < z.
Lemma lt_trans {x} y {z} : x < y -> y < z -> x < z.
```

Exercise 5.8.4 Prove the following variants of *linearity* and *irreflexivity*.

```
Lemma le_lin' x y : x <= y \/ y < x.
Lemma le_lin_tri x y : x < y \/ x = y \/ y < x.
Lemma le_strict' (x y : nat) : x < y -> x <> y.
Lemma le_strict'' (x y : nat) : x < y -> y < x -> False.
```

Exercise 5.8.5 Prove the following lemmas about comparisons and sums.

```
Lemma plus_le x y z : (x + y <= x + z) = (y <= z).
Lemma plus_le' x y z : (x + z <= y + z) = (x <= y).
Lemma plus_lt x y z : (x + y < x + z) = (y < z).
Lemma plus_lt' x y z : (x + z < y + z) = (x < y).
Lemma plus_le_refl x y : x <= x + y.
Lemma plus_le_refl' x y : x < x + S y.
Lemma le_ex_plus x y : x <= y -> exists z, x + z <= y.
Lemma le_plus_wk x y z : x + y <= z -> x <= z.
```

Hint: Use Lemma *plus_com* from Section 2.4 for the proofs *plus_le'* and *plus_lt'*.

Exercise 5.8.6 Prove the following lemma. It says that a proposition $x \leq y$ can be expressed without recursion by quantifying over predicates.

```
Definition lep (x y : nat) : Prop :=
forall p : nat -> nat -> Prop,
(forall x, p 0 x) ->
(forall x y, p x y -> p (S x) (S y)) ->
p x y.

Lemma le_lep (x y : nat) :
x <= y <-> lep x y.
```

Note that the definition of *lep* is impredicative. Mathematically, one says that the ordering on the natural numbers can be defined as the least relation satisfying two characteristic properties.

5.9 Size Induction

Given a type X and a function $f : X \to nat$, we can prove a claim $\forall x : X. px$ by induction on the "size" fx of x. Size induction makes it possible to prove $\forall x : X. px$ under the assumption that py holds for all y smaller than x.

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

Proof. intros A x. apply A. induction (f x). stauto.
intros y B. apply A. intros z C. apply IHn.
generalize (lt_trans' _ C B). stauto. Qed.
```

The proof is remarkable in that it first reduces the claim to a form that can be shown by induction on *nat*. Note that the *induction* tactic is applied to the term $f\,x$, which is not a variable. This means that a lemma is created that is applied to $f\,x$ after it has been shown by induction. Here is a proof script that states the auxiliary lemma with the tactic *assert*.

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

Proof. intros A x. apply A.
assert (L : forall n y, f y < n -> p y).
induction n. stauto.
intros y B. apply A. intros z C. apply IHn. generalize (lt_trans' _ C B). stauto.
exact (L (f x)). Qed.
```

Exercise 5.9.1 Give two proofs of the following lemma: one that applies the lemma *size_induction*, and one that does not use *size_induction*.

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

5.10 Type Constructors with Proper Arguments

There are many possibilities for specifying the even numbers. For instance, one may define a boolean predicate $evenb : nat \to bool$ as we did in Section 2.11. Another possibility characterizes the even numbers inductively:

1. 0 is an even number.
2. If n is an even number, then $S(S\,n)$ is an even number.
3. There are no other even numbers.

The inductive characterization can be expressed as an inductive definition in Coq.

```
Inductive even : nat -> Prop :=
| evenO : even 0
| evenS : forall n, even n -> even (S (S n)).
```

The member constructors of *even* provide for proofs of the inductive propositions obtained with *even*. Here is an example.

```
Lemma even4 :
even 4.
Proof. constructor. constructor. constructor. Qed.
```

The tactic *constructor* is a convenience that for an inductive claim tries to apply one of the associated member constructors.

```
Print even4.
% evenS (S (S O)) (evenS O evenO)
```

Note that the inductive predicate *even* takes a proper argument. The only type constructor we have seen so far that takes a proper argument is *eq*. Type constructors with proper arguments come with special matches whose return type depends on the proper arguments. Here is a match for *even*.

```
Check
fun (p : nat -> Prop) u v n (s : even n) =>
match s in even z return p z with
| evenO => u
| evenS x y => v x y
end.
% forall p : nat -> Prop,
% p O ->
% (forall x : nat, even x -> p (S (S x))) ->
% forall n : nat, even n -> p n
```

Note that z is a local variable connecting the proper argument of *even* with the return type. Here is the typing rule for matches at *even*.

$$\frac{s : even\ s_1 \qquad z : nat \Rightarrow t : Prop \qquad u : t^z_0 \qquad x : nat,\ y : even\ x \Rightarrow v : t^z_{S(Sx)}}{\text{match } s \text{ in } even\ z \text{ return } t \text{ with } evenO \Rightarrow u \mid evenS\ x\ y \Rightarrow v \text{ end} \ :\ t^z_{s_1}}$$

The most important constraint on the design of the typing rules for matches is the requirement that the reduction rules for matches must be type preserving. For the above rule the argument goes as follows.

1. Suppose $s = evenO$. Then $s : even\ 0$. Since we also have $s : even\ s_1$, we have $0 \approx s_1$. Thus $t^z_0 \approx t^z_{s_1}$.
2. Suppose $s = evenS\ w_1\ w_2$. Then $s : even\ (S\ (S\ w_1))$. Since we also have $s : even\ s_1$, we have $S\ (S\ w_1) \approx s_1$. Thus $t^z_{S\ (S\ w_1)} \approx t^z_{s_1}$.

With a dependent match for *even* we can prove the following lemma.

```
Lemma even_pred n :
even n -> even (pred (pred n)).

Proof. intros [|n' A]. constructor. exact A. Qed.

Print even_pred.
% fun (n : nat) (H : even n) =>
% match H in (even z) return (even (pred (pred z))) with
% | evenO => evenO
% | evenS _ A => A
% end
```

Exercise 5.10.1 Prove the following lemmas.

```
Lemma ex5911 n : even n -> even (4+n).
Lemma ex5912 n : even (S (S n)) -> even n.
```

5.10.1 Inversion

From the definition of *even* it is clear that 1 is not even. However, at first it seems impossible to prove $\neg even\ 1$. The trick is to use a return type function that employs a match.

```
Lemma even_inversion_1 :
~ even 1.

Proof. exact (fun A : even 1 =>
match A in even z
return match z return Prop with 1 => False | _ => True end
with evenO => I | evenS _ _ => I end). Qed.
```

Another, less elegant technique reformulates the claim such that the proper argument is a variable and the return type function yields contradictory equations.

```
Lemma even_inversion_1 n :
even n -> n = 1 -> False.
Proof. intros [|n' A]. discriminate. discriminate. Qed.
```

Note that the match for *even* uses the return type function $\lambda z.\, z{=}1 \to \bot$.

Here is another intuitively clear claim that at first seems difficult to prove.

```
forall n : nat, even (S (S n)) -> even n
```

The techniqes used for $\neg even\ 1$ work here as well.

```
Lemma even_inversion_SS n :
even (S (S n)) -> even n.

Proof. exact (fun (A : even (S (S n))) =>
match A in even z
return match z return Prop with S (S z') => even z' | _ => True end
with evenO => I | evenS _ A' => A' end). Qed.

Lemma even_inversion_SS' n k :
even k -> k = S (S n) -> even n.

Proof. intros [|n' A]. discriminate.
intros B. injection B as e. rewrite <- e. exact A. Qed.
```

There is also an elegant proof using Lemma *even_pred*.

```
Lemma even_inversion_SS'' n :
even (S (S n)) -> even n.

Proof. exact (even_pred _). Qed.
```

One says that the above claims follow by inversion from the inductive definition of *even*. There is a tactic *inversion* that derives the equations needed for these claims automatically and attempts to process them with *discriminate* and *injection*.

```
Lemma even_inversion_1'' :
~ even 1.
Proof. intros A. inversion A. Qed.

Lemma even_inversion_SS'' n :
even (S (S n)) -> even n.
Proof. intros A. inversion A as [|n' A']. exact A'. Qed.
```

Exercise 5.10.2 Prove the following lemmas.

```
Lemma ex59221 : ~ even 3.
Lemma ex5922 n : even (4+n) -> even n.
```

5.10.2 Recursion on Proof Terms

Given that *even* comes with the "recursive" member constructor

```
evenS : forall n : nat, even n -> even (S (S n))
```

we can define functions by structural recursion on the proof terms for *even*. Here is an example.

```
Lemma even_S n :
even n -> even (S n) -> False.

Proof. revert n. fix f 2. intros n [|n' A] B.
inversion B.
inversion B as [|k C]. exact (f n' A C). Qed.
```

Note that the type of the recursive function f is

```
forall n : nat, even n -> even (S n) -> False
```

and that the function recurses on its second argument. This is the first time we use a recursive abstraction with more than one argument that cannot be reduced to a recursive abstraction with a single argument (since the type of the second argument depends on the first argument). This becomes clear if we look at the standard induction scheme for *even*:

```
Check
fun (p : nat -> Prop) u v =>
fix f n (s : even n) :=
match s in even z return p z with
| evenO => u
| evenS x y => v x y (f x y)
end.

forall p : nat -> Prop,
p 0 ->
(forall x : nat, even n -> p x -> p (S (S x))) ->
forall n : nat, even n -> p n
```

Coq defines this induction scheme automatically under the name *even_ind* and uses it to support the tactic *induction*.

```
Lemma even_S' n :
even n -> even (S n) -> False.

Proof. induction 1 as [|n _ IH] ; intros A.
inversion A.
apply IH. inversion A as [|n' A']. exact A'. Qed.
```

The numeric argument 1 of the tactic *induction* says that the induction should be on the first argument type of the claim (*even n* in the example). The numeric argument 1 will always suffice for proof term inductions.

Exercise 5.10.3 Prove the following lemmas by induction on proof terms.

```
Lemma even_sum m n :  even m −> even n −> even (m+n).
Lemma even_sum' m n :  even (m+n) −> even m −> even n.
```

Exercise 5.10.4 Prove that the inductive definition of evenness agrees with the boolean definition.

```
Lemma evenib n :  even n <−> evenb n.
```

Exercise 5.10.5 Here is an impredicative definition of evenness.

```
Definition evenp (n : nat) : Prop :=
forall  p :  nat −> Prop,
p 0 −> (forall n, p n −> p (S (S n))) −> p n.
```

Prove that the impredicative definition of evenness agrees with the inductive definition.

```
Lemma evenpi n :  evenp n <−> even n.
```

5.11 Matches at eq

Recall the inductive definition of equality in Section 4.8.

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

Note that the inductive predicate *eq* has one proper argument. Following the rule for *even*, we obtain the following typing rule for matches at *eq*.

$$\frac{s : eq\ s_1\ s_2\ s_3 \qquad z : s_1 \Rightarrow t : Prop \qquad u : t^z_{s_2}}{\text{match } s \text{ in } eq\ \_\ \_\ z \text{ return } t \text{ with } eq\_refl \Rightarrow u \text{ end } : t^z_{s_3}}$$

The definition of the elimination operator for equality is now straightforward.

```
Definition eq_E {X : Type} {x y : X} {p :  X −> Prop} :
eq X x y −> p y −> p x
:= fun e =>
match e in eq _ _ z return p z −> p x with
| eq_refl => fun A : p x => A
end.
```

The elimination operator can be obtained with a very short proof script.

```
Lemma eq_E' {X : Type} {x y : X} {p : X −> Prop} :
eq X x y −> p y −> p x.
```

```
Proof. intros [] A. exact A. Qed.
```

Check that the proof script constructs exactly the proof term we give above for *eq_E*. Also make sure that you understand why *eq_E* justifies the *rewrite* tactic.

Exercise 5.11.1 Prove each of the following lemmas not using other lemmas. Give scripts and proof terms.

```
Lemma ex5101ref (X : Type) (x : X) : eq X x x.
Lemma ex5101sym (X : Type) (x y : X) : eq X x y -> eq X y x.
Lemma ex5101trans (X : Type) (x y z : X) : eq X x y -> eq X y z -> eq X x z.
Lemma ex5101f_equal (X Y : Type) (f : X -> Y) (x y : X) : eq X x y -> eq Y (f x) (f y).
Lemma ex5101rewrite_R (X : Type) (x y : X) (p : X -> Prop) : eq X x y -> p x -> p y.
```

Exercise 5.11.2 Consider the following inductive definition of equality where the inductive predicate *eq2* takes two proper arguments.

```
Inductive eq2 (X : Type) : X -> X-> Prop :=
| eq2_I : forall x : X, eq2 X x x.
```

a) Given the typing rule for matches at *eq2*.

b) Prove the following elimination lemma for *eq2*. Give a script and a proof term.

```
Lemma eq2_E (X : Type) (x y : X) (p : X -> Prop) : eq2 X x y -> p y -> p x.
```

Exercise 5.11.3 Consider the following inductive definition of equality where the inductive predicate *eq3* takes three proper arguments.

```
Inductive eq3 : forall X : Type, X -> X-> Prop :=
| eq3_I : forall (X : Type) (x : X), eq3 X x x.
```

a) Given the typing rule for matches at *eq3*.

b) Prove the following elimination lemma for *eq3*. Give a script and a proof term.

```
Lemma eq3_E (X : Type) (x y : X) : eq3 X x y -> forall p : X -> Prop, p y -> p x.
```

Exercise 5.11.4 Consider the following inductive definition of an order predicate for the natural numbers.

```
Inductive lei : nat -> nat -> Prop :=
| leiO : forall x : nat, lei O x
| leiS : forall x y, lei x y -> lei (S x) (S y).
```

a) Given the typing rule for matches at *lei*.

b) Prove the following lemmas.

```
Lemma lei_refl x : lei x x.
Lemma lei_trans x y z : lei x y -> lei y z -> lei x z.
Lemma leib x y : leb x y <-> lei x y.
```

5.12 Termination and Divergence

We assume a type X and a binary relation r on X.

```
Section Assume_X_r.
Variable X : Type.
Variable r : X -> X -> Prop.
```

We see the relation r as a directed graph whose nodes are the members of X and whose edges are the pairs (x, y) such that $r\,x\,y$ is provable. We define the set of **terminating nodes** inductively: A node is terminating if all its successors are terminating.

```
Inductive ter (x : X) : Prop :=
| terc : (forall y : X, r x y -> ter y) -> ter x.
```

Note that every node not having a successor is terminating. Properties of terminating nodes can be shown by **well-founded induction**: To show that a terminating node satisfies a property, one can assume that its successors satisfy the property.

```
Lemma wf_induction (p : X -> Prop) :
(forall x, (forall y, r x y -> p y) -> p x) ->
forall x, ter x -> p x.
```

```
Proof. intros R. fix f 2. intros x A. apply R.
intros y B. destruct A as [A']. apply f, A', B. Qed.
```

If we look at the proof term, we see a new form of structural recursion.

```
Print wf_induction.
fun (p : X -> Prop) (R : forall x : X, (forall y : X, r x y -> p y) -> p x) =>
fix f (x : X) (A : ter x) {struct A} : p x :=
R x (fun (y : X) (B : r x y) =>
     match A with terc A' => f y (A' y B) end)
```

The recursive function f recurses on its second argument A, which is of type $p\,x$. The recursion follows the type of the single constructor *terc*.

```
terc : forall x : X, (forall y : X, r x y -> p y) -> p x
```

Structural decomposition of A yields a function A'. The recursive application of f then takes the application $A'\,y\,B$ as second argument. The rule to learn here is as follows: If A is a member of an inductive type X and the function A' is obtained by structural decomposition of A, then every application of A' that yields a result in X yields a result that is structurally smaller then A. In the example above, $A'\,y\,B$ is thus structurally smaller than A and hence the recursive application of f to $A'\,y\,B$ is admissible. We speak of **recursion through a higher-order constructor**.

The induction lemma Coq synthesizes for *ter* looks as follows.

```
Check ter_ind.
forall p : X -> Prop,
(forall x : X, (forall y : X, r x y -> ter y) ->
               (forall y : X, r x y -> p y) ->
                p x) ->
forall x : X, ter x -> p x

Compute ter_ind.
fun (p : X -> Prop) R =>
fix f x (A : ter x) : p x :=
match A with terc A' => R x A' (fun y B => f y (A' y B)) end.
```

First look at the normal form of *ter_ind* to see the synthesized recursion scheme. Types that can be derived automatically are omitted. The type of *ter_ind* is more general than the type of *wf_induction* in that the inductive relaxation comes with the additional assumption that all successors are terminating.

We call a node of a graph **diverging** if there is an infinite path issuing from it.

```
Definition ipath (f : nat -> X) : Prop :=
forall n, r (f n) (f (S n)).

Definition div (x : X) : Prop :=
exists f, ipath f /\ f 0 = x.
```

Every node on an infinite path diverges.

```
Lemma div_ipath (f : nat -> X) :
ipath f -> forall n, div (f n).

Proof. intros A n. exists (fun k => f (k+n)). split.
intros k. exact (A _). reflexivity. Qed.
```

We prove that a node of a graph cannot be both terminating and diverging.

```
Lemma ter_div :
forall x, ter x -> div x -> False.

Proof. induction 1 as [x _ IH]. intros [f [B e]]. destruct e.
apply (IH (f 1)). exact (B _). exact (div_ipath _ B _). Qed.
```

We close the section *Assume_X_p*

```
End Assume_X_r.
Implicit Arguments ter [X].
Implicit Arguments div [X].
```

and show that the relation ">" terminates on every natural number.

```
Lemma gt_terminates :
forall n, ter (fun x y => x > y) n.

Proof. apply (size_induction (fun x => x)). intros x IH.
constructor. exact IH. Qed.
```

Note that the proof uses the lemma *size_induction* from Section 5.9. Next we show that the relation "<" diverges on every natural number.

```
Lemma lt_diverges :
forall x : nat, div (fun x y => x < y) x.

Proof. intros x. exists (fun n => n+x).
split. intros n. apply le_refl. reflexivity. Qed.
```

Given a graph homomorphism, every node mapped to a terminating node of the target graph terminates.

```
Lemma ter_hom {X Y : Type} (r : X -> X -> Prop) (r' : Y -> Y -> Prop)
(f : X -> Y)
(H : forall x y, r x y -> r' (f x) (f y)) :
forall x : X, ter r' (f x) -> ter r x.

Proof. fix F 2. intros x [A]. constructor. intros y B. apply F, A, H, B. Qed.
```

The proof is by recursion on the proof of $ter\ r'\ (f\ x)$. We formulate the recursion directly since the induction lemma *ter_ind* does not apply to the claim.[3]

The **lexical product** of two relations is defined as follows.

```
Definition lex {X Y : Type} (r : X -> X -> Prop) (s : Y -> Y -> Prop)
(p q : X * Y) : Prop :=
let (x,y) := p in let (x',y') := q in r x x' \/ x=x' /\ s y y'.
```

A relation is **terminating** if all its nodes are terminating.

```
Definition terminates {X : Type} (r : X -> X -> Prop) : Prop :=
forall x, ter r x.
```

We show that the lexical product of two terminating relations is terminating.

```
Lemma ter_lex {X Y : Type} (r : X -> X -> Prop) (s : Y -> Y -> Prop) :
terminates r -> terminates s -> terminates (lex r s).

Proof. intros R S [x y]. generalize (S y). revert y. generalize (R x).
induction 1 as [x _ IHR]. induction 1 as [y _ IHS].
constructor. intros [x' y'] [A|[e B]].
apply IHR. exact A. apply (S y').
destruct e. apply IHS. exact B. Qed.
```

Exercise 5.12.1 Prove the following lemmas.

```
Lemma ex51111 {X : Type} (r : X -> X -> Prop) (x : X) :
ter r x -> forall y, r x y -> ter r y.

Lemma ex51112:
~ exists f : nat -> nat, forall n, f n > f (S n).
```

[3] The induction lemma *ter_ind* applies and the tactic *induction* can be used if the claim is reformulated as follows: $\forall y.\ ter\ r'\ y \to \forall x.\ f\ x = y \to ter\ r\ x$.

```
Lemma ex51113 {X : Type} (r : X -> X -> Prop) (f : X -> nat) :
(forall x y, r x y -> f x > f y) -> forall x, ter r x.
```

Hint: Use the lemmas *ter_div*, *gt_terminates*, and *ter_hom*.

Exercise 5.12.2

a) Prove that the composition of two terminating relations is terminating.

b) Define the transitive closure of a relation with an inductive predicate

```
transc : forall X : Type, (X -> X -> Prop) -> X -> X -> Prop
```

c) Prove that the transitive closure of a terminating relation is terminating.

6 Sum and Sigma Types

6.1 Division by 2 as Certifying Function

In programming languages, the type of a function carries little infomation. For instance, the type $nat \to nat \to nat$ admits addition, multiplication and many other functions. We will now see that in constructive type theory one can write function types that fully specify the computational behavior of their member functions. As example we consider a type whose members are functions that divide their argument by 2. We start with a proposition:

```
Definition Div2p : Prop :=
forall n : nat, (exists k, n = 2 * k) \/ (exists k, n = S (2 * k)).
```

A proof of this proposition is a function that for every number n yields a construction from which one can obtain a number k such that $k = \lfloor \frac{n}{2} \rfloor$. Thus one would hope that from a proof of *Div2p* one can obtain a function that divides its argument by 2. However, this hope does not work out since Coq imposes the elim restriction, which makes it impossible to return a number that is part of a proof construction (see Section 4.11).

Types are partitioned in three classes: propositions, **informative types**, and universes. Propositions are also known as **logical types**. Note that the elim restriction applies to inductive logical types. To compensate for the elim restriction, Coq has predefined type constructors *sum* and *sig* that are like disjunction and existential quantification but yield informative types rather than logical types.

```
Inductive sum (X Y : Type) : Type :=
| inl : X -> sum X Y
| inr : Y -> sum X Y.

Notation "s + t" := (sum s t) : type_scope.
Implicit Arguments inl [X].
Implicit Arguments inr [Y].

Inductive sig {X : Type} (p : X -> Prop) : Type :=
| exist : forall x : X, p x -> sig p.

Notation "{ x | s }" := (sig (fun x => s)) : type_scope.
```

Types obtained with the constructor *sum* are called **sum types**, and types obtained with the constructor *sig* are called **sigma types**. We can now write an informative type mimicking the logical type *Div2p*.

```
Definition Div2 : Type :=
forall n : nat, {k | n = 2*k} + {k | n = S (2*k)}.
```

Next we construct a function of type *Div2* using Coq's scripting facility.

```
Definition div2c : Div2.
unfold Div2. induction n.
left. exists 0. reflexivity.
destruct IHn as [[k A]|[k A]].
right. exists k. f_equal. exact A.
left. exists (S k). rewrite A. simpl. rewrite plus_S. reflexivity. Defined.
```

The script is identical with a proof script for the proposition *Div2p*. Step through the script to see what happens. The lemma *plus_S* was proven in Section 2.4. Note that we close the script with the command *Defined* so that we can compute with the function *div2c*.

```
Compute div2c 11.
% inr {k : nat | 11 = 2*k}
%     (exist (fun k => 11 = S (2*k))
%            5
%            ...  )
```

We have omitted the third argument of the constructor *exist* since it is a long proof.[1] We call *div2c* a **certifying** function since it yields results containing proofs. Based on the certifying function *div2c*, we can now write ordinary functions.

```
Definition even (n : nat) : bool :=
if div2c n then true else false.

Compute even 9.
% false
```

Note that the if-then-else notation translates to a match with two rules. It can be used for every inductive informative type that has two member constructors.

```
Definition div2 (n : nat) : nat.
destruct (div2c n) as [[k _]|[k _]]. exact k. exact k. Defined.

Compute div2 9.
% 4
```

[1] You can obtain the canonical proof *eq_refl 11* if you close the proof script for *plus_S* with the command *Defined*.

```
Definition mod2 (n : nat) : nat :=
if div2c n then 0 else 1.

Compute mod2 9.
% 1

Definition divmod2 (n : nat) : nat + nat.
destruct (div2c n) as [[k _]|[k _]]. exact (inl nat k). exact (inr nat k). Defined.

Compute divmod2 9.
% inr nat 4
```

It is now clear that the certifying function *div2c* is useful computationally. As one would expect, it is also useful for proofs.

```
Lemma div2mod2 (n : nat) :
n = 2 * div2 n + mod2 n /\ mod2 n < 2.

Proof. unfold div2, mod2.
destruct (div2c n) as [[k A]|[k A]] ; split ; try stauto.
rewrite plus_O. exact A. rewrite plus_S, plus_O. exact A. Qed.
```

The lemmas *plus_O* and *plus_S* are shown in Section 2.4.

The following function converting booleans to members of proof-carrying sums will be useful in the following.

```
Definition bool2sum (b : bool) : b + ~b :=
if b as z return z + ~z then inl _ I else inr _ (fun f => f).
```

Exercise 6.1.1 Complete the following definitions.

```
Definition ex6111 (X : Prop) : X \/ X -> X + X.
Definition ex6112 (X : Prop) : X \/ False -> X + False.
```

Exercise 6.1.2 Complete the following definition and prove the lemma.

```
Definition forget {X Y} {p : X -> Prop} {q : Y -> Prop} : sig p + sig q -> X + Y.
Lemma forget_div2c (n : nat) : forget (div2c n) = divmod2 n.
```

6.2 Bounded Search

Given a boolean predicate $nat \to bool$ and a number n, we can search for a number $k \le n$ satisfying the predicate. Here is a type whose members are certifying functions that search for such a k.

```
Definition Search : Type :=
forall (p : nat -> bool) (n : nat),
{x | x <= n /\ p x} + (forall x, x <= n -> ~ p x).
```

We assume the boolean definition of the order "≤" from Section 5.8. We now write a recursive function *searchc* : *Search* that starts from the upper bound and searches downwards until it either finds a k satisfying p or knows that such a k does not exist. We use the lemmas *le_refl*, *le_O*, *le_rwk*, and *le_or* from Section 5.8 to construct the necessary proofs.

```
Definition searchc : Search.
intros p. fix f 1. intros n.
destruct (bool2sum (p n)) as [A|A].
left. exists n. split. exact (le_refl n). exact A.
destruct n. right. intros x B. rewrite (le_O B). exact A.
destruct (f n) as [[x [C D]]|B].
left. exists x. split. exact (le_rwk C). exact D.
right. intros x C. destruct (le_or C) as [D|D].
apply B. exact D. rewrite D. exact A. Defined.
```

We can now write an ordinary search function that does not bother about proofs.

```
Definition search (p : nat -> bool) (n : nat) : option nat.
destruct (searchc p n) as [[x _]|_]. exact (Some x). exact None. Defined.

Compute search (fun x => 5 * x >= x * x) 10.
% Some 5
```

Here is a correctness proof for *search*.

```
Lemma search_correct (p : nat -> bool) (n : nat) :
match search p n with
| None => forall k, k <= n -> ~ p k
| Some k => k <= n /\ p k
end.

Proof. unfold search. destruct (searchc p n) as [[k A]|A] ; exact A. Qed.
```

Exercise 6.2.1 The specification *Search* is loose in that it does not say which $k \leq n$ satisfying p is returned.

a) Write a specification *Search_max* that requires that the maximal $k \leq n$ satisfying p is returned. Use the following definition.

```
Definition max (p : nat -> bool) (n x : nat) : Prop :=
p x /\ forall k, k<=n -> p k -> k<=x.
```

b) Construct a function of type *Search_max*. Prove and use the following lemma.

```
Lemma max_S (p : nat -> bool) (n x : nat) :
max p n x -> ~ p (S n) -> max p (S n) x.
```

6.3 Least Number Search

Here is the specification for a certifying function that given a boolean predicate on *nat* and a proof that the predicate is satisfiable computes the least number satisfying the predicate.

```
Definition low (p : nat -> bool) (n : nat) : Prop :=
forall k, k<=n -> p k -> k = n.

Definition min (p : nat -> bool) (n : nat) : Prop :=
p n /\ low p n.

Definition First : Type :=
forall p : nat -> bool, ex p -> sig (min p).
```

Writing such a function requires a couple of new ideas. In a programming language, we would ignore the proofs and write a recursive function that starting from 0 searches for the first number satisfying p. In Coq, we face the problem that increasing a number is not structurally recursive. For that reason we will write a recursive function taking two arguments. The first argument is a *counter* that is incremented by 1 by each recursion step. The second argument is a *permission* that is decreased by each recursion step. So the recursion is on the permission. We represent permissions as proof terms for an inductive predicate *safe*.

```
Section Assume_p.
Variable p : nat -> bool.
Inductive safe (n : nat) : Prop :=
| safeI : p n -> safe n
| safeS : safe (S n) -> safe n.
```

If we have a permision $A : \mathit{safe}\ n$, we know that there is a $k \geq n$ satisfying p and that the size of the permission is an upper bound on the number of steps it takes an upward search starting at n to find a number satisfying p. We will write a function *first'* that given a number n and a permission of type *safe n* returns the least $k \geq n$ satisfying p.

```
Definition first' : forall n : nat, safe n ->  nat :=
fix f n A :=
match bool2sum (p n) return nat with
| inl _ => n
| inr B => f (S n) match A return safe (S n) with
                   | safeI C => match B C with end    (* B : ¬p n, C : p n *)
                   | safeS C => C
                   end
end.
```

The function is given a counter n and a permission A. It first tests whether the counter satisfies p. If this is the case, it returns the counter. Otherwise, the function recurses with the incremented counter and a permission that is obtained with a match on the given permission A. For the recursion to be structural, the permission obtained with the match must be structurally smaller than the permission A. This is the case if each rule of the match yields a permission that is smaller than A. The body of the first rule is a match with no rules and hence yields a permission smaller than A (since each of its rules does). The body of the second rule returns a permission that is obtained from A by stripping off the constructor *safeS*.

An ordinary recursive function first does a match on the argument it recurses on and then recurses in the bodies of some of the rules. The function *first'* modifies this pattern in that it delegates the match to an argument term of the recursive application. We speak of an **eager proof recursion**. With eager proof recursion it is possible to recurse and match on proofs but nevertheless return values that are not proofs. This is impossible with the ordinary recursion pattern since it would violate the elim restriction.

It is now routine to write a function $first : ex\ p \to nat$ that given a proof of $ex\ p$ yields the least number satisfying p.

```
Lemma safe_O : forall n : nat, safe n -> safe O.
Proof. induction n. tauto. intros A. apply IHn, safeS, A. Defined.

Lemma ex_safe : ex p -> safe O.
Proof. intros [n A]. apply (safe_O n), safeI, A. Defined.

Definition first : ex p -> nat := fun E => first' 0 (ex_safe E).
End Assume_p.

Check first.
% forall p : nat -> bool, ex (fun x : nat => bool2Prop (p x)) -> nat

Compute
let p x := 3 + 2 * x > 15
in first p (ex_intro p 100 I).
% 7
```

The design of *first* scales nicely to a certifying function $firstc : First$. A definition of such a function is shown in Figure 6.1. The main idea is to give *firstc'* an extra argument carrying a proof of the invariant *low p n*. The definition of *safe* and the lemmas *safe_O* and *ex_safe* are unchanged. The definitions of *low* and *min* are repeated to make the presentation self-contained. Note the use of the conversion tactic *hnf* (head normal form) in the script for *firstc'*. The tactic *hnf* reduces the top level of a claim until a variable or a functional abstraction or a function type shows up.

```
Definition low (p : nat -> bool) (n : nat) : Prop :=
forall k, k<=n -> p k -> k = n.

Definition min (p : nat -> bool) (n : nat) : Prop :=
p n /\ low p n.

Lemma lowO p : low p O.
Proof. intros [|k] ; stauto. Qed.

Lemma lowS p n : low p n -> ~ p n -> low p (S n).
Proof. unfold low. intros A B k C D. destruct (le_or C) as [E|E].
generalize D. rewrite (A k E D). tauto. exact E. Qed.

Section Assume_p.
Variable p : nat -> bool.
Inductive safe (n : nat) : Prop :=
| safeI : p n -> safe n
| safeS : safe (S n) -> safe n.

Lemma firstc' : forall n, safe n -> low p n -> sig (min p).
Proof. fix f 2. intros n A L.
destruct (bool2sum (p n)) as [B|B].
exists n. hnf. tauto.
apply (f (S n)).
destruct A as [C|C]. destruct (B C). exact C.
apply lowS ; assumption. Defined.

Lemma safe_O : forall n, safe n -> safe O.
Proof. induction n. tauto. intros A. apply IHn, safeS, A. Defined.

Lemma ex_safe : ex p -> safe O.
Proof. intros [n A]. apply (safe_O n), safeI, A. Defined.

Definition firstc : ex p -> sig (min p) :=
fun E => firstc' 0 (ex_safe E) (lowO p).
End Assume_p.

Check firstc.
% forall p : nat -> bool, (exists x : nat, p x) -> sig (min p)
```

Figure 6.1: Certifying least number search

7 Programming with Dependent Types

In this chapter we consider informative types which depend on members of informative types. For example, we will be interested in informative types that depend on members of *nat*. We will study three such types. In Chapter 5 we considered arithmetic functions. The type of arithmetic functions was given by $AF : nat \to \mathsf{T}$. We will generalize AF to so that for any types A and B we obtain the dependent type $A^n \to B$ for each $n : nat$. We will then consider length-indexed lists. The type of length-indexed lists depends on a natural number n corresponding to the length of the list. Finally, we will consider a type $Fin\, n$ which is a type with exactly n elements.

7.1 Cascaded Functions

Recall that earlier we considered arithmetic functions, i.e., functions of type $nat \to \cdots \to nat$. Types of this form were realized in Coq using the following definition of AF.

```
Fixpoint AF (n : nat) : Type :=
match n with
| O => nat
| S n' => nat -> AF n'
end.
```

A natural generalization of this is to allow arbitrary types as domain and codomain. That is, we will be interested in functions of type

$$A \to \cdots \to A \to B$$

We will abbreviate such types as $A^n \to B$. Given a type A and B, we say f is a *cascaded function from A to B* if f has type $A^n \to B$ for some n. In Coq we can realize the types of cascaded functions by a definition *Cascade*.

```
Fixpoint Cascade (A B : Type) (n : nat) : Type :=
match n with
| O => B
| S n => A -> Cascade A B n
end.

Notation "A ^ n --> B" := (Cascade A B n) (at level 20).
```

We have already seen many examples of cascaded functions. The functions *andb*, *orb* and *negb* are all cascaded functions from *bool* to *bool*.

```
Check (andb : bool ^ 2 −−> bool).
Check (orb : bool ^ 2 −−> bool).
Check (negb : bool ^ 1 −−> bool).
```

Just as with arithmetic functions, we can recursively define a function K which constructs constant cascaded functions from A to B.

```
Fixpoint K (A : Type) {B : Type} (b : B) (n : nat) : A ^ n −−> B :=
match n with
| O => b
| S n => fun _ => K A b n
end.
```

The type B can be left implicit since the return value b is of type B. As mathematical notation, we can write $K_b^{A,n}$ for $K\,A\,b\,n$. The behavior of K can be described either with the single equation

$$K_b^{A,n}\,x_0\,\cdots\,x_{n-1} \;=\; b$$

or with the two equations

$$K_b^{A,0} \;=\; b$$
$$K_b^{A,(S\,n)}\,x \;=\; K_b^{A,n}.$$

For two concrete examples, consider the ground term

$$K\;nat\;false\;2$$

of type $nat^2 \to bool$. An easy sequence of reductions converts this term to the canonical term

$$\lambda x_0\,x_1 : nat.\,false$$

as one would expect. Likewise,

$$K\;bool\;false\;2$$

has type $bool^2 \to bool$ and normal form

$$\lambda x_0\,x_1 : bool.\,false.$$

Exercise 7.1.1 Prove the correctness of K.

```
Lemma K_correct (A B : Type) (b : B) :
 K A b O = b /\ forall n x, K A b (S n) x = K A b n.
```

Let A, B, C be types. Given any two functions $f : B \to C$ and $g : A \to B$, it is easy to define the composition $f \circ g$ of type $A \to C$. We can generalize this to compose $f : B \to C$ with a cascaded function g from A to B. Suppose $f : B \to C$ and $g : A^n \to B$. The type of $f \circ g$ should be $A^n \to C$. That is, $f \circ g$ will be a cascaded function from A to C. The behavior of $f \circ g$ can be easily understood:

$$(f \circ g)x_0 \cdots x_{n-1} \;=\; f(g\,x_0 \cdots x_{n-1})$$

This operation can be defined as follows.

```
Fixpoint comp {A B C : Type} (f : B -> C) (n : nat) : (A ^ n --> B) -> A ^ n --> C :=
match n with
| O => f
| S n => fun g => fun x:A => comp f n (g x)
end.
```

The behavior of *comp* is expressed by the following reductions:

$$comp\,f\,O\,b \text{ reduces to } f\,b$$

$$comp\,f\,(S\,n)\,g\,x \text{ reduces to } comp\,f\,n\,(g\,x)$$

Recall that *negb* is the negation function on *bool*. Let g_1 be the ground term $comp\,negb\,1\,negb$. The type of g_1 is $bool^1 \to bool$ (i.e., $bool \to bool$). One can easily compute that the normal form of $g_1\,false$ is *false* and the normal form of $g_1\,true$ is *true*. Hence one can prove $\forall x : bool.g_1 x = x$.

Recall that *andb* is the conjunction on *bool* and has type $bool^2 \to bool$. We use *comp* to compose the negation function with conjunction. Let g_2 be the ground term $comp\,negb\,2\,andb$. This term has type $bool^2 \to bool$ and behaves as the nand function.

Next suppose we are given a binary function $f : B \to B \to C$ and two cascaded functiongs $g, h : A^n \to B$. We can compose f with g and h in a straightforward way:

$$(f \circ^2 (g,h))\,x_0 \cdots x_{n-1} \;=\; f(g\,x_0 \cdots x_{n-1})(h\,x_0 \cdots x_{n-1}).$$

We implement this operation in Coq as *comp2* as follows.

```
Fixpoint comp2 {A B C : Type} (f : B -> B -> C) (n : nat) :
 (A ^ n --> B) -> (A ^ n --> B) -> A ^ n --> C :=
match n with
| O => f
| S n => fun g h => fun x:A => comp2 f n (g x) (h x)
end.
```

In Figure 7.1 we summarize the mathematical notation and the corresponding Coq terms.

Mathematical notation	Coq representation
$A^n \to B$	$A \wedge n$ `-->` B or *Cascaded A B n*
$K_b^{A,n}$	*K A b n*
$f \circ g$	*comp f n g*
$f \circ^2 (g,h)$	*comp2 f n g h*

Figure 7.1: Notation for cascaded functions

Let g_3 be the ground term *comp2 andb* 1 *negb negb* of type $bool^1 \to bool$. One can easily check that $g_3\,true$ normalizes to *true* and that $g_3\,false$ normalizes to *false*.

We can easily define a function $allb : (bool \to bool) \to bool$ such that $allb\,s$ is *true* if and only if $\forall x : bool.\, s\,x$. It suffices to only check the two booleans and conjoin the result. Here is a direct definition in Coq:

```
Definition allb (p : bool -> bool) : bool := andb (p false) (p true).
```

Exercise 7.1.2 Prove *allb* is correct:

```
Lemma allb_correct (p : bool -> bool) : allb p <-> forall a, p a.
```

Suppose g is of type $bool^3 \to bool$. Consider the term $comp\,allb\,2\,g$. Is this well-typed? If one fills in the implicit arguments properly, it is easy to see that the subterm $comp\,allb\,2$ must have type $(bool^2 \to (bool \to bool)) \to bool^2 \to bool$. Since $bool^3 \to bool$ and $bool^2 \to (bool \to bool)$ both have normal form $bool \to bool \to bool \to bool$, $comp\,allb\,2\,g$ is well-typed and has type $bool^2 \to bool$.

Exercise 7.1.3 Prove the following:

```
Lemma allb3 : forall g:bool ^ 3 --> bool, forall x y,
  comp allb 2 g x y <-> forall z, g x y z.
```

Arguing as above, we see that the ground term

$$\lambda g : bool^{S\,t} \to bool.\, comp\,allb\,t\,g$$

is well-typed for any canonical term t of type *nat*. On the other hand, the abstracted term

$$\lambda n : nat, g : bool^{S\,n} \to bool.\, comp\,allb\,n\,g$$

is *not* well-typed. The problem is that the types $bool^{S\,n} \to bool$ is not convertible with $bool^n \to (bool \to bool)$ if n is a variable. We can overcome this problem with the following helper function:

```
Fixpoint unfoldR {A B : Type} (n : nat) : (A ^ (S n) −−> B) −> A ^ n −−> (A −> B) :=
match n with
| O => fun g => g
| S n => fun g x => unfoldR n (g x)
end.
```

The term

$$\lambda n : nat, g : bool^{S\,n} \to bool.comp\ allb\ n\ (unfoldR\ n\ g)$$

is well typed and has type

$$\forall n : nat.(bool^{S\,n} \to bool) \to (bool^{n} \to bool).$$

The function *unfoldR* maps cascaded functions from A to B (with at least one argument) to a cascaded function from A to $A \to B$ without changing the behavior of the underlying cascaded function. Likewise, a cascaded function from A to $A^m \to B$ can be mapped to a cascaded function from A to B. We define a function *foldM* to do this.

```
Fixpoint foldM {A B : Type} (n : nat) :
 forall m, (A ^ n −−> (A ^ m −−> B)) −> A ^ (n + m) −−> B :=
match n as z return forall m, (A ^ z −−> (A ^ m −−> B)) −> A ^ (z + m) −−> B
with
| O => fun m g => g
| S n => fun m g x => foldM n m (g x)
end.
```

7.2 Length-Indexed Lists

We next consider length-indexed lists (or, *ilists*). Ilists carry the length of the list in their types as a proper argument of the type constructor *ilist*.

```
Inductive ilist (A : Type) : nat −> Type :=
| nl : ilist A 0
| cns : forall n, A −> ilist A n −> ilist A (S n).

Implicit Arguments nl [A].
Implicit Arguments cns [A n].

Notation "s :: t" := (cns s t) (at level 60, right associativity).
```

In mathematical notation, we will often use names such as $\vec{a}$ and $\vec{b}$ for elements of type *ilist A n*. In Coq, we will use names such as *al* and *bl*.

Since the type constructor has a proper argument, the typing rule for the corresponding match makes use of an in clause. Using the index and the in clause,

we can define head and tail functions where we need not concern ourselves with the case in which the given list is empty. Note that any ilist of type *ilist A* (*S n*) is necessarily nonempty. When we match against an ilist of such a type, we will know that the bound variable of the in clause will never be *O*. We make a somewhat surprising use of this fact in the following definitions.

```
Definition hd {A : Type} {n : nat} (al : ilist A (S n)) : A :=
match al in (ilist _ n') return match n' with O => True | S n'' => A end with
| nil => I
| cons n' a ar => a
end.

Definition tl {A : Type} {n : nat} (al : ilist A (S n)) : ilist A n :=
match al in (ilist _ n') return match n' with O => True | S n'' => ilist A n'' end with
| nil => I
| cons n' a ar => ar
end.
```

We can also define an *append* function. The reader should fill in the in and return clauses and carefully check why this is well-typed.

```
Fixpoint append {A : Type} {n m:nat} (a:ilist A n) (b:ilist A m) : ilist A (n + m) :=
match a with
| nil => b
| cons n s a => s::(append a b)
end.

Notation "s ++ t" := (append s t) (at level 60, right associativity).
```

We can easily define a function *Ap* that applies a cascaded function to an ilist.

```
Fixpoint Ap {A B : Type} {n : nat} : forall (f : A ^ n -~> B) (al : ilist A n), B :=
match n with
| O => fun b _ => b
| S n => fun f al => Ap (f (hd al)) (tl al)
end.
```

In mathematical notation we will often omit *Ap*. That is, $f : A^n \to B$ and $\vec{a} :$ *ilist A n*, then we write $f\,\vec{a}$ for $Ap\,f\,\vec{a}$.

Induction on the index n allows us to prove many easy results we could not state before we had ilists. For example, the constant constructor *K* clearly satisfies

$$K\,A\,b\,n\,a_0 \cdots a_{n-1} = b$$

We can now state and prove this equation in the following form.

```
Lemma Ap_K {A B : Type} (b : B) (n : nat) : forall (al : ilist A n), Ap (K A b n) al = b.
induction n.
```

```
intros al. reflexivity.
intros al. simpl. apply IHn.
Qed.
```

We can also do induction on ilists. The induction principle on ilists is given by the following proposition:

$$\forall A : \mathsf{T}. \forall P : (\forall n : nat. ilist\ A\ n \to Prop).$$
$$P\ 0\ nl \to (\forall n : nat. \forall x : A. \forall l : ilist\ A\ n. P\ n\ l \to P\ (S\ n)\ (cns\ x\ l))$$
$$\to \forall n : nat. \forall l : ilist\ A\ n. P\ n\ l$$

(We have left the argument A implicit in nl and the arguments A and n implicit in cns.) The underlying match which justifies this induction principle contains a return type that depends on both an in and an as dependency. One can type check the proof of the induction principle in Coq:

```
Check
 (fun (A : Type) (P : forall n : nat, ilist A n -> Prop)
  (base : P 0 nil)
  (step : forall (n : nat) (a : A) (l : ilist A n), P n l -> P (S n) (cns a l)) =>
  fix F (n : nat) (l : ilist A n) : P n l :=
  match l as l' in (ilist _ n') return (P n' l') with
  | nil => base
  | cns n' x l' => step n' x l' (F n' l')
  end).
```

We use this induction principle (via Coq's induction tactic) to prove the following relationship between *Ap* and *append*.

```
Lemma Ap_append {A B : Type} {n m:nat} (f:A ^ n --> (A ^ m --> B)) (al1:ilist A n) (al2:ilist A m) :
Ap (Ap f al1) al2 = Ap (foldM n m f) (al1 ++ al2).
Proof. revert f. induction al1.
intros f. simpl. reflexivity.
intros f. simpl. exact (IHal1 (f a)).
Qed.
```

Exercise 7.2.1 Study the proof of *Ap_append*. Make sure you can state all the subgoals after each tactic.

Exercise 7.2.2 Formulate each of the equations below as a lemma in Coq and then prove the lemma.

a) $g\ a\ \vec{a} = g\ (a :: \vec{a})$ where $g : A^{S\,n} \to B$, $a : A$ and $\vec{a} : ilist\ A\ n$.

b) $(f \circ g)\ \vec{a} = f\ (g\ \vec{a})$ where $f : B \to C$, $g : A^n \to B$ and $\vec{a} : ilist\ A\ n$.

c) $(f \circ^2 (g,h))\ \vec{a} = f\ (g\ \vec{a})\ (h\ \vec{a})$ where $f : B \to C$, $g, h : A^n \to B$ and $\vec{a} : ilist\ A\ n$.

We can also make use of ilists to define equivalence of cascaded functions.

```
Definition Feq {A B:Type} {n : nat} (f g : A ^ n -->  B) : Prop :=
  forall al: ilist  A n, Ap f al = Ap g al.

Notation "f == g" := (Feq f g) (at level 11).
```

In mathematical notation we write $f \equiv g$ for $f == g$.

Exercise 7.2.3 Formulate the following equivalences as lemmas in Coq and prove them using the lemmas from Exercise 7.2.2.

a) $f \circ K_b^{A,n} \equiv K_{fb}^{A,n}$ where $f : B \to C$ and $b : B$.

b) $f \circ^2 (K_b^{A,n}, h) \equiv (fb) \circ h$ where $f : B \to C$, $b : B$ and $h : A^n \to B$.

Exercise 7.2.4 Consider the following boolean definition of implication (predefined in Coq).

```
Definition implb (b1 b2:bool) : bool := if b1 then b2 else true.
```

a) Prove the following lemma.

```
Lemma implb_negb_orb (a b : bool) : implb a b = orb (negb a) b.
```

b) Use the lemma from part (a) to prove the following equivalence of functions $bool^2 \to bool$.

```
Lemma Feq_implb_negb_orb (n : nat) (g h : bool ^ n -->  bool):
 comp2 implb n g h == comp2 orb n (comp negb n g) h.
```

7.3 Finite Types

Suppose we wish to define projection functions $P_k : A^n \to A$ such that

$$P_k\, x_0 \cdots x_{n-1} \;=\; x_k.$$

Clearly this only makes sense when $k < n$. This motivates the definition of a type function $Fin : nat \to \mathsf{T}$ such that $Fin\, n$ corresponds to the set $\{0, \ldots, n-1\}$. If we have such a *Fin*, then the type of P will be

$$\forall A : \mathsf{T}.\ \forall n : nat.\ Fin\, n \to A^n \to A.$$

In fact, we defined such a function (called *fin*) in Chapter 2 using *void*, *option* and *iter*. Here we consider an inductive definition of *Fin*.

```
Inductive Fin : nat -> Type :=
| FinO : forall {n}, Fin (S n)
| FinS : forall {n}, Fin n -> Fin (S n).
```

For each n : *nat*, *FinO* is the copy of 0 in the type $Fin\,(S\,n)$. Assume the elements of $Fin\,n$ are (copies of) $0,\ldots,n-1$. For each k : $Fin\,n$, k is a copy of some $k' \in \{0,\ldots,n-1\}$ and $FinS\,k$ is a copy of $k'+1$ in $\{1,\ldots,n\}$. Given any specific natural numbers k and n, we can easily define the copy F^n_k of k as a member of $Fin\,n$. Here are a few concrete examples listing all the elements of $Fin\,1$, $Fin\,2$ and $Fin\,3$.

```
Definition F01 : Fin 1 := FinO.
Definition F02 : Fin 2 := FinO.
Definition F12 : Fin 2 := FinS FinO.
Definition F03 : Fin 3 := FinO.
Definition F13 : Fin 3 := FinS FinO.
Definition F23 : Fin 3 := FinS (FinS FinO).
```

We can also extract the underlying natural number from a member of $Fin\,n$ by a simple recursive function.

```
Fixpoint FinVal {n : nat} (m : Fin n) : nat :=
match m with
| FinO n' => 0
| FinS n' m' => S (FinVal m')
end.
```

For any k and n with $k < n$, $FinVal\,F^n_k$ has normal form k.

Exercise 7.3.1 Give the normal forms of the following terms: *FinVal F01*, *FinVal F02*, *FinVal F12*, *FinVal F03*, *FinVal F13* and *FinVal F23*.

No member constructor of *Fin* gives an element of type $Fin\,0$, and so this type is empty. In fact, we can easily use a (hypothetical) element of $Fin\,0$ to obtain an element of any type.

```
Definition Fin0E (k : Fin 0) (A:Type) : A :=
match k in Fin n return match n with O => A | S _ => True end with
| FinO _ => I
| FinS _ _ => I
end.
```

Now that we have *Fin*, we can define a function P giving projection functions for each type A, natural number n and natural number $k < n$. The fundamental equation P should satisfy is

$$P\,A\,n\,F^n_k\,x_0\,\cdots\,x_{n-1} \;=\; x_k.$$

We never need consider F^0_k since there is no $k < 0$, and so it is enough to consider the equation when n is of the form $n'+1$ and $k \le n$:

$$P\,A\,n\,F^n_k\,x_0\,\cdots\,x_{n'} \;=\; x_k.$$

There are two cases for k: If k is 0, then we require

$$P\,A\,n\,F_0^n\,x_0\,\cdots\,x_{n'} \;=\; x_0$$

which is the same as requiring

$$P\,A\,n\,F_0^n\,x_0 \;=\; K\,A\,x_0\,n'.$$

If k is of the form $k'+1$, then we require

$$P\,A\,n\,F_{k'+1}^n\,x_0\,\cdots\,x_{n'} \;=\; x_{k'+1}$$

which yields the recursive equation

$$P\,A\,n\,F_{k'+1}^n\,x_0\,\cdots\,x_{n'} \;=\; P\,A\,n'\,F_{k'}^{n'}\,x_1\,\cdots x_{n'}$$

or more simply

$$P\,A\,n\,F_{k'+1}^n\,x_0 \;=\; P\,A\,n'\,F_{k'}^{n'}.$$

We can realize this definition in Coq as follows:

```
Fixpoint P {A : Type} {n : nat} (k : Fin n) : A ^ n --> A :=
match k in Fin n' return A ^ n' --> A with
| FinO n' => fun a => K A a n'
| FinS n' k' => fun _ => P k'
end.
end.
```

Exercise 7.3.2 Give the normal forms of the following terms: *@P bool 3 F03*, *@P bool 3 F13*, *P F03 0 1 2* and *P F23 0 1 2*.

Exercise 7.3.3 Prove the following lemmas.

```
Lemma Pex1 : comp negb 1 (P F01) == negb.

Lemma Pex2 :
comp2 implb 2 (P F02) (P F12) == (fun x y => implb x y)
\/
comp2 implb 2 (P F02) (P F12) == (fun x y => implb y x).

Lemma Pex3 :
comp2 implb 2 (P F12) (P F02) == (fun x y => implb x y)
\/
comp2 implb 2 (P F12) (P F02) == (fun x y => implb y x).
```

Exercise 7.3.4 Define a function *get* of type

$$\forall A : \mathsf{T}.\ \forall n : nat.\ Fin\, n \ \rightarrow\ ilist\, A\, n \ \rightarrow\ A$$

such that $get\, A\, n\, k\, l$ returns the k^{th} element of the length-indexed list l.

We end the chapter with by considering inversion principles for *Fin*. Suppose we wish to write a certifying predecessor function with the type *forall n:nat, Fin (S n) -> {k'| k = FinS k'} + {k = FinO}*.[1] A direct attempt to define this function will fail. Instead we define a more general inversion function.

```
Definition Fin_Inv {n:nat} (k:Fin n) :
match n as z return Fin z -> Type with
| O => fun k => False
| S n' => fun k =>
{k'| k = FinS k'} + {k = FinO}
end k.
Proof. destruct k.
tauto.
left. exists k. reflexivity.
Defined.
```

We can use *Fin_Inv* to define a certifying predecessor function.

```
Definition Fin_Pred {n:nat} (k:Fin (S n)) :
{k'| k = FinS k'} + {k = FinO} :=
(Fin_Inv k).
```

Exercise 7.3.5 Use *Fin_Pred* to define the predecessor function with the following type and prove it correct.

```
Definition Fin_pred {n:nat} (k:Fin (S n)) : option (Fin n).

Lemma Fin_pred_correct {n:nat} :
Fin_pred (@FinO n) = None /\ forall (k:Fin (S n)), Fin_pred (FinS k) = Some k.
```

Exercise 7.3.6 Use *Fin_Inv* and scripts to define a predecessor function with the following type and prove the function correct.

```
Definition predFin {n : nat} (x : Fin n) : option (Fin (pred n)).

Lemma predFin_correct {n:nat} :
predFin (@FinO n) = None /\ forall (k:Fin (S n)), predFin (FinS k) = Some k.
```

Exercise 7.3.7 Use *Fin_inv* to prove the following.

[1] We use {} here because of an issue with Coq's syntax.

```
Lemma Fin1 (k:Fin 1) : k = FinO 0.
```

Exercise 7.3.8 Prove for every n, $Fin\,n$ and $fin\,n$ are isomorphic.

Acknowledgement: Adam Chlipala's in-progress textbook *Certified Programming with Dependent Types* (`http://adam.chlipala.net/cpdt/`) as well as his responses on the Coq club mailing list were used in important ways in this chapter. The reader will find many more examples in Chlipala's textbook.

8 Boolean Logic

In this chapter we consider boolean functions $bool^n \to bool$. The main result will be that every function in $bool^n \to bool$ can be represented as a combination of the constant function K_{false} and the projections P_k via composition with the implication function *implb*. Note that composition with *implb* takes any two functions $g, h : bool^n \to bool$ and creates a new function $(implb \circ^2 (g, h)) : bool^n \to bool$. For simplicity, we define $Impl\, g\, h$ to be $implb \circ^2 (g, h)$.

In order to formulate and prove the main result, we will define a syntax of boolean formulas. The semantics of boolean formulas will be given as a denotation function mapping a boolean formula into the set of boolean functions generated by K_{false} and the projections P_k via *Impl*. Every boolean function will be the denotation of a boolean formula. We refer to this result as *denotational completeness*.

8.1 Syntax and Semantics of Boolean Logic

Let n be a natural number and $x_0, x_1, \ldots, x_{n-1}$ be distinct objects we will call *variables*. The set **B** of boolean formulas is defined by the following grammar

$$s, t ::= x_k \mid \sharp \mid s \Rightarrow t$$

where k ranges over $\{0, ..., n-1\}$. When necessary, we write $\mathbf{B}_n$ to make the dependence on n clear. The formula $\sharp$ corresponds to false and $s \Rightarrow t$ corresponds to implication. In Coq we can use $Fin\, n$ to represent the type of variables and realize the definition of **B** and corresponding notation (within a section) as follows:

```
Section BEn.

Variable n : nat.

Inductive B : Type :=
| B_Var : Fin n -> B
| B_Fal : B
| B_Imp : B -> B -> B.

Local Notation "#" := B_Fal.
```

```
Local Notation "s ==> t" := (B_Imp s t) (at level 90, right associativity).
```

We can easily define other syntactic operations from the basic ones.

```
Definition B_Tru : B := (# ==> #).
Definition B_Not (s : B) : B := (s ==> #).
Definition B_And (s t:B) : B := (B_Not (s ==> B_Not t)).
```

Exercise 8.1.1 Prove the following lemma.

```
Lemma B_NotTru_Fal_ex : (B_Not B_Tru) = B_Fal \/ (B_Not B_Tru) <> B_Fal.
```

Boolean formulas give the syntax of boolean logic. We now describe the semantics. For each formula s, we recursively define a boolean function

$$[\![s]\!] : bool^n \to bool$$

as follows:

$$[\![x_k]\!] := P_k$$

$$[\![\sharp]\!] := K_{false}$$

$$[\![s \Rightarrow t]\!] := Impl\ [\![s]\!]\ [\![t]\!]$$

In Coq we can realize this as follows:

```
Definition Impl (g h:bool ^ n --> bool) : bool ^ n --> bool := comp2 implb n g h.

Fixpoint B_Den (s : B) : bool ^ n --> bool :=
match s with
| B_Var k => P k
| B_Fal => K bool false n
| B_Imp s1 s2 => Impl (B_Den s1) (B_Den s2)
end.

Local Notation "[[ s ]]" := (B_Den s).
```

Exercise 8.1.2 Prove the following lemma.

```
Lemma B_NotTru_Fal_ex2 : [[B_Not B_Tru]] == [[B_Fal]] \/ ~ [[B_Not B_Tru]] == [[B_Fal]].
```

We can now prove our definitions of *B_Not* and *B_And* are interpreted using *negb* and *andb* as expected. We prove these using mathematical notation. In particular, we omit mentions of *Ap*.

Lemma 8.1.3 For every boolean formula s, we have

$$[\![B\_Not\ s]\!] \equiv negb \circ [\![s]\!]$$

Proof Let $\vec{a}$ be given.

$$\begin{aligned}
[\![B_Not\ s]\!]\ \vec{a} &= (Impl\ [\![s]\!]\ [\![\sharp]\!])\vec{a} \\
&= (implb\ \circ^2\ ([\![s]\!], K_{false}))\vec{a} \\
&= implb\ ([\![s]\!]\ \vec{a})\ (K_{false}\ \vec{a}) \\
&= implb\ ([\![s]\!]\ \vec{a})\ false \\
&= \text{if } ([\![s]\!]\ \vec{a}) \text{ then } false \text{ else } true \\
&= negb\ ([\![s]\!]\ \vec{a}) \\
&= (negb\ \circ\ [\![s]\!])\ \vec{a}
\end{aligned}$$

In the proof of Lemma 8.1.3 we have made use of the equation $K_b\,\vec{a} = b$ which was proven as the following Coq lemma in Chapter 7.

```
Lemma Ap_K {A B : Type} (b : B) (n : nat) :
 forall  (al :  ilist  A n), Ap (K A b n) al = b.
```

We have also made use of the equations

$$(f \circ g)\,\vec{a} = f\,(g\,\vec{a})$$

and

$$(f \circ^2 (g,h))\,\vec{a} = f\,(g\,\vec{a})\,(h\,\vec{a})$$

from Exercise 7.2.2. In Coq, these equations can be represented (and proven) as the following lemmas.

```
Lemma Ap_comp {A B C : Type} (f : B -> C) (n : nat) : forall (g : A ^ n --> B),
 forall  al: ilist  A n, Ap (comp f n g) al = f (Ap g al).

Lemma Ap_comp2 {A B C : Type} (f : B -> B -> C) (n : nat) : forall (g h : A ^ n --> B),
 forall  al: ilist  A n, Ap (comp2 f n g h) al = f (Ap g al) (Ap h al).
```

Given these lemmas, we can represent and prove Lemma 8.1.3 in Coq.

```
Lemma B_Neg_negb (s : B) : [[B_Not s]] == (comp negb n [[s]]).
intros a.  simpl. unfold Impl. rewrite Ap_comp2. rewrite Ap_K. rewrite Ap_comp. reflexivity.
Qed.
```

Lemma 8.1.4 For all boolean formulas s and t, we have

$$[\![B\_And\ s\ t]\!]\ \equiv\ andb\ \circ^2\ ([\![s]\!], [\![t]\!])$$

Proof Let $\vec{a}$ be given.

$$\begin{aligned}
[\![B_And\ s\ t]\!]\ \vec{a} &= (Impl\ (Impl\ [\![s]\!]\ (Impl\ [\![t]\!]\ [\![\sharp]\!]))\ [\![\sharp]\!])\vec{a} \\
&= implb\ (implb\ ([\![s]\!]\ \vec{a})\ (implb\ ([\![t]\!]\ \vec{a})\ false))\ false
\end{aligned}$$

By considering the four cases for the boolean values $[\![s]\!]\ \vec{a}$ and $[\![t]\!]\ \vec{a}$ we can prove

$$implb\ (implb\ ([\![s]\!]\ \vec{a})\ (implb\ ([\![t]\!]\ \vec{a})\ false))\ false$$

is equal to

$$andb\ ([\![s]\!]\ \vec{a})\ ([\![t]\!]\ \vec{a})$$

as desired. ∎

Exercise 8.1.5 Represent and prove Lemma 8.1.4 in Coq.

Exercise 8.1.6 Define *B_Or* and prove it is interpreted using *orb*.

```
Definition B_Or (s t : B) : B :=
...

Lemma B_Or_orb (s t : B) : [[B_Or s t]] == (comp2 orb n [[s]] [[t]]).
...
```

Finally, we close the Coq section and declare the dependence on n to be implicit in many cases. We also declare a coercion so that we can treat members of *Finn* as variables in $\mathbf{B}_n$ without explicitly mentioning the *B_Var* constructor.

```
End BEn.

Implicit Arguments B_Var [n].

Local Coercion B_Var : Fin >-> B.

Implicit Arguments B_Fal [n].
Implicit Arguments B_Imp [n].
Implicit Arguments B_Not [n].
Implicit Arguments B_And [n].
```

Exercise 8.1.7 Give the type and normal form of the following terms.

a) *[[F01 ==> #]] true*

b) *[[F01 ==> #]] false*

c) *[[(F02 ==> #) ==> F12]] false false*

d) *[[(F02 ==> #) ==> F12]] false true*

e) *[[(F02 ==> #) ==> F12]] false*

8.2 Decidability Results

We say two formulas $s, t \in \mathbf{B}_n$ are **equivalent** if $[\![s]\!] \equiv [\![t]\!]$. In this section we will prove equivalence of formulas is decidable. We prove this result by first constructing a certifying function *bfeqc* of type

```
forall (n:nat) (g h : bool ^ n --> bool),
{a: ilist bool n|Ap g a <> Ap h a} + {forall a:ilist bool n, Ap g a = Ap h a}
```

that decides whether or not $g \equiv h$ for cascaded boolean functions $g, h : bool^n \to bool$. The function is defined by recursion on n. If $n = 0$, then we simply compare the booleans g and h. For the recursive case, suppose $g, h : bool^{Sn} \to bool$. Calling *bfeqc* with n, $g\ false$ and $h\ false$, we obtain either some $\vec{a}$ such that $g\ false\ \vec{a} \neq h\ false\ \vec{a}$ or $g\ false\ \vec{a} \equiv h\ false\ \vec{a}$. In the first case $(false :: \vec{a})$ witnesses $g \not\equiv h$. Assume $g\ false\ \vec{a} \equiv h\ false\ \vec{a}$. Calling *bfeqc* with n, $g\ true$ and $h\ true$, we obtain either some $\vec{a}$ such that $g\ true\ \vec{a} \neq h\ true\ \vec{a}$ or $g\ true\ \vec{a} \equiv h\ true\ \vec{a}$. In the first case $(true :: \vec{a})$ witnesses $g \not\equiv h$. Otherwise, we know $g\ false \equiv h\ false$ and $g\ true \equiv h\ true$, and so $g \equiv h$.

```
Definition bfeqc (n:nat) (g h : bool ^ n --> bool) :
{a: ilist bool n|Ap g a <> Ap h a} + {forall a:ilist bool n, Ap g a = Ap h a}.
induction n as [|n IHn].
destruct g; destruct h ; try tauto.
left. exists nil. discriminate.
left. exists nil. discriminate.
destruct (IHn (g false) (h false)) as [ [a H]|H].
left. exists (false::a). exact H.
destruct (IHn (g true) (h true)) as [ [a H']|H'].
left. exists (true::a). exact H'.
right. intros a. simpl. destruct (hd a).
apply H'.
apply H.
Defined.
```

Using this result, we can easily construct a certifying function deciding equivalence of formulas.

```
Definition Beqc {n:nat} (s t : B n) :
{a: ilist bool n|Ap [[s]] a <> Ap [[t]] a} + {forall a: ilist bool n, Ap [[s]] a = Ap [[t]] a}.
exact (bfeqc n [[s]] [[t]]).
Defined.
```

A formula $s \in B_n$ is **valid** if $[\![s]\!]a_0 \cdots a_{n-1} = true$ for all $a_0, \ldots, a_{n-1}$. It is not difficult to see that s is valid iff $[\![s]\!] \equiv K^n_{true}$. We use this fact to prove validity is decidable. We first use *bfeqc* to obtain a certifying function checking if $g \equiv K^n_{true}$ for a cascaded function $g : bool^n \to bool$.

```
Definition bfvalc (n:nat) (g : bool ^ n --> bool) :
{a: ilist  bool n|~ Ap g a} + {forall a: ilist  bool n, Ap g a}.
destruct (bfeqc n g (K bool true n)) as [ [a H] | H].
left. exists a. intros H'. apply H. rewrite Ap_K. destruct (Ap g a).
reflexivity.
contradiction H'.
right. intros a. rewrite H, Ap_K. exact I.
Defined.
```

We now easily have a certifying function deciding validity of formulas.

```
Definition Bvalc {n:nat} (s : B n) :
{a: ilist  bool n|~ Ap [[s]] a} + {forall  a: ilist  bool n, Ap [[s]]  a}.
exact (bfvalc n [[s ]]).
Defined.
```

It is also possible to prove these decidability results in the other order. We could define *bfvalc* by recursion on n and then use *bfvalc* to define *bfeqc*.

Exercise 8.2.1 Which of the following formulas in $\mathbf{B}_2$ are equivalent?

a) x_0

b) $\sharp$

c) $\sharp \Rightarrow \sharp$

d) $x_0 \Rightarrow x_0$

e) $x_0 \Rightarrow \sharp$

f) $\sharp \Rightarrow x_1$

g) $(\sharp \Rightarrow x_1) \Rightarrow x_0$

h) $(x_0 \Rightarrow x_1) \Rightarrow x_0$

i) $((x_0 \Rightarrow x_1) \Rightarrow x_0) \Rightarrow x_0$

Exercise 8.2.2 Which of the following formulas in $\mathbf{B}_2$ are valid?

a) x_0

b) $\sharp \Rightarrow x_1$

c) $(x_0 \Rightarrow x_1) \Rightarrow x_0$

d) $((x_0 \Rightarrow x_1) \Rightarrow x_0) \Rightarrow x_0$

e) $((x_0 \Rightarrow \sharp) \Rightarrow \sharp) \Rightarrow x_0$

8.3 Denotational Completeness

In this section we will prove denotational completeness: for every boolean function $g : bool^n \to bool$ there is an $s \in B_n$ such that $[\![s]\!] \equiv g$.

First we define a shifting function $\mathbf{B}_S : \mathbf{B}_n \to \mathbf{B}_{n+1}$ as follows:

$$\mathbf{B}_S(x_k) := x_{k+1}$$

$$\mathbf{B}_S(\sharp) := \sharp$$

$$\mathbf{B}_S(s \Rightarrow t) := \mathbf{B}_S(s) \Rightarrow \mathbf{B}_S(t)$$

In Coq this function is defined as follows.

```
Fixpoint B_S {n : nat} (s : B n) : B (S n) :=
match s with
| B_Var k => FinS k
| B_Fal => #
| B_Imp s1 s2 => B_S s1 ==> B_S s2
end.
```

Note that the variable x_0 never occurs in a formula $\mathbf{B}_S s$. The function $\mathbf{B}_S$ shifts the variables away from x_0. We now prove the boolean function $[\![\mathbf{B}_S s]\!]$ ignores its first argument.

Lemma 8.3.1 For every $s \in \mathbf{B}_n$ and boolean b, $[\![\mathbf{B}_S s]\!] b \equiv [\![s]\!]$.

Proof We prove this by induction on s. We consider three cases.

- Suppose s is x_k for $k < n$. We easily compute for any $\vec{a}$,

$$\begin{aligned} [\![\mathbf{B}_S x_k]\!]\ b\ \vec{a} &= [\![x_{k+1}]\!]\ b\ \vec{a} \\ &= P_{k+1}\ b\ \vec{a} \\ &= P_k\ \vec{a} \\ &= [\![x_k]\!]\ \vec{a} \end{aligned}$$

- Suppose s is $\sharp$. For any $\vec{a}$ we have

$$\begin{aligned} [\![\mathbf{B}_S \sharp]\!]\ b\ \vec{a} &= \mathit{false} \\ &= [\![\sharp]\!]\ \vec{a} \end{aligned}$$

- Suppose s is $s_1 \Rightarrow s_2$. By induction hypothesis we know $[\![\mathbf{B}_S s_1]\!] b \equiv [\![s_1]\!]$ and $[\![\mathbf{B}_S s_2]\!] b \equiv [\![s_2]\!]$. For any $\vec{a}$ we have

$$\begin{aligned} [\![\mathbf{B}_S(s_1 \Rightarrow s_2)]\!]\ b\ \vec{a} &= [\![(\mathbf{B}_S s_1) \Rightarrow (\mathbf{B}_S s_2)]\!]\ b\ \vec{a} \\ &= \mathit{Impl}\ ([\![\mathbf{B}_S s_1]\!]\ b\ \vec{a})\ ([\![\mathbf{B}_S s_2]\!]\ b\ \vec{a}) \\ &= \mathit{Impl}\ ([\![s_1]\!]\ \vec{a})\ ([\![s_2]\!]\ \vec{a}) \\ &= [\![s_1 \Rightarrow s_2]\!]\ \vec{a} \end{aligned}$$

In Coq Lemma 8.3.1 can be written and proven as follows.

```
Lemma B_Den_S {n : nat} (b : bool) : forall (s : B n), ([[B_S s]] b) == [[s]].
induction s as [k| |s1 IH1 s2 IH2].
intros a. reflexivity.
intros a. reflexivity.
intros a. simpl. unfold Impl. simpl. repeat rewrite Ap_comp2.
rewrite IH1. rewrite IH2. reflexivity.
Qed.
```

Theorem 8.3.2 (Denotational Completeness) For every n and every $g : bool^n \to bool$ there is some $s \in \mathbf{B}_n$ such that $[\![s]\!] \equiv g$.

Proof We prove this by induction on n. If $n = 0$, then g is *true* or *false* and we can take s to be $\sharp \to \sharp$ or $\sharp$, respectively. Assume we know the result for n. We will prove the result for $n+1$. Let $g : bool^{n+1} \to bool$ be given. Note that $g\ false$ and $g\ true$ are both in $bool^n \to bool$. By the inductive hypothesis there are formulas $s_0 \in \mathbf{B}_n$ and $s_1 \in \mathbf{B}_n$ such that $[\![s_0]\!] \equiv g\ false$ and $[\![s_1]\!] \equiv g\ true$. Note that $(\mathbf{B}_S s_0)$ and $(\mathbf{B}_S s_1)$ are in $\mathbf{B}_{n+1}$. Let t_0 be the formula $(x_0 \Rightarrow \sharp) \Rightarrow (\mathbf{B}_S s_0)$ in $\mathbf{B}_{n+1}$ and t_1 be the formula $x_0 \Rightarrow (\mathbf{B}_S s_1)$ in $\mathbf{B}_{n+1}$. Let s be the formula in $\mathbf{B}_{n+1}$ given by $(B\_And\ t_0\ t_1)$. We prove $[\![s]\!] \equiv g$.

Let $a_0, \ldots, a_n$ be booleans and write $\vec{a}$ for the corresponding indexed list. We must prove $[\![s]\!]\vec{a}$ equals $g\vec{a}$. We compute

$$\begin{array}{rcll} [\![t_0]\!]\ \vec{a} & = & Impl\ (Impl\ P_0\ K_{false})\ [\![\mathbf{B}_S\ s_0]\!]\ \vec{a} & \\ & = & implb\ (implb\ a_0\ false)\ ([\![\mathbf{B}_S\ s_0]\!]\ a_0\ a_1 \cdots a_n) & \\ & = & implb\ (implb\ a_0\ false)\ ([\![s_0]\!]\ a_1 \cdots a_n) & \text{(by Lemma 8.3.1)} \\ & = & implb\ (implb\ a_0\ false)\ (g\ false\ a_1 \cdots a_n) & \text{(by the choice of } s_0) \end{array}$$

and

$$\begin{array}{rcll} [\![t_1]\!]\ \vec{a} & = & Impl\ P_0\ [\![\mathbf{B}_S\ s_1]\!]\ \vec{a} & \\ & = & implb\ a_0\ ([\![s_1]\!]\ a_1 \cdots a_n) & \text{(by Lemma 8.3.1)} \\ & = & implb\ a_0\ (g\ true\ a_1 \cdots a_n) & \text{(by the choice of } s_1) \end{array}$$

Assume a_0 is *false*. We know

$$[\![t_0]\!]\ \vec{a} \ = \ g\ false\ a_1 \cdots a_n \ = \ g\ a_0\ a_1 \cdots a_n \ = \ g\vec{a}$$

and $[\![t_1]\!]\ \vec{a}$ equals *true*. Using Lemma 8.1.4 we have

$$[\![s]\!]\ \vec{a} \ = \ andb\ ([\![t_0]\!]\ \vec{a})\ ([\![t_1]\!]\ \vec{a}) \ = \ g\vec{a}$$

as desired.

Next assume a_0 is *true*. We know

$$[\![t_1]\!]\ \vec{a} \ = \ g\ true\ a_1 \cdots a_n \ = \ g\ a_0\ a_1 \cdots a_n \ = \ g\vec{a}$$

and $[\![t_0]\!]\ \vec{a}$ equals *true*. Using Lemma 8.1.4 we have

$$[\![s]\!]\ \vec{a}\ =\ andb\ ([\![t_0]\!]\ \vec{a})\ ([\![t_1]\!]\ \vec{a})\ =\ g\vec{a}$$

as desired. ∎

We can represent the theorem and its proof in Coq as follows.

```
Lemma B_DenCompl (n : nat) : forall (g : bool ^ n --> bool), exists s:B n,
 [[s]] == g.
induction n.
intros [|].
exists (# ==> #). intros a. reflexivity.
exists #. intros a. reflexivity.
intros g.
destruct (IHn (g false)) as [s0 A0].
destruct (IHn (g true)) as [s1 A1].
exists (B_And (B_Not FinO ==> (B_S s0)) (FinO ==> (B_S s1))).
intros a.
rewrite B_And_andb. rewrite Ap_comp2.
simpl. unfold Impl. simpl. repeat rewrite Ap_comp2.
  repeat rewrite Ap_K. repeat rewrite B_Den_S.
destruct (hd a) ; simpl.
apply A1.
rewrite A0. destruct (Ap (g false) (tl a)) ; reflexivity.
Qed.
```

Exercise 8.3.3 Use a script to write a certifying function which, given a cascaded boolean function g, computes a boolean formula s such that $[\![s]\!] \equiv g$.

```
Definition B_DenCompl_F (n : nat) : forall (g : bool ^ n --> bool),
 {s:B n| [[s]] == g}.
```

(Hint: Start with the script proving denotational completeness and make any necessary modifications.)

Exercise 8.3.4 Let $g : bool^3 \to bool$ be the boolean function such that $g\,a\,b\,c$ is *false* iff a and b are *true* and c is *false*. Find a formula $s \in \mathbf{B}_3$ such that $[\![s]\!] \equiv g$. Prove your solution is correct by filling in the missing boolean formula in the following Coq proof.

```
Lemma exg :
let g := fun a b c => match a,b,c with true,true,false => false | _,_,_ => true end
in
{s:B 3|[[s]] == g}.
exists ...
intros a.
```

```
simpl.
destruct (hd a) ; destruct (hd (tl a)) ; destruct (hd (tl (tl a))) ; reflexivity.
Qed.
```

9 Quantified Boolean Logic

We now extend boolean logic to include a universal quantifier. An example of such a formula is

$$\forall x.(\forall y.x \Rightarrow y) \Rightarrow x$$

We will consider formulas to be the same if they are the same up to the names of bound variables. For example, the formula above is the same as

$$\forall y.(\forall x.y \Rightarrow x) \Rightarrow y.$$

One way to realize this convention is to standardize variable names. If we insist on binding the variables in the order $x_0, x_1, \ldots$, then the formula above is represented as

$$\forall x_0.(\forall x_1.x_0 \Rightarrow x_1) \Rightarrow x_0.$$

As in boolean logic, we will consider the formulas that depend on the first n variables: $x_0, \ldots, x_{n-1}$. We generalize our standardization of variables to account for this: If s is a formula depending on the first $n+1$ variables $(x_0, \ldots, x_{n-1}, x_n)$, then $\forall x_n.s$ is a formula depending on the first n variables $(x_0, \ldots, x_{n-1})$.

The semantics of quantified boolean formulas can be defined by recursion on formulas as with boolean logic. Formulas depending on no variables will be interpreted as booleans. Given such a formula s, the problem of determining its value is PSPACE-complete. This was proven by Stockmeyer and Meyer (1973).

We give a tableau method for determining the value of a formula. The tableau method is an alternative to brute force calculation and works well in many cases. We will also show how one can simulate the tableau method in Coq.

9.1 Syntax

We can define quantified boolean formulas $\mathbf{Q}_n$ with n variables to be the least set satisfying the following conditions:

- If $k < n$, then $x_k \in \mathbf{Q}_n$.
- $\sharp \in \mathbf{Q}_n$.
- If $s, t \in \mathbf{Q}_n$, then $(s \Rightarrow t) \in \mathbf{Q}_n$.
- If $s \in \mathbf{Q}_{n+1}$, then $(\forall x_n.s) \in \mathbf{Q}_n$.

In Coq, we represent $\mathbf{Q}_n$ using the following inductive type.

```
Inductive Q (n : nat) : Type :=
| Q_Var : Fin n -> Q n
| Q_Fal : Q n
| Q_Imp : Q n -> Q n -> Q n
| Q_All : Q (S n) -> Q n.
```

Again, we introduce notation.

```
Implicit Arguments Q_Var [n].

Local Coercion Q_Var : Fin >-> Q.

Implicit Arguments Q_Fal [n].
Implicit Arguments Q_Imp [n].

Local Notation "#" := Q_Fal.
Local Notation "s ==> t" := (Q_Imp s t) (at level 90, right associativity).
```

We additionally introduce notation for *Q_All*. We leave the reference to the natural number n.

```
Local Notation "'ALL' x , s " := (Q_All x s) (at level 100).
```

9.2 Semantics

We define the function mapping formulas s to cascaded boolean functions $[\![s]\!]$ as in boolean logic. The new case is for the definition is the quantifier (*Q_All*). Recall the definition of *allb*.

```
Definition allb (p : bool -> bool) : bool := andb (p false) (p true).
```

We can lift this to cascaded functions using composition.

```
Definition All (n:nat) (g:bool ^ (S n) --> bool) : bool ^ n --> bool :=
 comp allb n (unfoldR n g).
```

For each boolean quantified formula s, we recursively define a boolean function

$$[\![s]\!] : bool^n \to bool$$

as follows:

$$[\![x_k]\!] := P_k$$

$$[\![\sharp]\!] := K_{false}$$

$$[\![s \Rightarrow t]\!] := Impl\ [\![s]\!]\ [\![t]\!]$$

$$[\![\forall x_n.s]\!] := All\ [\![s]\!].$$

Study the $\forall$ case carefully. Note that $[\![s]\!] : bool^{n+1} \to bool$ and $[\![\forall x_n.s]\!] : bool^n \to bool$.

Here is the definition in Coq.

```
Fixpoint Q_Den {n : nat} (s : Q n) : bool ^ n --> bool :=
match s with
| Q_Var k => P k
| Q_Fal => K bool false n
| Q_Imp s1 s2 => Impl n (Q_Den s1) (Q_Den s2)
| Q_All s1 => All n (Q_Den s1)
end.

Implicit Arguments Q_Den [n].
Local Notation "[[ s ]]" := (Q_Den s).
```

As an example, we calculate the value of

$$[\![\forall x_0.(\forall x_1.x_0 \Rightarrow x_1) \Rightarrow (x_0 \Rightarrow \sharp)]\!].$$

Consider the subformula $\forall x_1.x_0 \Rightarrow x_1$ in $\mathbf{Q}_1$. In Coq, we can write this formula as *ALL 1, F02 ==> F12*. Note that

$$[\![\forall x_1.x_0 \Rightarrow x_1]\!] : \mathbf{B} \to \mathbf{B}$$

We can calculate

$$[\![\forall x_1.x_0 \Rightarrow x_1]\!]\ true = false$$

as follows:

$$\begin{array}{rcl} [\![\forall x_1.x_0 \Rightarrow x_1]\!]\ true & = & allb\ (\lambda b : bool.[\![x_0 \Rightarrow x_1]\!]\ true\ b) \\ & = & andb\ ([\![x_0 \Rightarrow x_1]\!]\ true\ false)\ ([\![x_0 \Rightarrow x_1]\!]\ true\ true) \\ & = & andb\ false\ true \\ & = & false \end{array}$$

Similarly, we can calculate

$$[\![\forall x_1.x_0 \Rightarrow x_1]\!]\ false = true.$$

Once we know this, it is clear that

$$[\![(\forall x_1.x_0 \Rightarrow x_1) \Rightarrow (x_0 \Rightarrow \sharp)]\!]\ false = true$$

and

$$[\![(\forall x_1.x_0 \Rightarrow x_1) \Rightarrow (x_0 \Rightarrow \sharp)]\!]\ true = true.$$

Using this, we can calculate

$$[\![\forall x_0.(\forall x_1.x_0 \Rightarrow x_1) \Rightarrow (x_0 \Rightarrow \sharp)]\!] = true.$$

Exercise 9.2.1 Calculate the values of $[\![s]\!]$ for each of the following formulas s in $\mathbf{Q}_0$. You can use Coq to do this, but you should also be able to determine the value without Coq.

a) $\forall x_0.x_0$

b) $\forall x_0.x_0 \Rightarrow \sharp$

c) $\forall x_0.(x_0 \Rightarrow \sharp) \Rightarrow x_0$.

d) $\forall x_0.((x_0 \Rightarrow \sharp) \Rightarrow \sharp) \Rightarrow x_0$.

e) $\forall x_0.(\forall x_1.x_0 \Rightarrow x_1) \Rightarrow x_0$.

f) $\forall x_0.((\forall x_1.x_0 \Rightarrow x_1) \Rightarrow x_0) \Rightarrow x_0$.

9.3 Tableaux for Quantified Boolean Formulas

We now consider a tableau method for determining if $[\![s]\!]$ is *true* or *false* for a given $s \in \mathbf{Q}_0$. The idea is that we assume $[\![s]\!]$ is the other boolean, and consider all consequences until we determine that the assumption is impossible.

Consider the formula

$$\forall x_0.\forall x_1.((x_0 \Rightarrow x_1) \Rightarrow x_0) \Rightarrow x_0.$$

This is (the quantified form of) **Peirce's Law**. We can use the definition of $[\![]\!]$ to calculate

$$[\![\forall x_0.\forall x_1.((x_0 \Rightarrow x_1) \Rightarrow x_0) \Rightarrow x_0]\!] = \mathit{true}.$$

Alternatively, we can argue that the value is *true* as follows.

1. Assume $[\![\forall x_0.\forall x_1.((x_0 \Rightarrow x_1) \Rightarrow x_0) \Rightarrow x_0]\!] = \mathit{false}$.
2. There is some boolean a such that $[\![\forall x_1.((x_0 \Rightarrow x_1) \Rightarrow x_0) \Rightarrow x_0]\!]\, a = \mathit{false}$.
3. There is some boolean b such that $[\![((x_0 \Rightarrow x_1) \Rightarrow x_0) \Rightarrow x_0]\!]\, a\, b = \mathit{false}$.
4. For this implication to evaluate to *false* we must have two facts:
5. $[\![(x_0 \Rightarrow x_1) \Rightarrow x_0]\!]\, a\, b = \mathit{true}$ and
6. $[\![x_0]\!]\, a\, b = \mathit{false}$. Equivalently, $a = \mathit{false}$.
7. By (5) we know either $[\![x_0 \Rightarrow x_1]\!]\, a\, b = \mathit{false}$ or $[\![x_0]\!]\, a\, b = \mathit{true}$. If $[\![x_0]\!]\, a\, b = \mathit{true}$, then $a = \mathit{true}$ which conflicts with (6). Hence we need only consider the first possibility.
8. Assume $[\![x_0 \Rightarrow x_1]\!]\, a\, b = \mathit{false}$.
9. Hence $[\![x_0]\!]\, a\, b = \mathit{true}$ which conflicts with (6).

We conclude that it is impossible for $[\![\forall x_0.\forall x_1.((x_0 \Rightarrow x_1) \Rightarrow x_0) \Rightarrow x_0]\!]$ to be *false*. Consequently, it is *true*.

Such arguments can be organized as tableau refutations. At each step of a tableau refutation, we have (possibly several) sets of assumptions of the form

$[\![s]\!]\,\vec{a} = \mathit{false}$ or $[\![s]\!]\,\vec{a} = \mathit{true}$. Here $\vec{a}$ will range over lists of boolean parameters and values. The only boolean values are *true* and *false*. We assume an infinite set of boolean parameters $b_0, b_1, b_2, \ldots$.

A **branch** is a finite set of assumptions of the form $[\![s]\!]\,\vec{a} = \mathit{false}$ or $[\![s]\!]\,\vec{a} = \mathit{true}$. We write $[\![s]\!]\,\vec{a}^+$ as shorthand for $[\![s]\!]\,\vec{a} = \mathit{true}$ and write $[\![s]\!]\,\vec{a}^-$ as shorthand for $[\![s]\!]\,\vec{a} = \mathit{false}$. We sometimes also write $[\![s]\!]\,\vec{a}^*$ where $*$ ranges over $+$ and $-$. We say a boolean parameter is b **occurs** in a branch A if there is an assumption $[\![s]\!]\,\vec{a}^*$ where b in the list $\vec{a}$. Otherwise, we say b is **fresh** for the branch A.

Our goal is to prove a contradiction from each such set of assumptions. We make progress towards this goal by applying certain directed rules. A branch A is **closed** if any of the following hold:

1. $[\![\sharp]\!]\,\vec{a}^+$ is in A.
2. $[\![x_k]\!]\,\vec{a}^+$ is in A and a_k is *false*.
3. $[\![x_k]\!]\,\vec{a}^-$ is in A and a_k is *true*.
4. There is some x_k such that both $[\![x_k]\!]\,\vec{a}^+$ and $[\![x_k]\!]\,\vec{b}^-$ are in A where a_k and b_k are the same boolean parameter.

The tableau system inductively defines the set of **refutable** branches as follows.

Closed: Every closed branch is refutable.

$\mathcal{T}^+_\Rightarrow$: If $[\![s \Rightarrow t]\!]\vec{a}^+$ is in A, $A \cup \{[\![s]\!]\vec{a}^-\}$ is refutable and $A \cup \{[\![t]\!]\vec{a}^+\}$ is refutable, then A is refutable.

$\mathcal{T}^-_\Rightarrow$: If $[\![s \Rightarrow t]\!]\vec{a}^-$ is in A and $A \cup \{[\![s]\!]\vec{a}^+,\ [\![t]\!]\vec{a}^-\}$ is refutable, then A is refutable.

$\mathcal{T}^+_\forall$: If $[\![\forall x.s]\!]\vec{a}^+$ is in A, b is a boolean value or boolean parameter which occurs in A and $A \cup \{[\![s]\!]\vec{a}b^+\}$ is refutable, then A is refutable.

$\mathcal{T}^-_\forall$: If $[\![\forall x.s]\!]\vec{a}^-$ is in A, b is a boolean parameter fresh for A and $A \cup \{[\![s]\!]\vec{a}b^-\}$ is refutable, then A is refutable.

The rules are summarized in Figure 9.1. We briefly describe the four rules and how to use them to refute a branch A.

The positive implication rule is

$$\mathcal{T}^+_\Rightarrow \quad \frac{[\![s \Rightarrow t]\!]\,\vec{a}^+}{[\![s]\!]\,\vec{a}^- \mid [\![t]\!]\,\vec{a}^+}$$

Suppose $[\![s \Rightarrow t]\!]\,\vec{a}^+$ is in A. Refute two branches: One with the extra assumption $[\![s]\!]\,\vec{a}^-$ and one with the extra assumption $[\![t]\!]\,\vec{a}^+$.

The negative implication rule is

$$\mathcal{T}^-_\Rightarrow \quad \frac{[\![s \Rightarrow t]\!]\,\vec{a}^-}{[\![s]\!]\,\vec{a}^+,\ [\![t]\!]\,\vec{a}^-}$$

Suppose $[\![s \Rightarrow t]\!]\,\vec{a}^-$ is in A. Refute the branch with two extra assumptions: $[\![s]\!]\,\vec{a}^+$ and $[\![t]\!]\,\vec{a}^-$.

$$\mathcal{T}^+_{\Rightarrow}\ \frac{[\![s \Rightarrow t]\!]\,\vec{a}^+}{[\![s]\!]\,\vec{a}^- \mid [\![t]\!]\,\vec{a}^+} \qquad \mathcal{T}^-_{\Rightarrow}\ \frac{[\![s \Rightarrow t]\!]\,\vec{a}^-}{[\![s]\!]\,\vec{a}^+,\ [\![t]\!]\,\vec{a}^-} \qquad \mathcal{T}^+_{\forall}\ \frac{[\![\forall x.s]\!]\,\vec{a}^+}{[\![s]\!]\,\vec{a}\,b^+}\ b : \mathit{bool}$$

$$\mathcal{T}^-_{\forall}\ \frac{[\![\forall x.s]\!]\,\vec{a}^-}{[\![s]\!]\,\vec{a}\,b^-}\ b : \mathit{bool}\ \text{fresh}$$

Figure 9.1: Tableau rules for quantified boolean formulas

The positive forall rule is

$$\mathcal{T}^+_{\forall}\ \frac{[\![\forall x.s]\!]\,\vec{a}^+}{[\![s]\!]\,\vec{a}\,b^+}\ b : \mathit{bool}$$

Suppose $[\![\forall x.s]\!]\,\vec{a}^+$ is in A. Choose b to be an appropriate boolean value (either *true* or *false*) or one of the boolean parameters which occurs in A. Refute the branch with the additional assumption $[\![s]\!]\,\vec{a}\,b^+$. Note that some creativity may be required to choose a helpful b. Also, this rule may be applied more than once to the same formula with different choices of b.

The negative forall rule is

$$\mathcal{T}^-_{\forall}\ \frac{[\![\forall x.s]\!]\,\vec{a}^-}{[\![s]\!]\,\vec{a}\,b^-}\ b : \mathit{bool}\ \text{fresh}$$

Suppose $[\![\forall x.s]\!]\,\vec{a}^-$ is in A. Let b be a boolean parameter fresh for A. Refute the branch with the additional assumption $[\![s]\!]\,\vec{a}\,b^-$.

A tableau refutation can be presented as a tree in which every branch is closed. For example, the refutation of Peirce's Law described earlier can be given as the tableau refutation shown in Figure 9.3. At the root of the tree is the assumption we wish to prove is impossible. Each of the other assumptions in the tree have been added by applying rules. Note that a and b are used to refer to (distinct) boolean parameters. There are two branches, both of which are closed because they contain $[\![x_0]\!]\,a\,b^+$ and $[\![x_0]\!]\,a\,b^-$.

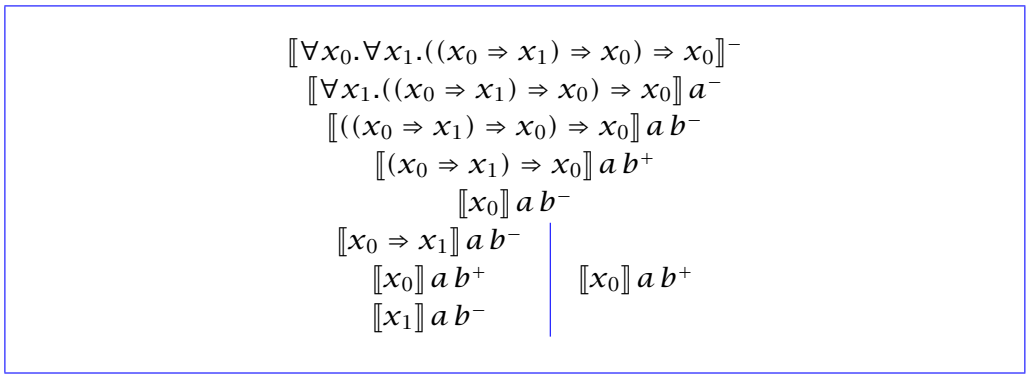
Exercise 9.3.1 Determine the value of $[\![s]\!]$ for each of the following formulas $s \in \mathbf{Q}_0$. Use a tableau refutation (presented as a tree) to justify your answer.

a) $\forall x_0.x_0$

b) $\forall x_0.x_0 \Rightarrow \sharp$

c) $\forall x_0.(\forall x_1.x_0 \Rightarrow x_1) \Rightarrow x_0$.

d) $\forall x_0.((\forall x_1.x_0 \Rightarrow x_1) \Rightarrow x_0) \Rightarrow x_0$.



9.4 Simulating Tableau in Coq

We can simulate tableau refutations using proof scripts in Coq. We first give two simple lemmas which will allow us to prove the goal we want by refuting a branch with a single assumption.

```
Lemma QT_Start (s:Q 0) :
 (Ap [[s]] nl = false -> False) -> [[s]] = true.
simpl. destruct [[s]]; stauto.
Qed.

Lemma QF_Start (s:Q 0) :
 (Ap [[s]] nl = true -> False) -> [[s]] = false.
simpl. destruct [[s]]; stauto.
Qed.
```

After applying one of these lemmas, the assumptions of the Coq goal will correspond to the branch. Also, variables of type *bool* will play the role of boolean parameters.

The following lemmas can be used to prove a branch is closed. (That is, a set of assumptions is inconsistent.)

```
Lemma Q_Conflict {b:bool} :
 b = true -> b = false -> False.
destruct b; discriminate.
Qed.

Lemma QT_Fal {n:nat} {a:ilist bool n} :
  Ap [[#]] a = true -> False.
simpl. rewrite Ap_K. discriminate.
Qed.
```

We next give lemmas corresponding to the positive and negative implication rules.

```
Lemma QT_Imp {n:nat} {s1 s2:Q n} {a:ilist bool n} :
   Ap [[s1 ==> s2]] a = true
-> (Ap [[s1]] a = false -> False)
-> (Ap [[s2]] a = true -> False)
-> False.
simpl. unfold Impl. rewrite Ap_comp2.
destruct (Ap [[s1]] a);  stauto.
Qed.

Lemma QF_Imp {n:nat} {s1 s2:Q n} {a:ilist bool n} :
   Ap [[s1 ==> s2]] a = false
-> (Ap [[s1]] a = true -> Ap [[s2]] a = false -> False)
-> False.
simpl. unfold Impl. rewrite Ap_comp2.
destruct (Ap [[s1]] a).  stauto. discriminate.
Qed.
```

Before we state and prove the positive and negative forall rules, we will need some more infrastructure. We need to be able to append a single element onto an *ilist* of length n and obtain an *ilist* of length Sn. We do this with a function *extend*.

```
Fixpoint extend {n:nat} {A:Type} (al:ilist A n) (b:A) :  ilist  A (S n) :=
match al with
| nl => (b::nl)
| (a::ar) => (a::(extend ar b))
end.
```

Using *extend* we can state and prove a lemma about the auxiliary function *unfoldR*.

```
Lemma Ap_unfoldR (A B:Type) (n:nat) (g:A ^ (S n) -—> B) (a:ilist A n) (a':A) :
  Ap (unfoldR n g) a a' = Ap g (extend a a').
induction a as [ |n a0 ar IH].
simpl. reflexivity.
simpl. exact (IH (g a0)).
Qed.
```

We now state and prove the lemmas corresponding to the positive and negative forall rules.

```
Lemma QT_All {n:nat} {s:Q (S n)} {a:ilist bool n} (b:bool) :
   Ap [[ALL n, s]] a = true
-> (Ap [[s]] (extend a b) = true -> False)
-> False.
simpl. unfold All. rewrite Ap_comp. unfold allb. repeat rewrite Ap_unfoldR.
simpl.
destruct b;
```

```
destruct (Ap ([[s]]  (hd (extend a true))) (tl  (extend a true)));
destruct (Ap ([[s]]  (hd (extend a false))) (tl  (extend a false)));
try  stauto.
Qed.

Lemma QF_All {n:nat} {s:Q (S n)} {a:ilist bool n} :
   Ap [[ALL n, s]]  a = false
-> (forall b:bool, Ap [[s]]  (extend a b) = false -> False)
-> False.
simpl. unfold All. rewrite Ap_comp. unfold allb. repeat rewrite Ap_unfoldR.
simpl. intros H1 H2. generalize (H2 false). generalize (H2 true).
destruct (Ap ([[s]]  (hd (extend a true))) (tl  (extend a true)));
destruct (Ap ([[s]]  (hd (extend a false))) (tl  (extend a false)));
try  stauto.
Qed.
```

We can now prove goals of the form $[\![s]\!] = \mathit{true}$ or $[\![s]\!] = \mathit{false}$ where $s \in \mathbf{Q}_0$ by simulating tableau refutations in Coq. We begin the proof script with either

```
apply QT_Start; intros H1.
```

or

```
apply QF_Start; intros H1.
```

The choice depends on whether one is proving $[\![s]\!]$ or $\neg[\![s]\!]$. After this, each line of the proof script will correspond to an application of a tableau rule or a tactic indicating the current branch is closed. If the current branch is closed because there is an assumption H of the form $x_k\vec{a}^+$ and a_k is *false* or $x_k\vec{a}^-$ and a_k is *true*, then the assumption is convertible to either $\mathit{false} = \mathit{true}$ or $\mathit{true} = \mathit{false}$ and we can indicate the branch is closed using *discriminate*:

```
discriminate H.
```

(Coq does not need the name H of the hypothesis, but let us always include it in tableau refutations for clarity.) If the current branch is closed because there is an assumption $H : [\![\sharp]\!]\,\vec{a}^+$, we can either use *discriminate H* or

```
exact (QT_Fal H).
```

If the current branch is closed because there is an assumption $H : [\![x_k]\!]\,\vec{a} = \mathit{true}$ and an assumption $H' : [\![x_k]\!]\,\vec{b} = \mathit{false}$ where a_k is the same as b_k, we can indicate the branch is closed as follows:

```
exact (QT_Conflict H H').
```

The four rules are be applied as follows:

$\mathcal{T}^+_\Rightarrow$: If Hi is an assumption $[\![s \Rightarrow t]\!]\vec{a}^+$, then

```
apply (QT_Imp Hi); intros Hj.
```

creates two new subgoals with one new assumption *Hj* in both cases.

$\mathcal{T}^-_\Rightarrow$: If *Hi* is an assumption $[\![s \Rightarrow t]\!]\vec{a}^-$, then

apply (QF_Imp Hi); intros Hj Hk.

creates a subgoal with two new assumptions *Hj* and *Hk*.

$\mathcal{T}^+_\forall$: If *Hi* is an assumption $[\![\forall x.s]\!]\vec{a}^+$ and *b* is either *false*, *true* or a boolean in the current assumption context, then

apply (QT_All b Hi); intros Hj.

adds an assumption *Hj*.

$\mathcal{T}^-_\forall$: If *Hi* is an assumption $[\![\forall x.s]\!]\vec{a}^-$, then

apply (QF_All Hi); intros b Hj.

adds an assumed boolean *b:bool* and an assumption *Hj*.

Here is the proof of Peirce's Law using a tableau refutation in Coq.

```
Lemma PeirceTabl: [[ALL 0, ALL 1, ((F02 ==> F12) ==> F02) ==> F02]] = true.
apply QT_Start; intros H1.
apply (QF_All H1); intros a H2.
apply (QF_All H2); intros b H3.
apply (QF_Imp H3); intros H4 H5.
apply (QT_Imp H4); intros H6.
apply (QF_Imp H6); intros H7 H8.
exact (Q_Conflict H7 H5).
exact (Q_Conflict H6 H5).
Qed.
```

9.5 Soundness and Completeness

We briefly consider two properties the notion of refutation should have: soundness and completeness. In the context of quantified boolean logic and tableau refutations we can state the soundness and completeness properties as follows. For simplicity we let σ range over $\{+,-\}$ and let $-\sigma$ be $+$ if σ is $-$ and let $-\sigma$ be $-$ if σ is $+$.

Soundness: For all $s \in \mathbf{Q}_0$ and $\sigma \in \{+,-\}$ if $\{[\![s]\!]^{-\sigma}\}$ is refutable, then $[\![s]\!]^{\sigma}$.

Completeness: For all $s \in \mathbf{Q}_0$ and $\sigma \in \{+,-\}$ if $[\![s]\!]^{\sigma}$, then $\{[\![s]\!]^{-\sigma}\}$ is refutable.

Soundness follows from the lemmas in the previous section and the fact that we can simulate tableau refutations in Coq.

Completeness can be argued as follows. Suppose $[\![s]\!]^{\sigma}$, but $[\![s]\!]^{-\sigma}$ is not refutable. By applying the tableau rules starting with the initial branch $\{[\![s]\!]^{-\sigma}\}$ we can obtain a branch H such that H is not closed, $[\![s]\!]^{-\sigma} \in H$ and the following conditions hold:

- If $[\![t_1 \Rightarrow t_2]\!]\vec{a}^+$ is in H, then either $[\![t_1]\!]\vec{a}^-$ or $[\![t_2]\!]\vec{a}^+$ is in H.
- If $[\![t_1 \Rightarrow t_2]\!]\vec{a}^-$ is in H, then $[\![t_1]\!]\vec{a}^+$ and $[\![t_2]\!]\vec{a}^-$ are in H.
- If $[\![\forall x.t]\!]\vec{a}^+$ is in H, then $[\![\forall x.t]\!]\vec{a}\mathit{false}^+$ and $[\![\forall x.t]\!]\vec{a}\mathit{true}^+$ are in H.
- If $[\![\forall x.t]\!]\vec{a}^-$ is in H, then there is some boolean parameter b such that $[\![\forall x.t]\!]\vec{a}b^-$ is in H.

Furthermore, by restricting the $\mathcal{T}_\forall^+$ rule to use *false* and *true* but note boolean parameters we can guarantee that for every boolean parameter b there is some k such that for all $[\![s]\!]\vec{a}^\sigma$ in H, if a_i is b, then $i = k$. That is, for every parameter b there is some k so that the parameter b occurs only in position k.

An **assignment** is a function ϕ mapping each boolean parameter into $\{\mathit{false}, \mathit{true}\}$. Given an assignment ϕ and a vector $\vec{a}$, let $\phi(\vec{a})$ be the vector such that $\phi(\vec{a})_k := \phi(a_k)$ if a_k is a parameter, $\phi(\vec{a})_k := \mathit{false}$ if a_k is *false* and $\phi(\vec{a})_k := \mathit{true}$ if a_k is *true*.

Lemma 9.5.1 There is an assignment ϕ such that if $[\![x_k]\!]\vec{a}^\sigma$ is in H, then $[\![x_k]\!]\phi(\vec{a})^\sigma$.

Proof For each boolean parameter b, let k be such that b occurs only in position k. Let $\phi b := \mathit{true}$ if there is some $[\![x_k]\!]\vec{a}^+$ in H such that a_k is b. Otherwise, let $\phi b := \mathit{false}$. Suppose $[\![x_k]\!]\vec{a}^+$ is in H. We know a_k is not *false* since H is not closed. If a_k is *true*, then we are done. If a_k is a boolean parameter b, then we have chosen ϕ such that ϕa_k is *true*. Next suppose $[\![x_k]\!]\vec{a}^-$ is in H. We know a_k is not *true* since H is not closed. If a_k is *false*, then we are done. Otherwise, a_k is a boolean parameter. We will be done if we know ϕa_k is *false*. Assume ϕa_k is *true*. There must be some $[\![x_k]\!]\vec{b}^+$ in H such that b_k is a_k. This contradicts the fact that H is not closed. ∎

Lemma 9.5.2 Let ϕ be an assignment such that if $[\![x_k]\!]\vec{a}^\sigma$ is in H, then $[\![x_k]\!]\phi(\vec{a})^\sigma$. For each $[\![s]\!]\vec{a}^\sigma$ in H we know $[\![s]\!]\phi(\vec{a})^\sigma$.

Proof We prove this by induction on s. The variable case is true by our assumption about ϕ. We know $[\![\sharp]\!]\phi(\vec{a})^-$ holds. Since H is not closed, $[\![\sharp]\!]\phi(\vec{a})^+$ is not in H.

Assume $[\![t_1 \Rightarrow t_2]\!]\vec{a}^+$ is in H. Then either $[\![t_1]\!]\vec{a}^-$ or $[\![t_2]\!]\vec{a}^+$ is in H. By the inductive hypothesis, either $[\![t_1]\!]\phi(\vec{a})^-$ or $[\![t_2]\!]\phi(\vec{a})^+$. In either case we have $[\![t_1 \Rightarrow t_2]\!]\phi(\vec{a})^+$.

Assume $[\![t_1 \Rightarrow t_2]\!]\vec{a}^-$ is in H. Then $[\![t_1]\!]\vec{a}^+$ and $[\![t_2]\!]\vec{a}^-$ are in H. By the inductive hypothesis, $[\![t_1]\!]\phi(\vec{a})^+$ and $[\![t_2]\!]\phi(\vec{a})^-$ and so $[\![t_1 \Rightarrow t_2]\!]\phi(\vec{a})^-$.

Assume $[\![\forall x_n.t]\!]\vec{a}^+$ is in H. Then $[\![t]\!]\vec{a}\mathit{true}^+$ and $[\![t]\!]\vec{a}\mathit{false}^+$ are in H. By the inductive hypothesis, $[\![t]\!]\phi(\vec{a}\mathit{true})^+$ and $[\![t]\!]\phi(\vec{a}\mathit{false})^+$ and so $[\![\forall x_n.t]\!]\phi(\vec{a})^+$.

Assume $[\![\forall x_n.t]\!]\vec{a}^-$ is in H. Then $[\![t]\!]\vec{a}b^-$ is in H for some boolean parameter b. By the inductive hypothesis, $[\![t]\!]\phi(\vec{a}b)^-$ and so $[\![\forall x_n.t]\!]\phi(\vec{a})^-$. ∎

Theorem 9.5.3 (Completeness) For all $s \in \mathbf{Q}_0$ and $\sigma \in \{+,-\}$ if $[\![s]\!]^\sigma$, then $\{[\![s]\!]^{-\sigma}\}$ is refutable.

Proof We argue as above with a branch H such that $[\![s]\!]^{-\sigma} \in H$. Let ϕ be an assignment given by Lemma 9.5.1. By Lemma 9.5.2 we know $[\![s]\!]^{-\sigma}$ holds. This contradicts $[\![s]\!]^\sigma$. ∎

9.6 Other Interpretations

We interpreted formulas of both boolean logic and quantified boolean logic as cascaded boolean functions. There are other interpretations. For example, we could interpret formulas as cascaded functions on the universe *Prop*. For $\mathbf{B}$ we define $[\![-]\!]^{\mathbf{B}}_P$ as follows:

$$[\![x_k]\!]^{\mathbf{B}}_P := P_k$$

$$[\![\sharp]\!]^{\mathbf{B}}_P := K_{False}$$

$$[\![s \Rightarrow t]\!]^{\mathbf{B}}_P := implP \circ^2 ([\![s]\!]^{\mathbf{B}}_P, [\![t]\!]^{\mathbf{B}}_P)$$

where

```
Definition implP (p q:Prop) : Prop := p -> q.
```

For $\mathbf{Q}$ we define $[\![-]\!]^{\mathbf{Q}}_P$ as above with the extra clause

$$[\![\forall x_n.s]\!]^{\mathbf{Q}}_P := allP \circ [\![s]\!]^{\mathbf{Q}}_P$$

where

```
Definition allP (p:Prop -> Prop) : Prop := forall q:Prop, p q.
```

We can then say a formula $s \in \mathbf{B}_n$ is **constructively valid** if the proposition $\forall \vec{a} : Prop.[\![s]\!]^{\mathbf{B}}_P \vec{a}$ is provable in Coq. Likewise we can say a formula $s \in \mathbf{Q}_n$ is **constructively valid** if the proposition $\forall \vec{a} : Prop.[\![s]\!]^{\mathbf{Q}}_P \vec{a}$ is provable in Coq.

A simple example of a formula in $\mathbf{B}_2$ which is valid, but not constructively valid is Peirce's law:

$$((x_0 \Rightarrow x_1) \Rightarrow x_0) \Rightarrow x_0$$

Also, the following double negation law in $\mathbf{B}_1$ is valid but not constructively valid

$$((x_0 \Rightarrow \sharp) \Rightarrow \sharp) \Rightarrow \sharp$$

The problem of determining whether $s \in \mathbf{B}_n$ is constructively valid is PSPACE-complete (Statman, 1979). The problem of determining whether $s \in \mathbf{Q}_n$ is constructively valid is undecidable (Löb, 1976).

10 Mathematical Assumptions

In Chapter 4 we briefly discussed excluded middle and the double negation law as propositions that are not provable in Coq. Nevertheless, these are natural assumptions one makes when working in a mathematical context. In this chapter we consider different propositions which are not provable in Coq but are natural mathematical assumptions.

10.1 Classical Assumptions

Recall the propositions *XM* (excluded middle) and *DN* (the double negation law).

```
Definition XM : Prop := forall X : Prop, X \/ ~X.

Definition DN : Prop := forall X : Prop, ~~X -> X.
```

Neither of these are provable in Coq, but both are natural mathematical assumptions. In fact, they are provably equivalent in Coq (see Chapter 4), so that by assuming either one, we obtain a proof of the other.

Another equivalent proposition is Peirce's law.

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

Exercise 10.1.1 Prove *PEIRCE* and *DN* are equivalent.

After considering *XM*, *DN* and *PEIRCE*, one may get the impression that all classical assumptions are equivalent. This is not the case. An example is the **Gödel-Dummett (*GD*)** proposition:

```
Definition GD :=
 forall X Y:Prop, (X -> Y) \/ (Y -> X).
```

The proposition *GD* is not provable in Coq. We can prove *GD* from *XM*, but *XM* is not provable from *GD* in Coq.

Exercise 10.1.2 Prove *GD* follows from *XM*.

```
Goal XM -> GD.
```

10.2 Extensional Assumptions

Extensional assumptions allow us to prove certain objects are equal by proving they have a common property. None of these propositions are provable in Coq.

- **Functional Extensionality (*FE*):** Two functions are equal if they have the same value on all arguments.

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

- **Propositional Extensionality (*PE*):** Two propositions are equal if they are equivalent.

```
Definition PE : Prop :=
 forall X Y : Prop, (X <-> Y) -> X = Y.
```

Recall that we often compared cascaded functions g and h using $\forall \vec{a}.g\vec{a} = h\vec{a}$ instead of simply $g = h$. Using functional extensionality we can prove $g = h$ follows from $\forall \vec{a}.g\vec{a} = h\vec{a}$.

```
Lemma FE_Ap {n:nat} (X Y:Type) (g h:X ^ n -=> Y) :
    FE
 -> (forall a: ilist  X n, Ap g a = Ap h a)
 -> g = h.
intros fe.
induction n.
intros H. exact (H nl).
intros H. apply (fe X (X ^ n -=> Y) g h).
intros x. apply IHn.
intros a. exact (H (x::a)).
Qed.
```

Exercise 10.2.1 Consider the following **Set Extensionality** property *SE*:

```
Definition SE : Prop :=
 forall X : Type, forall Y Z : X -> Prop, (forall x:X, Y x <-> Z x) -> Y = Z.
```

Prove *SE* follows from *FE* and *PE*.

```
Lemma FE_PE_SE : FE -> PE -> SE.
```

The following proposition is a combination of a classical principle and an extensional principle.

- **Propositional Case Analysis (*PCA*):** Every proposition is either *True* or *False*.

```
Definition PCA : Prop := forall X : Prop, X = True \/ X = False.
```

Exercise 10.2.2 Prove *PCA* is equivalent to the combination of *XM* and *PE*.

```
Goal (PCA -> XM).
```

```
Goal (PCA -> PE).
```

```
Goal (XM -> PE -> PCA).
```

10.3 Proof Irrelevance

Another proposition which is not provable in Coq, but is a natural assumption is **proof irrelevance (*PI*)**. Proof irrelevance says that there is at most one proof of any proposition.

```
Definition PI : Prop := forall X:Prop, forall y z:X, y = z.
```

In Coq, one can prove *XM* implies *PI*. Also, one can prove *PE* implies *PI*. These proofs are beyond the scope of this course. Note, however, that without the elim restriction, we could distinguish proofs. This would allow us to prove the negation of *PI* and hence the negation of both *XM* and *PE*. One important reason for the elim restriction is so that Coq remains consistent with classical logic.

11 First-Order Logic

In this chapter we consider the syntax and semantics of first-order logic. We will restrict ourselves to first-order logic with a single binary relation **E**. The semantics will be given using directed graphs.

For this chapter we assume excluded middle.

```
Variable xm:XM.
```

11.1 Syntax

We define the set $\mathbf{F}_n$ of first-order formulas with n variables to be the least set satisfying the following conditions:

- If $j, k < n$, then $\mathbf{E}(x_j, x_k) \in \mathbf{F}_n$.
- $\sharp \in \mathbf{F}_n$.
- If $s, t \in \mathbf{F}_n$, then $(s \Rightarrow t) \in \mathbf{F}_n$.
- If $s \in \mathbf{F}_{n+1}$, then $(\forall x_n.s) \in \mathbf{F}_n$.

Here is the definition in Coq, along with some notational conventions:

```
Inductive F (n : nat) : Type :=
| F_E : forall x y:Fin n, F n
| F_Fal : F n
| F_Imp : F n -> F n -> F n
| F_All : F (S n) -> F n.

Implicit Arguments F_E [n].
Implicit Arguments F_Fal [n].
Implicit Arguments F_Imp [n].

Local Notation "#" := F_Fal.
Local Notation "s ==> t" := (F_Imp s t) (at level 90, right associativity).
Local Notation "'ALL' x , s " := (F_All x s) (at level 100).
```

11.2 Semantics

The semantics of our first order language is given via models. A **model** M is a nonempty directed graph (V, E) where $V \neq \emptyset$ and $E \subseteq V \times V$. In Coq, we can define a type *Mod* of all models as an inductive type.

```
Definition inhabited (V : Type) : Prop := exists x : V, True.

Inductive Mod : Type :=
| Mc : forall  V:Type, inhabited V -> forall E:V -> V -> Prop, Mod.
```

We can also define functions extracting the type of vertices, the fact that there is a vertex and the edge relation from a given model.

```
Coercion V_ (M:Mod) : Type := match M with Mc V _ _ => V end.

Definition VI_ (M:Mod) : inhabited M := match M with Mc V' VI' _ => VI' end.

Definition E_ (M:Mod) : M -> M -> Prop := match M with Mc V' _ E' => E' end.
```

The fact that $V\_$ is defined as a coercion means we can treat any model $M : \mathit{Mod}$ as a type. In mathematical notation, given a model $M = (V, E)$ we will write V_M for V and E_M for E.

For each first-order formula $s \in \mathbf{F}_n$,

$$[\![s]\!]_M : (V_M)^n \to \mathit{Prop}$$

is defined as follows:

$$[\![\mathbf{E}(x_j, x_k)]\!]_M := E_M \circ^2 (P_j, P_k)$$

$$[\![\sharp]\!]_M := K_{\mathit{False}}$$

$$[\![s \Rightarrow t]\!]_M := (\lambda pq : \mathit{Prop}.p \to q) \circ^2 ([\![s]\!]_M, [\![t]\!]_M)$$

$$[\![\forall x_n.s]\!]_M := (\lambda p : V_M \to \mathit{Prop}.\forall v : V_M.pv) \circ [\![s]\!]_M.$$

Applying the cascaded functions to $\vec{a}$, we have

$$[\![\mathbf{E}(x_j, x_k)]\!]_M \vec{a} = E_M(a_j, a_k)$$

$$[\![\sharp]\!]\vec{a} := \mathit{False}$$

$$[\![s \Rightarrow t]\!]_M \vec{a} := [\![s]\!]_M \vec{a} \to [\![t]\!]_M \vec{a}$$

$$[\![\forall x_n.s]\!]_M \vec{a} := \forall v : V_M.[\![s]\!]_M \vec{a} v.$$

In Coq notation, we define this as

```
Section Interp1.

Variable M : Mod.

Fixpoint F_Den {n : nat} (s : F n) : M ^ n --> Prop :=
match s with
| F_E j k => comp2 (E_ M) n (P j) (P k)
| F_Fal => K M False n
| F_Imp s1 s2 => comp2 (fun p q:Prop => p -> q) n (F_Den s1) (F_Den s2)
| F_All s1 => comp (fun p : M -> Prop => forall v:M, p v) n (unfoldR n (F_Den s1))
end.

End Interp1.

Local Notation "[[ s ]] M" := (F_Den M s) (at level 40).
```

A formula $s \in \mathbf{F}_0$ is **valid** if $[\![s]\!]_M$ is true for every model M. A formula $s \in \mathbf{F}_0$ is **satisfiable** if $[\![s]\!]_M$ is true for some model M. We say $s \in \mathbf{F}_0$ is **unsatisfiable** if it is not satisfiable. In Coq we can define validity and satisfiability as follows.

```
Definition F_Valid (s : F 0) : Prop :=
 forall M:Mod, F_Den M s.

Definition F_Sat (s : F 0) : Prop :=
 exists M:Mod, F_Den M s.
```

It is easy to prove (using excluded middle) that a formula s is valid iff $s \Rightarrow \sharp$ is unsatisfiable. Likewise, a formula s is unsatisfiable iff $s \Rightarrow \sharp$ is valid. Here is a proof of one of these facts in Coq.

```
Lemma F_Val_Unsat (s:F 0) : F_Sat s <-> ~ F_Valid (s ==> #).
split.
intros [M H1] H2. exact (H2 M H1).
intros H1. destruct (xm (F_Sat s)) as [H2|H2].
exact H2.
destruct H1. intros M. simpl. intros H3. apply H2. exists M. exact H3.
Qed.
```

Exercise 11.2.1 Prove s is valid iff $s \Rightarrow \sharp$ is unsatisfiable. (Hint: Use excluded middle.)

```
Lemma F_Val_Unsat' (s:F 0) : F_Valid s <-> ~ F_Sat (s ==> #).
```

Proving validity of formulas in Coq is quite easy once one understands the definitions. Suppose we wish to prove the formula

$$(\forall x_0.(\forall x_1.\mathbf{E}(x_0,x_1)) \Rightarrow \mathbf{E}(x_0,x_0)$$

is valid. The can be written as the following goal in Coq.

```
Goal (F_Valid (ALL 0, (ALL 1, F_E F02 F12) ==> F_E F01 F01)).
```

We begin the proof by assuming we have a model M. We then simplify.

```
intros M. simpl.
```

Simplification reduces $[\![\forall x_0.(\forall x_1.\mathbf{E}(x_0,x_1)) \Rightarrow \mathbf{E}(x_0,x_0)]\!]_M$ to the Coq proposition $\forall v : V_M.(\forall w : M.E_M\ v\ w) \rightarrow E_M\ v\ v$. This is, of course, easy to prove.

```
intros x H. exact (H x).
Qed.
```

Exercise 11.2.2 Prove the formula

$$(\forall x_0.\forall x_1.(\forall x_2.\mathbf{E}(x_0,x_2)) \Rightarrow (\forall x_2.\mathbf{E}(x_2,x_1) \Rightarrow \sharp) \Rightarrow \sharp$$

is valid by proving the following proposition in Coq.

```
Goal (F_Valid (ALL 0, ALL 1, (ALL 2, F_E F03 F23) ==> (ALL 2, F_E F23 F13 ==> #) ==> #)).
```

Validity of first-order formulas is not decidable. It is semi-decidable. That is, one can write an algorithm to enumerate valid formulas. One cannot write an algorithm to enumerate satisfiable formulas.

Coq Summary

This appendix lists the Coq commands used in these notes. Consult the Coq reference manual for more information.

A.1 Commands

The Coq interpreter processes *commands*. Commands start with a keyword and end with a dot. The interpreter has two modes, the *top mode* and the *proof mode*. The interpreter starts in top mode. There are commands that switch from top mode to proof mode and back. We will use the word command for both the operation performed by the command and the application of the operation to arguments.

In this section we summarize the commands for the top mode. Commands for the top mode also work in proof mode. Most commands for the proof mode are so-called *tactics*. Tactics will be summarized in the next section.

Definitions

- *Definition* $x : s := t$ Defines variable x.
- *Inductive* $x : s := x_1 : s_1 \mid \cdots \mid x_n : s_n$ Defines variables x and $x_1, \ldots, x_n$.
- *Fixpoint* ... Variant of *Definition*, inserts a recursive abstraction.

Commands Switching to Proof Mode

- *Definition* $x : s$ Switches to proof mode with the initial goal s.
- *Lemma* $x : s$ Same as *Definition* $x : s$. One may also write *Theorem*, *Corollary*, or *Proposition* for *Lemma*.
- *Goal* s Like *Definition* $x : s$ where variable x is generated by Coq.

Commands Switching Back to Top Mode

- *Defined* Checks whether the term synthesized has the type specified by the initial definition command. If this is the case, the definition is completed and the interpreter switches back to top mode.
- *Qed* Like *Defined* but defined variables is marked as opaque. Defined variables marked opaque do not delta reduce.

Sections

- *Section x* Opens a section x.
- *Variable x : t* Declares a local variable x.
- *End x* Ends section x.

Requests

- *Check t* Elaborates and type checks t.
- *Compute t* Computes normal for of t.
- *Eval cbv in t* Variant of *Compute* preserving names of local variables.
- *Print x* Prints the definition of the defined variable x.
- *About x* Says in which library x is defined.
- *Show Proof* Shows state of proof synthesis. Works only in proof mode.

Notational Commands

- *Notation ...* Defines a notation.
- *Implicit Arguments x ...* Defines implicit arguments for defined function x. Implicit arguments are not shown in regular display mode and are inserted automatically during elaboration.
- *Coercion x ...* Defines a coercion function x. Applications of coercions functions are not shown in regular display mode and are inserted automatically during elaboration to make terms well-typed.

Definition of Tactics

- *Ltac x* $x_1 \cdots x_n :=$ *t* Defines a tactic x taking arguments $x_1, \ldots, x_n$.

A.2 Tactics

Tactics are commands providing for type-directed synthesis of terms. Tactics can only be used in proof mode. In proof mode, there is a *list of goals*. A goal specifies a term still to be synthesized. A goal consists of type assumptions for variables the term may use and of the type of the term. We refer to the type a goal specifies fo the term to be synthesized as *claim*. A tactic always works on the first goal in the list of goals. We refer to the first goal as the *current goal*.

Basic Tactics

- *intros* $x_1 \ldots x_n$ Introduces arguments for functional claims.
- *exact t* Solves goal with term t.
- *refine t* Like *exact* but introduces subgoals for underlines in t that cannot be filled in.

- *apply t* Solves goal with *refine t* || *refine t _* || *refine t _ _* || · · · .
- *generalize t* Rewrites claim s to $u \to s$ where $t : u$.
- *clear x* Deletes assumption x.
- *revert x* Equivalent to *generalize x*; *clear x*.

Conversion Tactics

Apply the conversion rule to the claim. Can be applied to an assumption x by adding "*in x*".

- *hnf* Reduces to head normal form.
- *cbv* Reduces to normal form.
- *change t* Canges claim to t if claim and t are convertible.
- *simpl* [x | t] Simplifies [applications of x | subterm t].
- *unfold x* Delta reduces x, then beta reduces.
- *pattern t* Patterns out subterm t by creating a beta redex st.
- *pattern t at n* Patterns out n-th occurrence of subterm t.

Let Tactics

Synthesize let terms.

- *assert* ($x := t$) Adds assumption $x : u$ where $t : u$.
- *assert* ($x : t$) Adds assumption $x : t$ to current subgoal and adds subgoal for t.
- *cut t* Rewrites claim s to $t \to s$ and adds a subgoal for t.
- *pose* ($x := t$) Adds let with delta reducible x.

Inductive Tactics

- *destruct t* Obtains claim with a match on t; adds a subgoal for each rule of the match.
- *case_eq t* Like *destruct t* but for each rule introduces an assumption $t = u$.
- *fix x n* Obtains claim with a recursive function x decreasing its n-th argument.
- *induction t* Applies elimination lemma of the inductive type of t.
- *induction 1* Expects functional claim with an inductive argument type. Applies elimination lemma for the inductive argument type.
- *constructor* Expects inductive claim and applies a member constructor.

Equational Tactics

- *reflexivty* Solves goal if claim has form $s = s$ up to conversion.
- *rewrite t* Rewrites subterm u of claim to v provided $t : u{=}v$.

- *rewrite<- t* Rewrites subterm v of claim to u provided $t : u{=}v$.
- *f_equal* Strengthens claims $su = sv$ to $u = v$.
- *discriminate t* Proves claim if t is a proof of an equation contradicting constructor disjointness.
- *injection t* Weakens claim by equational premises that follow by constructor injectivity from the equation proved by t.
- *inversion t* Solves goal by contradiction or derives equational assumptions. The type of t must be inductive. Exploits disjointness, injectivity, and exhaustiveness of constructors.

Logical Tactics

- *contradiction t* Solves goal if t is a proof of *False*.
- *absurd t* Replaces claim with subgoals for $\neg t$ and t.
- *split* Splits conjunctive claim.
- *left* Strengthens disjunctive claim to left side.
- *right* Strengthens disjunctive claim to right side.
- *exists t* Strengthens existential claim to witness t.

Automation Tactics

Try to solve goal. Fail if they cannot.

- *assumption* Solves every goal whose claim appears as an assumption.
- *tauto* Solves every goal that can be solved with *intros*, *reflexivity*, the definitions of negation and equivalence, and the introduction and elimination rules for falsity, truth, implication, conjunction, and disjunction.
- *congruence* Solves every goal that can be solved by *intros*, *reflexivity*, and rewriting with unquantified equations.

Tacticals

Compose tactics into more powerful tactics.

- *s*; *t* Applies tactic s, then applies tactic t to every subgoal added by s.
- *s* || *t* Applies tactic s. If application of s fails, tactic t is applied.
- *repeat t* Applies tactic t until it either fails or leaves goal unchanged.
- *try t* Applies tactic t. If t fails, *try t* leaves goal unchanged and succeeds.