Semantics

Lecture Notes WS 2011

February 5, 2012

Gert Smolka
Department of Computer Science
Saarland University

Contents

1 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.

1.1 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
```

1.2 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 1.2.1 (Commutativity of conjunction) Prove $x \wedge y = y \wedge x$ in Coq.

Exercise 1.2.2 (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.

1.3 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 1.3.1 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*.

1.4 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 1.4.1 Prove Lemma *plus_com* by induction on y.

Exercise 1.4.2 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 1.4.3 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*).

1.5 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 1.5.1 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 1.5.2 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

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

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.

1.6 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):

$$\mathit{iter}\ 0\ f\ x = x$$
$$\mathit{iter}\ (Sn)\ f\ x = f(\mathit{iter}\ n\ f\ x)$$

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 1.6.1 Prove $mult\ x\ y = iter\ x\ (plus\ y)\ O$ for all numbers x and y.

Exercise 1.6.2 Define a function *power* recursively (see Exercise 1.3.1) and prove *power x n = iter n (mult x) (S O)* for all $x, n : nat$.

Exercise 1.6.3 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 1.6.4 (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.

1.7 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 1.7.1 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)).
```

1.8 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.

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

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.

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

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 1.8.1 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.
```

1.9 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 1.9.1 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 1.9.2 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 1.9.3 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*.

1.10 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 1.10.1 Define a predecessor function $nat \to option\ nat$.

Exercise 1.10.2 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 1.10.3 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 1.10.4 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.
```

1.11 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.
rewrite IHn. rewrite negb_negb. reflexivity. Qed.
```

Exercise 1.11.1 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 1.11.2 Identify the nested match in *evenb′*.

1.12 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*.

1.13 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 =

2 Case Study: Compiler Correctness

By now we know enough to prove the correctness of a simple compiler translating arithmetic expressions into programs for a stack machine.

Our presentation is inspired by the introductory Coq example of Adam Chlipala's book *Certified Programming with Dependent Types* (see http://adam.chlipala.net/cpdt). Read Chlipala's presentation for a more detailled presentation.

Abstract Syntax

We consider expressions that are obtained from natural numbers and binary operations. We consider two binary operators, addition and multiplication. We define the abstract syntax of expressions as follows.

```
Inductive binop : Type :=
| Plus : binop
| Times : binop.

Inductive exp : Type :=
| Const : nat -> exp
| Binop : binop -> exp -> exp -> exp.
```

Evaluation

We fix the semantics of the syntactic objects with two evaluation functions mapping operators to functions and expressions to numbers.

```
Definition evalBinop (b : binop) : nat -> nat -> nat :=
  match b with
    | Plus => plus
    | Times => mult
  end.

Fixpoint evalExp (e : exp) : nat :=
  match e with
    | Const n => n
    | Binop b e1 e2 => (evalBinop b) (evalExp e1) (evalExp e2)
  end.
```

We test our definitions with the expression $(2 + 3) \cdot 4$.

```
Definition test : exp := Binop Times (Binop Plus (Const 2) (Const 3)) (Const 4).

Compute evalExp test.
% 20
```

Stack Machine

Our stack machine has two instructions.

```
Inductive instr : Type :=
| iConst : nat -> instr
| iBinop : binop -> instr.
```

An instruction *iConst n* puts the number *n* on the stack. An instruction *iBinop o* takes two numbers from the stack, applies the operation designated by the operator *o*, and pushes the result on the stack. We define the semantics of instructions with a function *runInstr* maping an instruction and a stack to a stack option.[1]

```
Definition stack := list nat.

Definition runInstr (i : instr) (s : stack) : option stack :=
  match i with
    | iConst n => Some (n :: s)
    | iBinop b =>
      match s with
        | arg1 :: arg2 :: s' => Some ((evalBinop b) arg1 arg2 :: s')
        | _ => None
      end
  end.
```

The function *runInstr* returns *None* if an operator is applied to a stack that doesn't have at least two elements.

A program is a list of instructions.

```
Definition prog := list instr.
```

The execution of programs is defined as follows.

[1] We assume that the library *list* has been loaded with the command *Require Import List*. The library provides the "::" infix notation for *cons* and the lemmas *app_assoc* and *app_nil_r* we will use in the following.

```
Fixpoint runProg (p : prog) (s : stack) : option stack :=
  match p with
    | nil => Some s
    | i :: p' =>
      match runInstr i s with
        | None => None
        | Some s' => runProg p' s'
      end
  end.
```

Compiler

The compiler translates expressions into programs.

```
Fixpoint compile (e : exp) : prog :=
  match e with
    | Const n => iConst n :: nil
    | Binop b e1 e2 => compile e2 ++ compile e1 ++ iBinop b :: nil
  end.
```

We test the compiler and the stack machine as follows.

```
Compute compile test.
% iConst 4 :: iConst 3 :: iConst 2 :: iBinop Plus :: iBinop Times :: nil

Compute runProg (compile test) nil.
% Some (20 :: nil)
```

Correctness

We now prove the correctness of the compiler. We formulate the correctness of the compiler with an equation for all expressions e and all stacks s.

```
runProg (compile e) s = Some (evalExp e :: s)
```

An attempt to prove this equation by induction on e fails since the inductive hypothesis is not general enough. Thus we prove a more general equation.

```
Lemma compile_correct' e p s:
runProg (compile e ++ p) s = runProg p (evalExp e :: s).

Proof. revert p s. induction e ; intros p s.
simpl. reflexivity.
simpl. rewrite <- app_assoc. rewrite IHe2.
rewrite <- app_assoc. rewrite IHe1. simpl. reflexivity. Qed.
```

Look carefully at the proof. First we use *revert* to obtain an inductive hypothesis that quantifies over p and s. The induction introduces subgoals for the constructors *Const* and *Binop*. The goal for *Binop b e1 e2* comes with inductive assumptions for $e1$ and $e2$. This is our first example of a binary recursion and a corresponding binary induction.

Now the proof of the main result is straightforward.

```
Theorem compile_correct e s :
runProg (compile e) s = Some (evalExp e :: s).

Proof. rewrite <- (app_nil_r (compile e)).
rewrite compile_correct'. simpl. reflexivity. Qed.
```

Exercise 2.0.1 Extend the above development with an operator for subtraction.

Exercise 2.0.2 Write a decompilation function that recovers an expression from the program it compiles to and prove the correctness of your function.

Exercise 2.0.3 Write a function $optimize : exp \to exp$ that simplifies an expression with the rules $0 + e \to e$, $0 \cdot e \to 0$, and $1 \cdot e \to e$. Prove the correctness of your optimizer (i.e., $evalExp(optimize\ e) = optimize\ e$).

Exercise 2.0.4 Extend the expressions with variables. Use the code you have so far and prefix it with

```
Section Compilation.
Variable var : Type.
Variable state : var -> nat.
```

Evaluate variables to the values given by *state*. Extend the machine with an instruction for variables and adapt the compiler and the correctness proof. Observe that the extension to variables is straightforward. Close the section with ***End*** *Compilation*.

3 Propositions and Proofs

In Coq, statements of properties are called propositions. So far we have only seen propositions that are equations. We now move to more complex propositions involving implication, universal quantification, and other logical operations.

3.1 Implication and Universal Quantification

Here is a proposition stating that equality is symmetric.

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

Proof. intros X x y A. rewrite A. reflexivity. Qed
```

The command *Goal* is like the command *Lemma* but leaves it to Coq to choose a name for the proof to be constructed. The tactic *intros* strips off the universal quantification (*forall*) and the implication (→) by representing the assumptions made by these operations as assumptions of the goal.

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

The rest of the proof is straightforward since we have the assumption $A : x = y$, which gives us a proof A of the equation $x = y$. Note that the effect of the *intros* tactic can be undone with the *revert* tactic we have seen before.

Here is a proposition stating the *modus ponens law* for implication.

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

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

The proof first strips off the universal quantification and the outer implications.[1] This brings us to the goal

[1] Implication adds missing parentheses to the right, i.e., $X \to (X \to Y) \to Y$ elaborates to $X \to ((X \to Y) \to Y)$.

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

The proof now finishes by applying the proof A of the implication $X \to Y$ to the proof x of X, which yiels a proof of Y (i.e., the claim). This is done with the tactic *exact*.

Next we show the transitivity of implication.

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

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

Here is a theorem involving a nested universal quantification and two predicates p and q.

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

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

Think of p and q as two properties of numbers. Technically, p and q are functions that yield propositions. Functions that eventually yield propositions are called **predicates**. The proof introduces all internal assumptions and then applies the proof of the quantification to 7 (for x) and to the assumed proof of $p\,7$.

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

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

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

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

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

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

Here is a theorem involving a nested quantification over a predicate.

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

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

Run the proof with Coq to understand. The introductions lead to the the goal

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

Applying the proof A to the predicate $\lambda z.x{=}z$ gives us a proof of the implication $x{=}x \to x{=}y$.[2] We backward apply this proof to the claim, thus arriving at the trivial claim $x{=}x$, which we solve with reflexivity.

Exercise 3.1.1 Prove that equality is transitive.

Exercise 3.1.2 Prove the following goals.

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

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

Exercise 3.1.3 Prove the following goals. Hint: Use the *induction* tactic.

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

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

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

Exercise 3.1.4 Prove the following goal.

```
Goal forall x y y', x + y = x + y' -> y = y'.
```

3.2 Falsity and Negation

Coq comes with a proposition *False* that cannot be proved without assumptions. Given inconsistent assumptions, a proof of *False* may however be possible. There is a basic logic principle called *explosion*, which says that from a proof of *False* one can optain a proof of every proposition. Coq provides the explosion principle through the tactic *contradiction*.

[2] Note that $\lambda z.\, x{=}z$ is mathematical notation for the function that given z yields the equation $x{=}z$. The Coq notation for the function $\lambda z.\, x{=}z$ is *fun z => x=z* (see Section 1.1).

```
Goal False -> 2=3.

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

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

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

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

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

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

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

Exercise 3.2.1 Prove the following goals.

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

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

3.3 Assumption and Auto

Here are two *automation tactics*.

- *assumption* Solves every goal whose claim appears as an assumption.
- *auto* Tries to solve the current goal by applying assumptions with the tactic *apply*. It subsumes reflexivity and assumption.

Run the following examples to get an better idea of the tactics. If you want to know more, consult the Coq documentation.

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

Proof. auto. Qed.

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

Proof. induction xs ; auto. Qed.
```

3.4 Discriminate and Injection

The tactics *discriminate* and *injection* provide basic proof rules for inductive types.

- *discriminate t* Solves current goal if t is a proof of an equation contradicting constructor disjointness.
- *discriminate* Solves current goal if an assumption assumes a proof of an equation contradicting constructor disjointness.
- *injection t* Weakens the claim of the current goal by equational premises that follow by constructor injectivity from the equation proved by t.

```
Goal 3<>5.
Proof. intros A. discriminate A. Qed.

Goal forall n, S n <> n.
Proof. induction n ; intros A.
discriminate A.
injection A. assumption. Qed.
```

Exercise 3.4.1 Prove the following goal.

```
Goal forall x y, andb x y = true -> x = true.
```

3.5 Conjunction, Disjunction, and Equivalence

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

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

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

```
Goal forall X Y : Prop, X \/ Y -> Y \/ X.
Proof. intros X Y A. destruct A as [x|y]. right. exact x. left. exact y. Qed.
```

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

We obtain shorter proofs if we destructure the proofs of conjunctions and disjunctions with the *intros* tactic.

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

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

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

We can also nest destructuring patterns:

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

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

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

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

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

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

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

Exercise 3.5.1 Prove the following goals.

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

Goal False <-> forall Z : Prop, Z.

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

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

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

3.6 Tauto

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

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

Proof. tauto. Qed.
```

3.7 Existential Quantification

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

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

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

Russell's law is a simple fact about nonexistence that has amazing consequences.[3] One such consequence is the undecidability of the halting problem. We state Russell's law as follows:

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

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

The proof of Russell's law is not difficult.

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

Proof. intros [A B]. apply A ;
apply B ; intros x ; exact (A x x). Qed.

Goal Russell.

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

Exercise 3.7.1 Prove the De Morgan Law for existential quantification.

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

[3] Because of its amazing consequences Russell's law is often called Russell's paradox.

Exercise 3.7.2 Prove the exchange rule for existential quantifications.

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

Exercise 3.7.3 Prove the following goal. It shows that existential quantification can be expressed with implication and universal quantification.

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

3.8 Introduction and Elimination Rules

Figure 3.1 summarizes the basic proof rules associated with the logical operations. Each rule says that the proposition appearing below the rule can be proved by proving the propositions appearing above the rule. The rules are written with the following notations:

- $s \Rightarrow t$ says that t can be proved by assuming there is a proof of s.
- $x : s \Rightarrow t$ says that t can be proved by assuming that x is a member of s.
- s_t^x stands for the proposition obtained from s by replacing x with t.

The rules appearing on the left are called **introduction rules**, and the rules appearing on the right are called **elimination rules**. Note that disjunction is special since it has two introduction rules, and that falsity is special in that is has no introduction rule. The introduction rule for a logical operation O tells us how we can prove propositions obtained with O, and the elimination rule tells us how we can make use of a proof of a proposition obtained with O.

Coq realizes the introduction and elimination rules with the following tactics.

	introduction	elimination
$\rightarrow$	intros	exact, apply
forall	intros	exact, apply
False		contradiction
$\wedge$	split	destruct
$\vee$	left, right	destruct
exists	exists	destruct
$=$	reflexivity	rewrite

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

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

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

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

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

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

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

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

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

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

Figure 3.1: Introduction and elimination rules

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

3.9 Propositions as Types Principle

We describe the meaning of the logical operations by relating them to proofs:

- A proof of $s \to t$ is a function that for every proof of s yields a proof of t.
- A proof of $\forall x : s.t$ is a function that for every $x : s$ yields a proof of t.
- There is no proof of $\bot$.
- A proof of $s \wedge t$ consists of a proof of s and a proof of t.
- A proof of $s \vee t$ is either a proof of s or a proof of t.
- A proof of $\exists x : s.t$ consists of a value $u : s$ and a proof of t_u^x.
- A proof of $s = t$ only exists if s and t are equal.

This proof-theoretic semantics explains why we can apply proofs of implications and universal quantifications as first done in Section 3.1. It also validates the introduction and elimination rules shown Figure 3.1. Finally, the proof-theoretic semantics motivates a fundamental design principle underlying Coq known as *propositions as types principle*. Following this principle, propositions are types inhabiting the universe *Prop*, and the proofs of a proposition are the members of the proposition. This explains why we write $x : X$ to say that x is a proof of X. Implications and universal quantifications are obtained as function types, so their proofs are in fact functions as required by the proof-theoretic semantics. The remaining propositions are obtained as inductive types, as we will see in the next chapter.

Consider the following commands.

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

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

The commands bind the identifier *modus_ponens* to a function whose type is the proposition stated as lemma.

```
Print modus_ponens.
% modus_ponens =
% fun (X Y : Prop) (x : X) (A : X -> Y) => A x
% : forall X Y : Prop, X -> (X -> Y) -> Y
```

So we learn that a lemma is actually a proof of the proposition stated by the lemma. If the proposition is a universal quantification or an implication, the lemma is a function that can be applied to arguments. Thus applications of lemmas is functional application.

We may use the *exact* tactic with more complex proofs.

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

Proof. exact (fun X Y x A => A x). Qed
```

The missing types in the proof term are automatically inferred by Coq.

Exercise 3.9.1 Give proofs of the following propositions as terms. Check your results with Goal *prop*. Proof. exact *proof*. Qed.

a) $\forall X\ Y\ Z.\ (X \to Y) \to (Y \to Z) \to X \to Z$

b) $\forall X\ Y.\ ((X \to X) \to Y) \to Y$

3.10 Excluded Middle

In Mathematics, one assumes that every proposition is either false or true. Consequently, if X is a proposition, the proposition $X \vee \neg X$ must be true. The

assumption that $X \vee \neg X$ is true for every proposition X is known as *principle of excluded middle*, XM for short.

XM is not validated by the proof-theoretic semantics. In fact, we cannot prove the proposition

```
Definition XM : Prop := forall X : Prop, X \/ ~X
```

in Coq. Neither can we prove the proposition $\neg XM$ in Coq.

Since we cannot prove $\neg XM$ in Coq, it is consistent to assume a proof of XM. Given the assumption that XM has a proof, we can prove propositions like $X \vee \neg X$ and $\neg\neg X \to X$.

A logic based on proof-theoretic semantics not assuming a proof of XM is called **intuitionistic**, while a logic that can prove XM without assumptions is called **classical**. An intuitionistic logic that cannot prove $\neg XM$ is more general than a classical logic since one may or may not assume a proof of XM.

Intuitionistic logics have the important property that every function definable in the logic is computable. This property does not hold for classical logic.

Exercise 3.10.1 Prove that the following propositions are equivalent. There are short proofs if you use *tauto*.

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

4 Inductive Propositions

Following the propositions as types principle, Coq provides for the definition of inductive propositions. Inductive propositions are defined in the same way inductive types are defined. We will see inductive definitions realizing a new form of argument dependency.

4.1 Logical Operations as Inductive Predicates

Coq defines the propositions *True* and *False* inductively.

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

Inductive False : Prop := .
```

The definitions introduce the propositions together with their proofs. Since *False* is defined without a proof constructor, it has no proof. Thus, given a proof of *False*, we can obtain a proof of every proposition.

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

Proof. intros X A. destruct A. Qed.
```

Here is a definition of an **inductive predicate** *And* that gives us a family of **inductive propositions** that behave like conjunctions.

```
Inductive And (X Y : Prop) : Prop :=
| AndI : X -> Y -> And X Y.

Goal forall X Y, And X Y <-> X /\ Y.

Proof. split.
intros [x y]. split ; assumption.
intros [x y]. constructor ; assumption. Qed.
```

We refer to the members of an inductive proposition (e.g., *And True True*) as **proofs** and to the value constructors of inductive predicates (e.g., *AndI*) as **proof constructors**.

Exercise 4.1.1 Rewrite the above proof using the tactics *exact* and *apply* in place of *assumption* and *constructor*.

Exercise 4.1.2 Define a family of inductive propositions that behave like disjunctions and prove that your disjunctions are equivalent to Coq's predefined disjunctions.

Exercise 4.1.3 Define a family of inductive propositions that behave like existential quantifications. Arrange your definition such that you obtain an inductive predicate *Ex* for which you can prove

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

Finally, we define an inductive predicate that behaves like equality.

```
Inductive Eq (X : Type) : X -> X -> Prop :=
| EqI : forall x, Eq X x x.

Goal forall (X : Type) (x y : X),
Eq X x y <-> x=y.

Proof. split ; intros A.
destruct A. reflexivity.
rewrite A. constructor. Qed.
```

Exercise 4.1.4 Prove the following goal.

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

Exercise 4.1.5 Prove that *True* has exactly one proof.

```
Goal forall x y : True, x=y.
```

Exercise 4.1.6 Prove that there is a proposition that has exactly one proof.

```
Goal exists X : Prop, X /\ forall x y : X, x=y.
```

Exercise 4.1.7 With universal quantification one can write a proposition *True′* that is equivalent to the inductive proposition *True*.

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

Goal True <-> True'.
```

Prove the Goal. Remark: *True* and *True′* are equivalent but not equal. For instance, one can prove that *True* has exactly one proof, but this is impossible for the proposition*True′*.

4.2 Induction on Proofs of Inductive Propositions

We define an inductive predicate *even* that expresses evenness of numbers.

```
Inductive even : nat -> Prop :=
| evenO : even 0
| evenS : forall n, even n -> even (S (S n)).
```

We may express this definition informally with two inference rules.

$$\frac{}{even\ 0} \qquad\qquad \frac{even\ n}{even\ (S(S\ 0))}$$

We prove a few properties of *evenness*.

```
Goal even 4.
```

```
Proof. constructor. constructor. constructor. Qed.
```

```
Goal forall n, even n -> even (4+n).
```

```
Proof. simpl. intros. repeat constructor. assumption. Qed.
```

```
Goal ~even 1.
```

```
Proof. intros A. inversion A. Qed.
```

```
Goal forall n, even (S (S n)) -> even n.
```

```
Proof. intros n A. inversion A. assumption. Qed.
```

Note the use of the tactic *inversion*. This tactic subsumes the features of the tactics *destruct*, *discriminate*, and *injection*. It exploits the fact that every proof of an inductive proposition must be obtained with a proof constructor of the underlying inductive predicate.

Since the proof constructor *evenS* is recursive, some properties of evenness can only be shown with induction. We use induction on the proofs of inductive propositions obtained with the inductive predicate *even* (or shorter: induction on *even*).

```
Goal forall n, even n -> even (S n) -> False.
```

```
Proof. intros n A B. induction A.
now inversion B.
inversion B. auto. Qed.
```

Note the use of the tactical *now* and the automation tactic *auto*.

- The tactical *now* says that the tactic given as its argument must solve the current goal. The use of *now* makes proof scripts more readable. You may have noticed that we print all-or-nothing tactics in red (i.e., tactics that fail if they do not solve the current goal, (e.g., *reflexivity* and *assumption*)).

- The automation tactic *auto* tries to solve the current goal by applying the implications (possibly universally quantified) appearing as assumptions.

Exercise 4.2.1 Prove the following goals.

```
Goal forall n, even n -> even (pred (pred n)).
Goal forall m n, even m -> even n -> even (m+n).
Goal forall m n, even (m+n) -> even m -> even n.
Goal forall n, even n -> n = 0 \/ exists n' n = S (S n').
```

Exercise 4.2.2 Prove the following goals.

```
Goal forall n, even n -> ~ even (S n).
Goal forall n, even n \/ even (S n).
Goal forall n, ~even n -> even (S n).
Goal forall n, ~even (S n) -> even n.
```

Hint: Use the proof of the second goal to prove the third and fourth goal. There seem to be no direct proofs.

Exercise 4.2.3 One can define evenness with a boolean function.

```
Fixpoint evenb (n : nat) : bool :=
match n with
| 0 => true
| 1 => false
| S (S n') => evenb n'
end.
```

Prove that the boolean and the inductive definition agree. The proof goes through if you generalize the claim as follows.

```
Goal forall n,
(evenb n = true <-> even n) /\
(evenb (S n) = true <-> even (S n)).
```

Exercise 4.2.4 Define an inductive predicate $odd : nat \to Prop$ and prove the equivalence $odd\, n \leftrightarrow even(S\, n)$ for all $n : nat$.

Exercise 4.2.5 There are many possible definitions of an inductive proposition that is equivalent to *False*. Here is one with a recursive proof constructor.

```
Inductive XFalse : Prop :=
| XFalseI : XFalse -> XFalse.
```

Prove the following goal.

```
Goal XFalse -> False.
```

4.3 Parameters and Proper Arguments

If we look at the definition

```
Inductive Eq (X : Type) : X -> X -> Prop :=
| EqI : forall x, Eq X x x.
```

of the inductive predicate *Eq*, we notice that the arguments of *Eq* come in two forms: As **parameters** (the first argument *X* of *Eq*), and as **proper arguments** (the second and third argument of *Eq*). The distinction between parameters and proper arguments of inductive predicates (and type constructors in general) is of great importance. For the inductive predicates defined so far we have the following:

- *And* has 2 parameters and no proper agument.
- *Eq* has 1 parameter and 2 proper aguments.
- *even* has no parameter and 1 proper agument.

The important thing to know about proper arguments is that the tactics *destruct* and *induction* for case analysis and induction only work as expected if the proper arguments of the accompanying inductive propositions are distinct variables. So *induction A* where *A* : *even n* is fine, but *destruct A* where *A* : *even 1* is not. We can use the tactic *remember* to unfold terms that appear as proper arguments:

```
Goal ~ even 1.

Proof. intros A. remember 1 as n.
destruct A. discriminate. discriminate. Qed.

Goal ~ Eq bool true false.

Proof. intros A. remember true as x. remember false as y.
destruct A. congruence. Qed.
```

Note the use of the tactic *congruence*, which combines the abilities of *discriminate* and *injection* with basic equational reasoning. Both goals have shorter proofs if we use the powerful inversion tactic ("*intros A. inversion A.*" does it for both goals).

An argument of an inductive predicate can be introduced as a parameter if in principle it could be introduced before the inductive definition. For the equality predicate, we can give another inductive definition that has only one proper argument:

```
Inductive EQ (X : Type)(x : X) : X -> Prop :=
| EQI : EQ X x x.
```

It is preferable to have as few proper arguments as possible since this will ease proofs (explained in Section 4.6). Note that the predicates *even* and *EQ* cannot

be defined without a proper argument. Proper arguments provide a form of dependency that is not available with parameters. The predicate *Eq* was the first inductive type constructor with a proper argument we have seen.

Exercise 4.3.1 Give two proofs of each of the the following goal, one with *inversion*, and one without *inversion*.

```
Goal forall n, even (S (S n)) -> even n.
```

```
Goal forall n, even (S n) -> exists n', n = S n'.
```

4.4 Natural Order

With the command

```
Locate "<=".
```

we find out that Coq realizes the order "≤" on *nat* with the predicate *le*. With the command

```
Print le.
```

we find out that Coq defines *le* inductively as follows.

```
Inductive le (x : nat) : nat -> Prop :=
| lex : le x x
| leS : forall y, le x y -> le x (S y).
```

With can depict this definition with 2 inference rules.

$$\frac{}{x \leq x} \qquad \frac{x \leq y}{x \leq S\,y}$$

Note that *le* has one parameter and one proper argument. We prove two properties of the predicate *le*.

```
Lemma le_transitive x y z :
le x y -> le y z -> le x z.
```

```
Proof. intros A B. induction B.
assumption. constructor. assumption. Qed.
```

```
Lemma le_Sleft x y : le (S x) (S y) -> le x y.
```

```
Proof. intros A. inversion A. now constructor.
apply le_transitive with (y := S x). repeat constructor. assumption. Qed.
```

Note the *with* clause appearing with the apply tactic. It says that *apply* should instantiate the variable y of the lemma *le_transitive* as $S\,x$.

Exercise 4.4.1 Prove the following claims. Do not use *omega*.

```
Lemma le_Sright x y : le x y -> le (S x) (S y).

Lemma le_O y : le 0 y.

Goal forall x, le x 0 -> x = 0.

Goal forall x, ~ le (S x) x.

Goal forall x y, le x y \/ le y x.
```

4.5 Omega

There is an automation tactic *omega* that can prove many properties of *le* automatically. To use *omega*, we must first load the module *Omega*.

```
Require Import Omega.

Goal forall x y, x <= y \/ y <= x.

Proof. intros x y. omega. Qed.
```

We can use *omega* to prove that *le* agrees with the boolean definition of the natural order appearing in Section 1.3.

```
Goal forall x y, leb x y = true <-> le x y.

Proof. induction x.
split ; intros A. omega. reflexivity.
destruct y ; simpl ; split ; intros A.
  discriminate. exfalso. omega.
  apply IHx in A. omega. apply IHx. omega. Qed.
```

The proof is worth studying.

- The tactic *exfalso* is used to prepare the appplication of *omega*. It replaces the claim by *False*. The tactic *exfalso* realizes the explosion principle and is useful when the assumptions of a goal are inconsistent (i.e., *true* = *fase*).
- The tactic *apply* can be used with equivalences. Moreover, *apply* can apply an implication to an assumption of a goal if the assumption is specified with the keyword *in*.

The module *Omega* also strengthens the automation tactic *firstorder*. This provides us with a shorter proof of the above goal.

```
Goal forall x y, leb x y = true <-> le x y.

Proof. induction x ; destruct y ; firstorder. discriminate. Qed.
```

4.6 Induction Principles

Performing an induction with the tactic *induction* is a complex affair comprising three main steps:

1. Move assumptions of the goal to the claim so that the induction principle of the relevant type constructor becomes applicable (can be done with *revert*).
2. Apply the induction principle.
3. For each subgoal obtained, do introduction steps so that the claim of the subgoal corresponds to the initial claim.

Performing a case analysis with *destruct* is like performing an induction with *induction* except that the destructuring principle of the type constructor is applied in place of the induction principle. The destructuring principle can be obtained from the induction principle by omitting the inductive hypotheses, and vice versa the induction principle can be obtained from the destructuring principle by adding the inductive hypotheses.

Induction and destructuring principles are valid proof principles that can be expressed as propositions. Let us start with the induction principle for the type constructor *nat*:

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

First look at the last line: The principle tells us how to prove claims of the form $\forall n : nat. pn$ where pn can represent any proposition. We then have a premise for every constructor, which provides for the case analysis. The idea is that n must be obtained with one of the constructors and hence it suffices to show p for each constructor. For the recursive constructor S we get an inductive hypothesis pn since n is smaller than Sn. The premises for the constructors are easily obtained from the types of the constructors:

```
O : nat
S : nat -> nat
```

For the inductive predicate *even* we obtain the following induction principle.

```
forall p : nat -> Prop,
p O ->
(forall n, even n -> p n -> p (S (S n))) ->
forall n, even n -> p n
```

This time the conclusion quantifies over the proper argument of the inductive predicate *even*. The premises again are derived from the types of the constructors:

```
evenO : even O
evenS : forall n, even n -> even (S (S n))
```

Next we look at the induction principle for the inductive predicate *le*.

```
forall (x : nat) (p : nat -> Prop),
p x ->
(forall y, le x y -> py -> p (S y)) ->
forall y, le x y -> p y
```

Once more the conclusion quantifies over the proper argument of the inductive predicate. Since x is a parameter it is quantified at the outside of the induction principle. The premises for the constructors are derived from the types of the constructors

```
lex : forall x, le x x
leS : forall x y, le x y -> le x (S y)
```

and acknowledge that fact that the parameter x is quantified outside.

The reasons for two rules we have stated before now become apparent:

1. In the conclusion of the induction principle the proper arguments appear as pairwise distinct variables. Hence the need for *remember*.
2. More proper arguments mean more quantifications in the conclusion of the induction principle. Since every assumption containing a proper argument variable must be moved to the claim, more proper arguments lead to more complex induction hypotheses. Hence the advice to treat arguments of inductive predicates as parameters rather than proper arguments whenever this is possible.

For every defined inductive type constructor C Coq establishes the corresponding induction principle under the name *C_ind*. You can display the induction principle with the command *Check C_ind*. For instance:

```
Check le_ind.
% forall (x : nat) (P : nat -> Prop),
% P x ->
% (forall y : nat, le x y -> P y -> P (S y)) ->
% forall n : nat, le x n -> P n
```

With *Print C_ind* you can find out how Coq proves the induction principles (with *match* and *fix* so that inductive proofs appear as recursive proofs with case analysis; we will not discuss this further).

We now prove the lemma *le_transitive* from Section 4.4 by simulating the induction tactic as described with *revert*, *apply*, and *intros*.

```
Goal forall x y z, le x y -> le y z -> le x z.
```

```
Proof. intros x y z A B.
revert z B.   (* move assumptions to claim *)
apply (le_ind y (le x)).   (* apply induction principle for le *)
assumption.   (* prove subgoal for constructor lex *)
intros z B IHB.   (* introduce previous assumptions and inductive hypothesis *)
constructor. assumption. Qed.   (* prove subgoal for constructor leS *)
```

Step carefully through the proof and make sure you understand every detail. Note that the predicate P of *le_ind* is instantiated with $le\,x$ and that the parameter x is instantiated with y.

Exercise 4.6.1 Give the induction principles for the following inductive predicates defined in this chapter and check you results with Coq.

a) *And*

b) *Eq*

c) *EQ*

Exercise 4.6.2 Prove the following goal by applying the induction principle *even_ind*. Do not use the induction tactic.

```
Goal forall n, even n -> even (S n) -> False.
```

Exercise 4.6.3 Consider the following definition.

```
Inductive le2 : nat -> nat -> Prop :=
| le2x : forall x, le2 x x
| le2S : forall x y, le2 x y -> le2 x (S y).
```

a) Give the induction principle for *le2*.

b) Prove the transitivity of *le2*.

4.7 Relational Semantics

In Chapter 2 we characterized the semantics of expressions with a recursive function $evalExp : exp \to nat$. One speaks of a *denotational semantics*. It is also possible to characterize the semantics of expressions with an inductively defined relation $e \Downarrow n$. One then speaks of a *relational semantics*.

$$\frac{}{n \Downarrow n} \qquad \frac{e_1 \Downarrow n_1 \qquad e_2 \Downarrow n_2}{e_1 \circ e_2 \Downarrow n} \quad n = (evalBinop\,\circ)\,n_1\,n_2$$

```
Inductive relExp : exp -> nat -> Prop :=
| relExpC : forall n, relExp (Const n) n
| relExpB : forall b e1 e2 n1 n2,
              relExp e1 n1 -> relExp e2 n2 ->
              relExp (Binop b e1 e2) (evalBinop b n1 n2).

Goal forall e, relExp e (evalExp e).

Proof. induction e ; simpl ; constructor ; assumption. Qed.
```

Exercise 4.7.1 Prove the equivalence of the denotational and operational semantics of expressions.

```
Goal forall e n, relExp e n <-> evalExp e = n.
```

Exercise 4.7.2 Give the induction principle for *relExp*. Explain why the premise for the constructor *relExpB* contains two inductive hypotheses.

4.8 Reflexive Transitive Closure

Sets and relations can be represented in constructive type theory as follows:

1. A set A of members of a type X can be represented as a predicate $p : X \to Prop$. Then we have $x \in X$ if and only if px is provable.
2. A binary relation R on the members of a type X can be represented as a predicate $p : X \to X \to Prop$. Then we have Rxy if and only if pxy is provable.

A given set can be represented by different predicates. To acknowledge this fact, one says that predicates are **intentional** representations of sets. One also says that the set represented by a predicate is the **extention** of a predicate. Finally, one says that sets are **extentional** since two sets that have the same elements are identical. The same speak applies to relations and predicates.

We will now formalize the notion of a reflexive transitive closure of a relation in constructive type theory. We will use the speak of set theory also we work with intentional representations of relations.

Given a relation $R : X \to X \to Prop$, we can define its reflexive transitive closure R^* inductively as follows:

$$\frac{}{R^*xx} \qquad\qquad \frac{Rxy \qquad R^*yz}{R^*xz}$$

For the formal definition, we use Coq's section device, which makes it possible to keep the parameters X and R implicit.

```
Section Star.
Variable X : Type.
Variable R : X -> X -> Prop.

Inductive star : X -> X -> Prop :=
| starR : forall x, star x x
| starT : forall x y z, R x y -> star y z -> star x z.
```

Within the section, *star* represents R^*. We prove that *star* is transitive.

```
Goal forall x y z, star x y -> star y z -> star x z.

Proof. intros x y z A B. induction A. assumption.
apply starT with (y:=y) ; auto. Qed.
```

Exercise 4.8.1 Prove the following goal. Do not use a lemma.

```
Goal forall x y z, star x y -> R y z -> star x z.
```

We now close the section.

```
End Star.
```

This discharges the assumptions X and R by making them parameters of the defined objects. For instance:

```
Check star.
% star : forall X : Type, (X -> X -> Prop) -> X -> X -> Prop
```

We make X an implicit argument of *star*.

```
Implicit Arguments star [X].
```

We can now write *star R* for R^*.

Exercise 4.8.2 Give the induction principle for the inductive predicate *star*.

Exercise 4.8.3 Prove that taking the reflexive transitive closure preserves invariants.

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

Goal forall (X : Type) (R : X -> X -> Prop) (p : X -> Prop),
invariant p R -> invariant p (star R).
```

Exercise 4.8.4 We can define a reflexive transitive closure predicate *star1* with a single proper argument.

```
Inductive star1 (x : X) : X -> Prop :=
| star1R : star1 x x
| star1T : forall y z, star1 x y -> R y z -> star1 x z.
```

a) Give the induction principle for *star1*.

b) Prove that *star1* is reflexive and transitive.

c) Prove $\forall x\, y.\ star\, x\, y \leftrightarrow star1\, x\, y$

Exercise 4.8.5 You may have seen $R^* := \bigcup_{n \in \mathbb{N}} R^n$ as a definition of the reflexive transitive closure. Using the function *iter*, we can express this definition in Coq.

```
Definition comp {X : Type} (R S : X -> X -> Prop) (x z : X) : Prop :=
exists y, R x y /\ S y z.

Definition stari {X : Type} (R : X -> X -> Prop) (x y : X) :=
exists n, iter n (comp R) (fun x y => x=y) x y.
```

Prove the equivalence of the inductive and the iterative definition.

```
Goal forall (X : Type) (R : X -> X-> Prop) (x y : X),
star R x y <-> stari R x y.
```

5 Confluence and Termination

The small step-semantics of computational systems yields binary relations known as reductions. In this chapter we consider two important properties of such reductions known as confluence and termination. For informal explanations and a full mathematical treatment see Baader and Nipkow's textbook (Chapter 2).

5.1 Basic Definitions

We think of a binary relation on a set X as a directed graph whose nodes are the elements of X and that has an edge from x to y if (x, y) is a pair of the relation.

```
Section Relation.
Variable X : Type.

Definition rel : Type := X -> X -> Prop.

Definition reflexive (r : rel) : Prop :=
forall x, r x x.

Definition symmetric (r : rel) : Prop :=
forall x y, r x y -> r y x.

Definition transitive (r : rel) : Prop :=
forall x y z, r x y -> r y z -> r x z.

Definition functional (r : rel) : Prop :=
forall x y z, r x y -> r x z -> y=z.

Definition reducible (r : rel) (x : X) : Prop :=
exists y, r x y.

Definition normal (r : rel) (x : X) : Prop :=
~ reducible r x.

Definition total (r : rel) : Prop :=
forall x, reducible r x.
```

The members of the type *rel* are binary predicates representing relations. The relation represented by a predicate $r : rel$ is the set of all pairs (x, y) such that rxy is provable. Different predicates may represent the same relation. This motivates the following definitions.

```
Definition rle (r r' : rel) : Prop :=
forall x y, r x y -> r' x y.

Definition req (r r' : rel) : Prop :=
rle r r' /\ rle r' r.
```

In general it is undecidable for a relation whether a node is reducible or normal (take the big-step relations of Imp commands). We call this property **reduction decidability**.

```
Definition reddec (r : rel) : Prop :=
forall x, reducible r x \/ normal r x.
```

Here are operators that yield the composition and the inverse of relations.

```
Definition comp (r s : rel) : rel :=
fun x z => exists y, r x y /\ s y z.

Definition inverse (r : rel) : rel :=
fun x y => r y x.
```

Exercise 5.1.1 Prove the following statements:

a) Composition preserves functionality of relations. That is, the composition of two functional relations is always functional.

b) Composition preserves reflexivity of relations.

c) Composition preserves totality of relations.

d) Composition does not preserve symmetry of relations. That is, there are two symmetric relations whose composition is not symmetric.

Exercise 5.1.2 Prove the following goals.

```
forall r, symmetric r <-> rle (inverse r) r.

forall r, transitive r <-> rle (comp r r) r.
```

Exercise 5.1.3 Consider the predefined predicate $gt : nat \to nat \to Prop$ (">") and prove the following.

a) gt is reduction decidable.

b) 0 is normal and every other number is reducible.

5.2 Star Operator

In the graph metaphor, the reflexive transitive closure r^* of a relation r is the reachability relation of the graph r. We call the operator mapping a relation to its reflexive transitive closure **star operator**.

```
Inductive star (r : rel) : rel :=
| starR z : star r z z
| starS x y z : r x y -> star r y z -> star r x z.
```

The definition has the drawback that *star* has two proper arguments. If we reverse the order of the last two arguments we can give a definition with only one proper argument. We don't do this since we do not want to disagree with the usual mathematical notation. We formulate and prove the most important properties of the star operator.

```
Lemma star_reflexive r :
reflexive (star r).

Proof. intros x. constructor. Qed.

Lemma star_transitive r :
transitive (star r).

Proof. intros x y z A B. induction A ; eauto using star. Qed.

Lemma star_expansive r :
rle r (star r).

Proof. hnf. eauto using star. Qed.

Lemma star_right r x y z :
star r x y -> r y z -> star r x z.

Proof. intros A B. induction A ; eauto using star. Qed.

Lemma star_least r s :
reflexive s -> transitive s -> rle r s -> rle (star r) s.

Proof. intros R T A x y B. induction B ; eauto. Qed.

Lemma star_monotone r r' :
rle r r' -> rle (star r) (star r').

Proof. intros A x y B. induction B ; eauto using star. Qed.

Lemma star_symmetry r :
symmetric r -> symmetric (star r).

Proof. intros A x y B.
induction B ; eauto using star, star_right. Qed.

Lemma star_idempotent r :
req (star (star r)) (star r).

Proof. split.
apply star_least. apply star_reflexive. apply star_transitive. hnf. trivial.
apply star_expansive. Qed.
```

Exercise 5.2.1 Prove the following lemma.

```
Lemma star_normal r x y :
normal r x -> star r x y -> x = y.
```

Exercise 5.2.2 We define operators taking the **reflexive closure** and the **transitive closure** of a relation.

```
Definition ref (r : rel) : rel :=
fun x y => x=y \/ r x y.

Inductive plus (r : rel) : rel :=
| plusO x y : r x y -> plus r x y
| plusS x x' y : r x x' -> plus r x' y -> plus r x y.
```

a) Prove that *ref r* is reflexive.

b) Prove that *plus r* is transitive.

c) Prove $req\ (star\ r)\ (ref(plus\ r))$.

Exercise 5.2.3 Prove that the predicate *le* is equivalent to $star\ (\lambda xy.Sx{=}y)$.

5.3 Convertibility and Joinability

Two nodes are **convertible** if they are connected through edges ignoring the direction of the edges.

```
Definition sym (r : rel) : rel :=
fun x y => r x y \/ r y x.

Definition convertible (r : rel) (x y : X) : Prop :=
star (sym r) x y.
```

Two nodes are **joinable** if they can reach a common node.

```
Definition joinable (r : rel) (x y : X) : Prop :=
exists z, star r x z /\ star r y z.
```

Joinability implies convertibility.

```
Lemma star_convertible r :
rle (star r) (convertible r).
```

```
Proof. apply star_monotone ; cbv ; auto. Qed.
```

```
Lemma convertible_symmetric r :
symmetric (convertible r).
```

```
Proof. apply star_symmetry. firstorder. Qed.
```

```
Lemma joinable_convertible r :
rle (joinable r) (convertible r).
```

```
Proof. intros x y J.  destruct J as [z [A B]].
apply star_convertible in A.
apply star_convertible, convertible_symmetric in B.
eapply star_transitive ; eassumption. Qed.
```

For every relation r we have the following chain of extensions.

$$r \preceq star\ r \preceq joinable\ r \preceq convertible\ r = star(sym\ r)$$

Reductions for which convertibility agrees with joinability are called **Church-Rosser**.

```
Definition Church_Rosser (r : rel) : Prop :=
rle  (convertible r) (joinable r).
```

Exercise 5.3.1 Show that convertibility is an equivalence relation.

Exercise 5.3.2 Prove the following lemma.

```
Lemma star_joinable r :  rle (star r) (joinable r).
```

5.4 Confluence

Confluence is a property that is equivalent to Church-Rosser.

```
Definition confluent (r : rel) :  Prop :=
forall  x y z,  star r  x y −> star r x z −> joinable r y z.
```

```
Lemma Church_Rosser_confluent r :
Church_Rosser r −> confluent r.
```

```
Proof. intros A x y z B C. apply A.
apply star_transitive with (y:=x).
apply star_convertible, star_symmetry in B ; firstorder.
apply star_convertible in C ; assumption. Qed.
```

For the proof of the other direction, we define **semi-confluence**.

```
Definition semi_confluent (r : rel) : Prop :=
forall  x y z,  r  x y −> star r x z −> joinable r y z.
```

```
Goal forall r,  confluent r −> semi_confluent r.
```

```
Proof. firstorder using star_expansive. Qed.
```

```
Lemma semi_confluent_Church_Rosser r :
semi_confluent r −> Church_Rosser r.
```

The lemma has a nice graphical proof (see Baader-Nipkow). The formal proof follows the graphical proof.

```
Proof. intros A x y B.
induction B. now exists x ; split ; constructor.
destruct IHB as [u [B1 B2]].
destruct H. now exists u ; eauto using star.
assert (C : joinable r x u) by eauto.
destruct C as [v [C1 C2]].
exists v ; firstorder using star_transitive. Qed.
```

We now know that semi-confluence, confluence, and Church-Rosser are pairwise equivalent properties.

Exercise 5.4.1 Prove that *joinable r* is reflexive and symmetric relation.

Exercise 5.4.2 Prove that a relation is confluent if and only if its joinability relation is transitive.

5.5 Strong Confluence

Strong confluence is a sufficient condition for confluence.

```
Definition strongly_confluent (r : rel) : Prop :=
forall x y z, r x y -> r x z ->
exists u, star r y u /\ ref r z u.

Lemma strong_confluence r :
strongly_confluent r -> semi_confluent r.
```

There is a nice graphical proof and the formal proof follows the graphical proof.

```
Proof. intros S x y z A B.
revert y A ; induction B as [x | x z' z] ; intros y A.
now exists y ; eauto using star.
destruct (S x y z' A H) as [u [S1 S2]].
destruct S2 ; subst.
now exists z ; firstorder using star, star_transitive.
assert (C : joinable r u z) by auto.
destruct C as [v [C1 C2]].
exists v ; firstorder using star_transitive. Qed.
```

Exercise 5.5.1 Give a finite relation that is confluent but not strongly confluent.

Exercise 5.5.2 Prove that functional relations are strongly confluent.

Exercise 5.5.3 The **diamond property** is defined as follows:

```
Definition diamond (r : rel) : Prop :=
forall x y z, r x y -> r x z ->
exists u, r y u /\ r z u.
```

a) Prove that r is confluent if and only if *star r* satisfies the diamond property.

b) Prove that relations satisfying the diamond property are strongly confluent.

c) Prove that *star* preserves the diamond property.

5.6 Normal Forms and Normalization

A normal node y is a **normal form** of a node x if y is reachable from x.

```
Definition normal_form (r : rel) : rel :=
fun x y => star r x y /\ normal r y.
```

For confluent reductions, a node has at most one normal form.

```
Lemma confluent_functional_normal_form r :
confluent r -> functional (normal_form r).
```

```
Proof. intros A x y z [B1 B2] [C1 C2].
assert (D : joinable r y z) by eauto.
destruct D as [u [D1 D2]].
assert (y = u) by eauto using star_normal.
assert (z = u) by eauto using star_normal.
congruence. Qed.
```

A node is **normalizing** if it has a normal form. We employ an inductive definition.

```
Inductive normalizes (r : rel) : X -> Prop :=
| normalizesI1 x : normal r x -> normalizes r x
| normalizesI2 x y : r x y -> normalizes r y -> normalizes r x.
```

```
Lemma normalizes_normal_form r x :
normalizes r x <-> reducible (normal_form r) x.
```

```
Proof. split.
intros A. induction A.
   exists x. now firstorder using star_reflexive.
   destruct IHA as [z B]. exists z. now firstorder using star.
intros [y [A B]]. induction A ; eauto using normalizes. Qed.
```

A relation is **normalizing** if every node has a normal form.

```
Definition normalizing (r : rel) : Prop :=
forall x, normalizes r x.
```

```
Goal forall r,
normalizing r <-> total (normal_form r).
```

```
Proof. split ; intros A x ; apply normalizes_normal_form, A. Qed.
```

For a normalizing and confluent relation, two nodes are convertible if and only if they have a common normal form.

```
Lemma normalizing_confluent r x y :
normalizing r -> confluent r ->
(convertible r x y <-> exists z, normal_form r x z /\ normal_form r y z).

Proof. intros N C. split.
intros A.
  assert (B: joinable r x y)
  by (apply semi_confluent_Church_Rosser ;
      firstorder using star_expansive).
  destruct B as [u [B1 B2]].
  assert (D : reducible (normal_form r) u)
  by (apply normalizes_normal_form ; trivial).
  destruct D as [v [D1 D2]]. exists v.
  now firstorder using star_transitive.
intros [z [[A _] [B _ ]]].
  apply joinable_convertible. firstorder. Qed.
```

Exercise 5.6.1 Consider the predicate $gt : nat \to nat \to Prop$ representing the natural order ">".

- Prove that gt is normalizing.
- Prove that 0 is the only normal form of gt.
- Prove that gt is confluent.

5.7 Semantic Confluence

In practice, many reductions respect a semantic equality. In this case a normalizing reduction is confluent if different normal forms are semantically different.

```
Section Semantic_Confluence.
Variable X V : Type.
Variable eval : X -> V.
```

Two elements of the syntactic domain X are semantically equal if they evaluate to the same value of the semantic domain V. A relation on X is sound if it respects the semantic equality.

```
Definition sound (r : rel X) : Prop :=
forall x y, r x y -> eval x = eval y.
```

A relation on X is complete if different normal forms are semantically different.

```
Definition complete (r : rel X) : Prop :=
forall x y, normal r x -> normal r y ->
eval x = eval y -> x = y.
```

Reflexive transitive closure preserves soundness.

```
Lemma star_sound (r : rel X) :
sound r -> sound (star r).

Proof. intros S x y A. induction A. reflexivity.
apply S in H. congruence. Qed.
```

We show that normalizing relations are confluent if they are sound and complete.

```
Lemma semantic_confluence (r : rel X) :
sound r -> complete r ->
normalizing r -> confluent r.

Proof. intros S C N x y z A B.
assert (Ny := N y). apply normalizes_normal_form in Ny.
destruct Ny as [u [Nu Ny]].
assert (Nz := N z). apply normalizes_normal_form in Nz.
destruct Nz as [v [Nv Nz]].
cut (u=v). now intros e ; subst v ; exists u ; auto.
apply C ; auto.
apply star_sound in S.
apply S in A. apply S in B. apply S in Nu. apply S in Nv.
congruence. Qed.
```

Exercise 5.7.1 We consider arithmetic expressions

$$e ::= 0 \mid Se \mid e + e$$

where n is a natural number.

a) Define an abstract syntax as an inductive type *exp*.

b) Define a semantics $eval : exp \to nat$.

c) Define an inductive predicate $step : rel\ exp$ representing the rewrite rules

$$\begin{aligned} 0 + e &\to e \\ Se_1 + e_2 &\to S(e_1 + e_2) \end{aligned}$$

d) Prove that *step* is sound.

e) Prove that *step* is complete.

f) Prove that *step* is normalizing.

g) Prove that *step* is confluent.

5.8 Termination

Informally, a node in a graph *terminates* if there is no infinite path departing from it. Formally, we define termination inductively: A node terminates if all its successors terminate.

```
Inductive terminates (r : rel) : X -> Prop :=
| terminatesI x : (forall y, r x y -> terminates r y) -> terminates r x.
```

A relation is terminating if all its nodes terminate.

```
Definition terminating (r : rel) : Prop :=
forall x, terminates r x.
```

For functional relations, normalization implies termination.

```
Lemma functional_normalizes_terminates r x :
functional r -> normalizes r x -> terminates r x.
```

```
Proof. intros F N. induction N as [x A|x y A B] ; constructor.
intros y B. exfalso. apply A. now exists y ; trivial.
intros y' C. assert (y=y') by (eapply F ; eauto). subst. trivial. Qed.
```

The induction coming with the inductive definition of termination is known as **well-founded induction**.[1] Well-founded induction allows us to prove a property p for a terminating node x by assuming that p holds for all successors of x. Here is the induction principle for *terminates*:

```
forall (r : rel) (p : X -> Prop),
(forall x : X,
   (forall y : X, r x y -> terminates r y) ->
   (forall y : X, r x y -> p y) ->
   p x) ->
forall x : X, terminates r x -> p x
```

Termination is also known as **strong normalization**. For reduction decidable relations, termination in fact implies normalization.

```
Lemma reddec_terminates_normalizes r x :
reddec r -> terminates r x -> normalizes r x.
```

The proof is by well-founded induction.

```
Proof. intros D T. induction T as [x _ IH].
destruct (D x) ; firstorder using normalizes. Qed.
```

A beautiful application of well-founded induction is **Newman's lemma**. Newman's lemma says that for terminating relations a relaxation of strong confluence know as **local confluence** is equivalent to confluence.

```
Definition locally_confluent (r : rel) : Prop :=
forall x y z, r x y -> r x z -> joinable r y z.
```

```
Lemma Newman (r : rel) :
terminating r -> locally_confluent r -> confluent r.
```

[1] In Mathematics, a well-founded relation is a relation whose inverse terminates. Well-founded induction was first studied by Emmy Noether in the 1920's.

There is a straightforward graphical proof that translates into a formal proof.

```
Proof. intros T L x. specialize (T x).
induction T as [x _ IH]. intros y z A B.
destruct A as [x|x y' y]. now exists z ; eauto using star.
destruct B as [x|x z' z]. now exists y ; eauto using star.
assert (C : joinable r y' z') by eauto. (* loc confluenced used *)
destruct C as [u [C1 C2]].
assert (D : joinable r u z) by eauto. (* IH used *)
destruct D as [w [D1 D2]].
assert (E : joinable r y w).
now apply IH with (y:=y') ; trivial ; eapply star_transitive ; eauto.
destruct E as [w' [E1 E2]].
exists w'. intuition. eapply star_transitive ; eauto. Qed.
```

Exercise 5.8.1 Prove the following goals stating two variants of the principle of well-founded induction.

```
Goal forall (r : rel) (p : X -> Prop) (x : X),
terminates r x ->
(forall x, (forall y, r x y -> p y) -> p x) ->
p x.
```

```
Goal forall (r : rel) (p : X -> Prop) (x : X),
terminates r x ->
(forall x, terminates r x -> (forall y, r x y -> p y) -> p x) ->
p x.
```

Exercise 5.8.2 Prove that subrelations of terminating relations are terminating.

```
Goal forall r x s, terminates r x -> rle s r -> terminates s x.
```

Exercise 5.8.3 (Infinitely Branching Trees) We define infinitely branching trees together with their direct subtree relation:

```
Inductive tree : Type :=
| treeL : tree
| treeN : (nat -> tree) -> tree.
```

```
Definition subtree : rel tree :=
fun s t => match s with
| treeL => False
| treeN f => exists n, f n = t
end.
```

a) Prove that *subtree* is terminating.

b) Prove that *treeL* is a normal form of every tree.

c) Prove that *subtree* is confluent.

5.9 More about Termination

Taking the transitive closure of a relations preserves termination.

```
Goal forall r x, terminates r x -> terminates (plus r) x.

Proof. intros r x A.
induction A as [x _ IHA].
constructor ; intros y B.
destruct B as [x y B | x x' y B C] ; auto.
apply IHA in B. inversion B. auto. Qed.
```

We have already shown that every subrelation of a terminating relation terminates. More generally, every relation that can be embedded into a terminating relation terminates.

```
Lemma homomorphism X Y (r : rel X) (s : rel Y) (f : X -> Y) x :
(forall x y, r x y -> s (f x) (f y)) ->
terminates s (f x) -> terminates r x.

Proof. intros A B.
remember (f x) as u ; revert x Hequ ;
  induction B as [v _ IH] ; intros x B ; subst.
constructor. eauto. Qed.
```

Note the combination of *remember* and *revert* preparing the induction; *remember* is needed since $f\,x$ appears as proper argument, and *revert* is needed since otherwise the inductive hypothesis is too weak.

Exercise 5.9.1 The **lexical product** of two relations is defined as follows.

```
Definition lex (X Y : Type) (r : rel X) (s : rel Y) : rel (X * Y) :=
fun p q => let (x,y) := p in let (x',y') := q in
r x x' \/ x=x' /\ s y y'.
```

a) Prove that the lexical product of two terminating relations is terminating.

```
Lemma lex_terminates (X Y : Type) (r : rel X) (s : rel Y) x y :
terminates r x -> terminating s -> terminates (lex r s) (x,y).
```

b) Find an example that shows that the lemma is unprovable if the termination of s is only required for y.

5.10 Complete Induction

It is not difficult to prove a lemma providing for proofs by complete induction.

```
Lemma complete_induction (p : nat -> Prop) (x : nat) :
(forall x, (forall y, y<x -> p y) -> p x) -> p x.
```

```
Proof. intros A. apply A. induction x ; intros y B.
exfalso ; omega.
apply A. intros z C. apply IHx. omega. Qed.
```

With complete induction we can show that the relation ">" on *nat* terminates.

```
Lemma gt_terminates :
terminating gt.
```

```
Proof. intros x. apply complete_induction. clear x.
intros x A. constructor. exact A. Qed.
```

Exercise 5.10.1 Size induction generalizes complete induction to arbitrary types by employing a size function. Prove the following lemma providing for proofs by size induction.

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

Hint: Follow the proof script for complete induction. Before doing the induction insert *remember* $(f\,x)$ *as* n so that you can do induction on n.

Exercise 5.10.2 Prove the following lemma, which says that a relation terminates if each step decreases the size of a node.

```
Lemma size_termination (X : Type) (r : rel X) (f : X -> nat) :
(forall x y, r x y -> f x > f y) -> terminating r.
```

6 PCF

We now consider functional languages. ML and Haskell are popular functional programming languages. The kernel language of Coq is also a functional language. In contrast to Coq, programming languages admit recursive functions that do not terminate on some or all arguments.

6.1 Abstract Syntax

PCF is an idealized functional programming language with recursive functions (i.e., procedures). It is designed such that it can describe all computable functions taking arguments from *nat* to results from *nat*. In fact, PCF stands for partial computable functions. The original version of PCF was designed by Gordon Plotkin in the 1970's.

The **types** of PCF are obtained by closing the base type *nat* under function types.

$$T \;::=\; nat \;\mid\; T \to T$$

The elements of a function type $T_1 \to T_2$ are functions that when applied to an argument of type T_1 either terminate with a result in T_2 or diverge (i.e., do not terminate).

The **terms** of PCF are defined as follows, where x ranges over an alphabet of **variables** (e.g., *nat*):

$$t \;::=\; O \;\mid\; St \;\mid\; case\ t\ t\ t \;\mid\; x \;\mid\; \lambda x\!:\!T.t \;\mid\; tt \;\mid\; fix\ t$$

Here is a term that describes a function $nat \to nat$ that doubles its argument.

$$fix\ (\lambda f\!:\!nat \to nat.\ \lambda n\!:\!nat.\ case\ n\ O\ (\lambda n'\!:\!nat.\ S(S(f\ n'))))$$

This translates to the following Coq term.

```
fix f (n : nat) : nat := match n with O => O | S n' => S (S (f n')) end
```

An important aspect of PCF is that a term $\lambda x\!:\!T.t$ introduces a **local variable** x whose scope is the term t. One says that a variable x is **free** in a term t if t has a subterm x that is not in the scope of a term $\lambda x\!:\!T.t'$. A term t is called **closed**

if no variable is free in t, and **open** if at least one variable is free in t. Local variables are also called **bound** variables.

In Coq, we represent PCF's abstract syntax as follows.

```
Inductive var : Type :=
| varN : nat -> var.

Inductive ty : Type :=
| tyN : ty                          (* nat *)
| tyF : ty -> ty -> ty.             (* function type *)

Inductive tm : Type :=
| tmO : tm                          (* zero *)
| tmS : tm -> tm                    (* successor *)
| tmC : tm -> tm -> tm -> tm        (* case *)
| tmV : var -> tm                   (* variable *)
| tmL : var -> ty -> tm -> tm       (* lambda *)
| tmA : tm -> tm -> tm              (* application *)
| tmF : tm -> tm.                   (* fix  *)
```

Exercise 6.1.1

a) Write a term $t : tm$ representing a function that adds two numbers.

b) Define an inductive predicate $free : var \to tm \to Prop$ such that $free\ x\ t$ is provable iff x is free in t.

c) Define a predicate $closed : tm \to Prop$ such that $closed\ t$ is provable iff t is closed.

Exercise 6.1.2 (Boolean Equality on Variables)

a) Define a function $eq\_var : var \to var \to bool$ and prove

```
Goal forall x y, eq_var x y = true <-> x = y.
```

b) Define a function $freeb : var \to tm \to bool$ that checks whether a variable is free in a term.

6.2 Evaluation

A closed term represents a program that may be evaluated. We will capture *evaluation* of terms with a small step semantics. We will also define a *typing predicate* that associates terms with types. The definitions of evaluation and typing will not depend on each other. In fact, evaluation will be defined for all terms, no matter whether they are closed or open or well-typed or ill-typed.

To define evaluation, we need a **substitution operation**. Given two terms s and t and a variable x, we define t^x_s as the term that is obtained from t by

replacing every free subterm occurrence of x with s. The substitution operator for PCF satisfies the equation

$$(\lambda y:T.t)^x_s \;:=\; \text{if } x = y \text{ then } \lambda y:T.t \text{ else } \lambda y:T.t^x_s$$

There is no need to avoid variable capture since the term s that is substituted in will be closed in all cases that matter (since evaluation only matters for closed terms). The simple definition of the substitution operation greatly simplifies the proofs involving substitution.

Terms of the form O, $S\,O$, $S(S\,O)$, ... are called **numerals**. We define a **value** to be a term that is either a numeral or an **abstraction** $\lambda x:T.t$.

The small-step semantics of PCF is defined with reduction rules and descent rules. Here are the **reduction rules**.

$$\begin{aligned}
\mathit{case}\ O\ t_1\ t_2 &\to t_1 && \\
\mathit{case}\ (S\,t)\ t_1\ t_2 &\to t_2\,t && \text{if } S\,t \text{ is a value} \\
(\lambda x:T.t)\,v &\to t^x_v && \text{if } v \text{ is a value} \\
\mathit{fix}\,(\lambda x:T.t) &\to t^x_{\mathit{fix}\,(\lambda x:T.t)} &&
\end{aligned}$$

Note that a **β-redex** $(\lambda x:T.t)s$ can only be reduced if the argument term s is a value. This restriction is known as **call by value**.

The **descent rules** are as follows.

$$\begin{aligned}
St &\to St' && \text{if } t \to t' \\
\mathit{case}\ t\ t_1\ t_2 &\to \mathit{case}\ t'\ t_1\ t_2 && \text{if } t \to t' \\
t_1\ t_2 &\to t'_1\ t_2 && \text{if } t_1 \to t'_1 \\
(\lambda x:T.t)\,t_2 &\to (\lambda x:T.t)\,t'_2 && \text{if } t_2 \to t'_2
\end{aligned}$$

Note that the descent rules do not provide for reduction inside abstractions. This is typical for programming languages and in contrast to Coq where reductions are possible everywhere.

From the reduction and descent rules it is clear that PCF's reduction relation is functional.

Exercise 6.2.1 Find a PCF term whose evaluation does not terminate.

Exercise 6.2.2 (Small-Step Semantics)

a) Define a function $\mathit{subst} : \mathit{tm} \to \mathit{var} \to \mathit{tm} \to \mathit{tm}$ such that $\mathit{subst}\ t\ x\ s$ yields the term that is obtained from t by replacing every free occurrence of x with s. Capture of free variables in s by variable binders in t is fine.

b) Define predicates $\mathit{nvalue} : \mathit{tm} \to \mathit{Prop}$ and $\mathit{value} : \mathit{tm} \to \mathit{Prop}$ saying which terms are numeric values and values.

c) Define the small-step semantics of PCF with an inductive predicate $step : tm \to tm \to Prop$.

d) Prove $\forall\ t\ t'.\ value\ t \to step\ t\ t' \to \bot$.

Exercise 6.2.3 (Big-Step Semantics)

a) Define the big-step semantics of PCF with an inductive predicate $sem : tm \to tm \to Prop$.

b) Prove $forall\ t\ t',\ sem\ t\ t' \to value\ t'$.

6.3 Primitive Recursion and T

The general recursion operator of PCF can be replaced by a primitive recursion operator. In Coq, the primitive recursion operator can be defined as follows.

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

The primitive recursion operator can express the case operator of PCF. This observation leads to a language T whose types and terms are defined as follows.

$$T ::= nat \mid T \to T$$
$$t ::= O \mid St \mid primrec\ t\ t\ t \mid x \mid \lambda x{:}T.t \mid tt$$

It turns out that the evaluation of well-typed T programs always terminates (first shown by William Tait in 1967). The language T was first proposed by Kurt Gödel in 1958.

Exercise 6.3.1 Write a function in T that adds two numbers. Translate your function to Coq and test it for some arguments. Note that T translates directly to Coq with *primrec* as defined above.

Exercise 6.3.2 Define an abstract syntax and a small-step semantics for T in Coq. Follow the development of PCF.

6.4 Typing Discipline and E

The type discipline for PCF should have the following properties:

1. **Preservation** If a term t of type T reduces to a term t', then t' has type T.
2. **Progress** A closed and well-typed term is either a value or reducible.

To prepare the definition of the typing discipline for PCF, we first consider a simpler functional language E without variables. The expressions of E denote boolean values and numbers.

$$t ::= \mathit{true} \mid \mathit{false} \mid \mathit{if}\ t\ t\ t \mid O \mid S\,t \mid P\,t \mid Z\,t$$

The terms *true* and *false* are called **boolean values**. Terms of the form O, $S\,O$, $S(S\,O)$, ... are called **numerals**. A **value** is either a boolean value or a numeral. The **reduction rules** for E are as follows.

$$\begin{aligned}
\mathit{if}\ \mathit{true}\ t_1\ t_2 &\to t_1 \\
\mathit{if}\ \mathit{false}\ t_1\ t_2 &\to t_2 \\
P\,O &\to O \\
P(S\,t) &\to t && \text{if } S\,t \text{ is a value} \\
Z\,O &\to \mathit{true} \\
Z(S\,t) &\to \mathit{false} && \text{if } S\,t \text{ is a value}
\end{aligned}$$

Here are the **descent rules** for E.

$$\begin{aligned}
\mathit{if}\ t\ t_1\ t_2 &\to \mathit{if}\ t'\ t_1\ t_2 && \text{if } t \to t' \\
St &\to St' && \text{if } t \to t' \\
Pt &\to Pt' && \text{if } t \to t' \\
Zt &\to Zt' && \text{if } t \to t'
\end{aligned}$$

Plotkin's original PCF was obtained from E by adding variables, λ-abstractions, applications, and the recursion operator *fix*. In our PCF, the operator *case* replaces the operators *if*, P, and Z. Moreover, *case* makes it possible to omit boolean values.

E's type discipline is given by two types

$$T ::= \mathit{bool} \mid \mathit{nat}$$

and one **typing rule** for each syntactic construct.

$$\frac{}{\mathit{true} : \mathit{bool}} \qquad \frac{}{\mathit{false} : \mathit{bool}} \qquad \frac{t : \mathit{bool} \qquad t_1 : T \qquad t_2 : T}{\mathit{if}\ t\ t_1\ t_2}$$

$$\frac{}{O : \mathit{nat}} \qquad \frac{t : \mathit{nat}}{S\,t : \mathit{nat}} \qquad \frac{t : \mathit{nat}}{P\,t : \mathit{nat}} \qquad \frac{t : \mathit{nat}}{Z\,t : \mathit{bool}}$$

The formalization of E in Coq is straightforward.

```
Inductive tm : Type :=
| tmT : tm
| tmF : tm
| tmI : tm -> tm -> tm -> tm
| tmO : tm
| tmS : tm -> tm
| tmP : tm -> tm
| tmZ : tm -> tm.

Inductive nvalue : tm -> Prop :=
| nvalueO : nvalue tmO
| nvalueS t : nvalue t -> nvalue (tmS t).

Inductive value : tm -> Prop :=
| valueF : value tmT
| valueT : value tmF
| valueN t : nvalue t -> value t.

Inductive step : tm -> tm -> Prop :=
| stepIT t1 t2 : step (tmI tmT t1 t2) t1
| stepIF t1 t2 : step (tmI tmF t1 t2) t2
| stepPO : step (tmP tmO) tmO
| stepPS t : nvalue t -> step (tmP (tmS t)) t
| stepZO : step (tmZ tmO) tmT
| stepZS t : nvalue t -> step (tmZ (tmS t)) tmF
| stepDI t t' t1 t2 : step t t' -> step (tmI t t1 t2) (tmI t' t1 t2)
| stepDS t t' : step t t' -> step (tmS t) (tmS t')
| stepDP t t' : step t t' -> step (tmP t) (tmP t')
| stepDZ t t' : step t t' -> step (tmZ t) (tmZ t').

Inductive ty : Type :=
| Nat : ty
| Bool : ty.

Inductive type : tm -> ty -> Prop :=
| typeT : type tmT Bool
| typeF : type tmF Bool
| typeI t t1 t2 T : type t Bool -> type t1 T -> type t2 T -> type (tmI t t1 t2) T
| typeO : type tmO Nat
| typeS t : type t Nat -> type (tmS t) Nat
| typeP t : type t Nat -> type (tmP t) Nat
| typeZ t : type t Nat -> type (tmZ t) Bool.
```

Exercise 6.4.1

a) Normal terms that are not values are called **stuck**. Find a stuck term.

b) Find an example showing that the step relation does not preserve types from right to left.

Exercise 6.4.2 Suppose we add two new reduction rules:

$$P\ true \rightarrow P\ false$$
$$P\ false \rightarrow P\ true$$

Which of the following properties remain true in the presence of these rules?

a) Determinacy of *step*

b) Termination of *step* for well-typed terms

c) Progress

d) Preservation

Exercise 6.4.3 Suppose we add a new typing rule:

$$\frac{t_1 : T}{if\ true\ t_1\ t_2 : T}$$

Which of the following properties remain true in the presence of these rules?

a) Determinacy of *step*

b) Termination of *step* for well-typed terms

c) Progress

d) Preservation

Exercise 6.4.4 Prove the following lemmas.

```
Lemma value_normal t t' :
value t -> step t t' -> False.

Lemma preservation t T t' :
type t T -> step t t' -> type t' T.

Lemma progress t T :
type t T -> value t \/ exists t', step t t'.

Lemma type_unique t T T' :
type t T -> type t T' -> T = T'.

Lemma step_deterministic t t1 t2 :
step t t1 -> step t t2 -> t1 = t2.
```

Exercise 6.4.5 Prove that *step* terminates.

Exercise 6.4.6 Write a function $tycheck : tm \rightarrow option\ ty$ and prove the following lemma.

```
Lemma tycheck_correct t T :
type t T <-> tycheck t = Some T.
```

6.5 Type-Indexed Syntax for E

In Coq it is possible to define an inductive constructor $tm : ty \to Type$ such that the members of $tm\ T$ are exactly the well-typed terms of type T of E.

```
Inductive tm : ty -> Type :=
| tmT : tm Bool
| tmF : tm Bool
| tmI T : tm Bool -> tm T -> tm T -> tm T
| tmO : tm Nat
| tmS : tm Nat -> tm Nat
| tmP : tm Nat -> tm Nat
| tmZ : tm Nat -> tm Bool.
```

Given this **type-indexed syntax** for E, we can define the small-steps semantics of E such that type preservation is guaranteed by type checking.

```
Inductive value : forall T : ty, tm T -> Prop :=
| valueF : value tmT
| valueT : value tmF
| valueO : value tmO
| valueS t : value t -> value (tmS t).

Inductive step : forall T : ty, tm T -> tm T -> Prop :=
| stepIT T (t1 t2 : tm T) : step (tmI tmT t1 t2) t1
| stepIF T (t1 t2 : tm T) : step (tmI tmF t1 t2) t2
| stepPO : step (tmP tmO) tmO
| stepPS t : value t -> step (tmP (tmS t)) t
| stepZO : step (tmZ tmO) tmT
| stepZS t : value t -> step (tmZ (tmS t)) tmF
| stepDI t t' T (t1 t2 : tm T) : step t t' -> step (tmI t t1 t2) (tmI t' t1 t2)
| stepDS t t' : step t t' -> step (tmS t) (tmS t')
| stepDP t t' : step t t' -> step (tmP t) (tmP t')
| stepDZ t t' : step t t' -> step (tmZ t) (tmZ t').
```

When we attempt to prove properties of the type-indexed presentation of E, it turn out that the tactic *inversion* does not work for members of types obtained with *value step*. The problem can be circumvented by loading the right library

```
Require Import Program.Equality.
```

and using the tactic *dependent destruction* in place of *inversion*.

```
Lemma value_normal (T : ty) (t t' : tm T) :
value t -> step t t' -> False.

Proof. intros A B.
induction B ; dependent destruction A ; auto. Qed.
```

Exercise 6.5.1 Prove the following lemmas,

```
Lemma step_deterministic (T : ty) (t t1 t2 : tm T) :
step t t1 -> step t t2 -> t1 = t2.

Lemma Progress (T : ty) (t : tm T) :
value t \/ exists t', step t t'.
```

Exercise 6.5.2 (Challenge) Consider the function

```
Definition ty_den (T : ty) : Type :=
match T with Bool => bool | Nat => nat end.
```

Define a function $tm\_den : \forall T.\ tm\ T \to ty\_den T$ and prove that two terms are convertible with respect to *step* if and only if their denotation under *tm_den* agrees.

6.6 Simply Typed Lambda Calculus

We now shift our attention to functions. For this we consider a minimal system known as **simply typed lambda calculus** (STLC):

$$\begin{array}{rcl} T & ::= & X \mid T \to T \\ t & ::= & x \mid \lambda x{:}T.t \mid t\ t \end{array}$$

Types are obtained by closing a single base type X under function types, and terms are obtained by closing a set of variables (isomorphic to *nat*) under lambda abstraction and application. The values of STLC are exactly the terms $\lambda x{:}T.t$. There is a singe reduction rule and two descent rules.

$$\begin{array}{rcll} (\lambda x{:}T.t)\,v & \to & t^x_v & \text{if } v \text{ is a value} \\ t_1\ t_2 & \to & t_1'\ t_2 & \text{if } t_1 \to t_1' \\ (\lambda x{:}T.t)\ t_2 & \to & (\lambda x{:}T.t)\ t_2' & \text{if } t_2 \to t_2' \end{array}$$

This yields a reduction relation that is call by value and weak (i.e., no reduction below lambda. The novel part of STLC is the typing relation. To maintain the typing assumptions for variables we use **contexts** Γ, which are partial functions mapping variables to types. There are three typing rules:

$$\begin{array}{ll} \Gamma \vdash x : T & \text{if } \Gamma x = T \\ \Gamma \vdash \lambda x{:}T.t : T \to T' & \text{if } \Gamma^x_T \vdash t : T' \\ \Gamma \vdash t_1 t_2 : T & \text{if } \Gamma \vdash t_1 : T_2 \to T \text{ and } \Gamma \vdash t_2 : T_2 \end{array}$$

Exercise 6.6.1 Formalize STLC in Coq. For the abstract syntax and the small-step semantics of follow the development of PCF. For contexts and the typing relation follow the development in the SF text.

7 Termination of Simply Typed λ-Calculus

Recall the simply typed λ-calculus.

$$S, T ::= X \mid T \to T$$
$$s, t ::= x \mid \lambda x{:}T.t \mid t\ t$$

Terms of the form $\lambda x{:}T.t$ are called **λ-abstractions**. Values are exactly the λ-abstractions.

We can represent these in Coq as follows.

```
Inductive ty : Type :=
  | tyX : ty
  | tyA : ty -> ty -> ty.

Inductive tm : Type :=
  | tmV : var -> tm
  | tmA : tm -> tm -> tm
  | tmL : var -> ty -> tm -> tm.

Inductive value : tm -> Prop :=
  | v_abs : forall x T t, value (tmL x T t).
```

A variable x occurs free in a term t if it occurs in a position that is not bound by a λ. This can be defined as an inductive proposition in Coq as follows:

```
Inductive free : var -> tm -> Prop :=
  | freeV : forall x,
      free x (tmV x)
  | freeA1 : forall x t1 t2,
      free x t1 -> free x (tmA t1 t2)
  | freeA2 : forall x t1 t2,
      free x t2 -> free x (tmA t1 t2)
  | freeL : forall x y T11 t12,
        y <> x
      -> free x t12
      -> free x (tmL y T11 t12).
```

We say a term is *closed* if no variable occurs free in it.

In this chapter we will prove that a certain weak reduction relation terminates. From this it will follow that the weak, call-by-value reduction relation terminates.

7.1 Partial Maps, Substitutions, and Contexts

We will define a notion of (simultaneous) substitution and an operation applying it to a term. Subsitutions will be partial functions from variables to terms. Contexts will also be partial functions (in this case from variables to types).

In general, we can consider partial functions f from variables to some set A. We use $\mathsf{dom} f$ to refer to the set of variables on which f is defined. We call $\mathsf{dom} f$ the *domain of* f. The empty function $\emptyset$ is such a partial function. Given any such partial function f, a variable x and an element $a \in A$, let f_a^x denote the update of f to send x to a and otherwise behave as f. Note that $\mathsf{dom} f_a^x$ is $\mathsf{dom} f \cup \{x\}$. Given any such partial function f and a variable x, let f^{-x} denote the partial function removing x from the domain of f. Note that $\mathsf{dom} f^{-x}$ is $\mathsf{dom} f \setminus \{x\}$.

We use θ to range over substitutions (partial functions from variables to terms). We use Γ to range over contexts (partial functions from variables to types).

In Coq we can represent partial functions using types of the form *var -> option A*. A variable x is in the domain of a partial function f if $f\ x$ is of the form *Some a*. A variable x is not in the domain of f if $f\ x$ is *None*.

```
Definition partial_map (A:Type) := var -> option A.
Definition sub : Type := partial_map tm.
Definition ctx : Type := partial_map ty.

Definition empty {A:Type} : partial_map A := (fun _ => None).

Definition update {A:Type} (Gamma : partial_map A) (x:var) (T : A) :=
  fun x' => if beq_var x x' then Some T else Gamma x'.

Definition drop {A:Type} (Gamma : partial_map A) (x:var) :=
  fun x' => if beq_var x x' then None else Gamma x'.
```

We can apply a substitution to a term as follows:

$$\begin{aligned} \theta x &:= s \text{ if } x \in \mathsf{dom}\theta \text{ and } \theta x = s \\ \theta x &:= x \text{ if } x \notin \mathsf{dom}\theta \\ \theta(st) &:= (\theta s)(\theta t) \\ \theta(\lambda x{:}T.t) &:= \lambda x{:}T.\theta^{-x}t \end{aligned}$$

We use t_s^x to denote $\emptyset_s^x t$.

Lemma 7.1.1 (Coincidence) If $\theta x = \theta' x$ for all x free in t, then $\theta t = \theta' t$.

Proof By induction on t. ∎

Lemma 7.1.2 (Id) $\emptyset t = t$.

Proof By induction on t. ∎

Lemma 7.1.3 If t is closed, then $\theta t = t$.

Proof By the Coincidence and Id lemmas. ∎

A substitution θ is closed if for every $x \in \mathsf{dom}\theta$, θx is closed.

Lemma 7.1.4 If θ is closed, then $(\theta^{-x} t)^x_s = \theta^x_s t$.

Proof By induction on t using the Coincidence and Id lemmas. ∎

7.2 Reduction

Earlier we considered a reduction relation that was call by value and weak (no reduction below λ).

$$\begin{array}{rcll} (\lambda x{:}T.t)v & \to & t^x_v & \text{if } v \text{ is a value} \\ t_1\ t_2 & \to & t'_1\ t_2 & \text{if } t_1 \to t'_1 \\ (\lambda x{:}T.t)\ t_2 & \to & (\lambda x{:}T.t)\ t'_2 & \text{if } t_2 \to t'_2 \end{array}$$

In this chapter we will prove this reduction relation terminates. In fact, we will show a more general nondeterministic weak reduction relation defined as follows.

$$\begin{array}{rcll} (\lambda x{:}T.t)s & \Rightarrow & t^x_s & \\ t_1\ t_2 & \Rightarrow & t'_1\ t_2 & \text{if } t_1 \Rightarrow t'_1 \\ t_1\ t_2 & \Rightarrow & t_1\ t'_2 & \text{if } t_2 \Rightarrow t'_2 \end{array}$$

7.3 Typing

The typing relation is defined as before.

$$\begin{array}{ll} \Gamma \vdash x : T & \text{if } \Gamma x = T \\ \Gamma \vdash \lambda x{:}S.t : S \to T & \text{if } \Gamma^x_S \vdash t : T \\ \Gamma \vdash ts : T & \text{if } \Gamma \vdash t : S \to T \text{ and } \Gamma \vdash s : S \end{array}$$

In Coq we can write this as follows.

```
Inductive type : ctx -> tm -> ty -> Prop :=
  | typeV : forall Gamma x T,
      Gamma x = Some T ->
      type Gamma (tmV x) T
  | typeL : forall Gamma x S T t,
      type (update Gamma x S) t T ->
      type Gamma (tmL x S t) (tyA S T)
  | typeA : forall S T Gamma t s,
      type Gamma t (tyA S T) ->
      type Gamma s S ->
      type Gamma (tmA t s) T.
```

We have the following results.

Lemma 7.3.1 (Invariance) Suppose $\Gamma x = \Gamma' x$ for every x free in t. If $\Gamma \vdash t : T$, then $\Gamma' \vdash t : T$.

Proof By induction on $\Gamma \vdash t : T$. ∎

Lemma 7.3.2 If x is free in t and $\Gamma \vdash t : T$, then $x \in \mathsf{dom}\Gamma$.

Proof By induction on $\Gamma \vdash t : T$. ∎

Lemma 7.3.3 If $\emptyset \vdash t : T$, then $\Gamma \vdash t : T$.

Proof Apply Invariance and Lemma 7.3.2. ∎

Lemma 7.3.4 (Substitution) Suppose $\Gamma \vdash t : T$ and for every $x \in \mathsf{dom}\Gamma$ we have $x \in \mathsf{dom}\theta$ and $\emptyset \vdash \theta x : \Gamma x$. Then $\emptyset \vdash \theta t : T$.

Proof A generalization can be proven by induction on $\Gamma \vdash t : T$. We leave this as an exercise. ∎

Lemma 7.3.5 (Preservation) If $\emptyset \vdash t : T$ and $t \Rightarrow t'$, then $\emptyset \vdash t' : T$.

Proof By induction on $t \Rightarrow t'$ using the substitution lemma. ∎

7.4 The Logical Relation R

We say a term t terminates if it terminates relative to $\Rightarrow$. Our goal is to prove that if $\emptyset \vdash t : T$, then t terminates. We cannot directly prove this. Instead, we define an apparently stronger property that will allow us to do the inductive proofs.

We define a relation $R_T t$ between types T and terms t by recursion on types.

- $R_X t$ holds if $\emptyset \vdash t : X$ and t terminates.

- $R_{S\to T}t$ holds if $\emptyset \vdash t : S \to T$, t terminates and for every s, if $R_S s$, then $R_T(ts)$.

We can extend R to contexts and substitutions in the following obvious way. We say $R_\Gamma\theta$ holds if $\mathsf{dom}\Gamma = \mathsf{dom}\theta$ and $R_{\Gamma x}\theta x$ for every $x \in \mathsf{dom}\Gamma$.

The following lemmas are obvious.

Lemma 7.4.1 If $R_T t$, then $\emptyset \vdash t : T$.

Lemma 7.4.2 If $R_T t$, then t terminates.

We can also easily obtain the following results.

Lemma 7.4.3 $R_\Gamma\theta$ implies θ is closed.

Proof Let $x \in \mathsf{dom}\theta$ such that $\theta x = s$. We know $\emptyset \vdash s : \Gamma x$. By Lemma 7.3.2 we know s is closed, as desired. ∎

Lemma 7.4.4 If $\Gamma \vdash t : T$ and $R_\Gamma\theta$, then $\emptyset \vdash \theta t : T$.

Proof This follows from Substitution (Lemma 7.3.4). ∎

Lemma 7.4.5 If $R_\Gamma\theta$ and $R_T t$, then $R_{\Gamma^x_T}(\theta^x_t)$.

Proof Trivial. ∎

Lemma 7.4.6 If $t \Rightarrow t'$ and $R_T t$, then $R_T t'$.

Proof Induction on T using preservation. ∎

We will need to prove that under certain conditions if every reduct of a term t satisfies R_T, then t satisfies R_T. Let us define a notation for this concept. Let $E_T t$ hold if $R_T t'$ holds for every t' such that $t \Rightarrow t'$.

Lemma 7.4.7 Suppose for every u such that $\emptyset \vdash u : T$ and u is not a λ-abstraction, if $E_T u$, then $R_T u$. If t is not a λ-abstraction, $\emptyset \vdash t : (S \to T)$, and $E_{S\to T}t$, then for every s, if $R_S s$, then $R_T(ts)$.

Proof Since every s satisfying $R_S s$ is terminating (see Lemma 7.4.2), we can prove this by induction on the termination of s. Lemma 7.4.6 is helpful. ∎

Lemma 7.4.8 Suppose $\emptyset \vdash t : T$ and t is not a λ-abstraction. If $E_T t$, then $R_T t$.

Proof By induction on the type T using Lemma 7.4.7. ∎

Lemma 7.4.9 Suppose $\emptyset^x_S \vdash t : T$ and forall s, if $R_S s$, then $R_T(t^x_s)$. Then for all s, if $R_S s$, then $R_T((\lambda x : S.t)s)$.

Proof Since every s satisfying $R_S s$ is terminating (see Lemma 7.4.2), we can prove this by induction on the termination of s. Lemma 7.4.8 is helpful. ∎

Lemma 7.4.10 If $\Gamma \vdash t : T$ and $R_\Gamma \theta$, then $R_T(\theta t)$.

Proof We prove this by induction on $\Gamma \vdash t : T$. In the variable case, $\Gamma x = T$ and so we know $R_T(\theta x)$ since $R_\Gamma \theta$. For the application case, assume $\Gamma \vdash t : S \to T$, $\Gamma \vdash s : S$ and $R_\Gamma \theta$. We wish to prove $R_T(\theta(ts))$. The inductive hypotheses imply $R_{S\to T}(\theta t)$ and $R_S(\theta s)$. By the definition of $R_{S\to T}$ we conclude $R_T((\theta t)(\theta s))$ as desired.

The most involved case is the λ case. Assume $\Gamma_S^x \vdash t : T$ and $R_\Gamma \theta$. We wish to prove $R_{S\to T}(\theta(\lambda x : S.t))$. We know $\emptyset \vdash \theta(\lambda x : S.t) : S \to T$ by Lemma 7.4.4, $R_\Gamma \theta$ and the fact that $\Gamma \vdash \lambda x : S.t : S \to T$. Note that $\theta(\lambda x : S.t)$ is $\lambda x : S.\theta^{-x} t$. We know this term terminates because λ-abstractions do not reduce. Finally, we must prove $R_T((\lambda x : S.\theta^{-x} t)s)$ holds for every s such that $R_S s$. First, note that for every s such that $R_S s$ we know $R_{\Gamma_S^x}(\theta_s^x)$ holds by Lemma 7.4.5 and so $R_T(\theta_s^x t)$ holds by the inductive hypothesis. Second, note that $\emptyset_S^x \vdash \theta^{-x} t : T$ since $\emptyset \vdash \lambda x : S.\theta^{-x} t : S \to T$. The proof is completed by applying Lemma 7.4.9 with $\theta^{-x} t$. ∎

Lemma 7.4.11 If $\emptyset \vdash t : T$, then t terminates.

Proof By Lemma 7.4.10 with $\emptyset$ as the substitution we know $R_T t$ and so t terminates by Lemma 7.4.2. ∎

8 Calculus of Constructions

The calculus of constructions (CC) is a subsystem of the type theory underlying Coq (calculus of inductive constructions). It my be seen as a radical generalization of the simply typed lambda calculus giving first-class status to types (i.e., types can be arguments and results of functions). The original version of the calculus of constructions was proposed in 1985 by Thierry Coquand and Gérard Huet. The canonical reference for the calculus of constructions is Luo's book.

8.1 Syntax

The terms of CC are obtained from **variables** (x, y, z) and **universes** (U).

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

Since types have first-class status in CC, they are represented as terms. Terms starting with $\forall$ are called **function types**. Universes are types whose members are function types and universes. There is a variable x_n and a universe U_n for every natural number n.

Terms of the form $\lambda x{:}s.t$ and $\forall x{:}s.t$ bind the local variable x in their body t. The term s acts as type of x. **Free variables** and **closed terms** are defined accordingly.

CC is based on an abstract representation of local variables. This means that terms that a equal up to renaming of local variables are in fact equal. For instance, $\lambda x{:}U_0.x$ and $\lambda y{:}U_0.y$ are different notations for the same term. Abstract local variables can be formalized using a technique invented by de Bruijn (so-called **de Bruijn terms**).

8.2 Substitution and Reduction

Substitution is defined such that the binders λ.and $\forall$ do not capture variables. For instance, $(\lambda x{:}y.fxy)^y_x = \lambda z{:}x.fzx$. There is a single reduction rule, called **beta reduction**, that can be applied everywhere in a term.

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

It can be shown that beta reduction is confluent (see Luo's book for a proof). Two terms are **convertible** if they reduce to the same term. We write $s \to^* t$ to say that s reduces to t and $s \approx t$ to say that s and t are convertible.

8.3 Typing

A **context** (Γ) is a list of variable declarations $x:t$. The typing relation concerns **judgements** $\Gamma \vdash s : t$ and is defined inductively by the following rules.

$$\text{CE}\quad \frac{}{\emptyset \vdash U_0 : U_1}$$

$$\text{CV}\quad \frac{\Gamma \vdash t : U}{\Gamma, x:t \vdash U_0 : U_1}\quad x \text{ not declared in } \Gamma$$

$$\text{Var}\quad \frac{\Gamma, x:t, \Gamma' \vdash U_0 : U_1}{\Gamma, x:t, \Gamma' \vdash x : t}$$

$$\text{Lam}\quad \frac{\Gamma, x:s \vdash t : u}{\Gamma \vdash \lambda x:s.t : \forall x:s.u}$$

$$\text{App}\quad \frac{\Gamma \vdash s : \forall x:u.v \qquad \Gamma \vdash t : u}{\Gamma \vdash st : v^x_t}$$

$$\text{Fun}\quad \frac{\Gamma \vdash s : U \qquad \Gamma, x:s \vdash t : U}{\Gamma \vdash \forall x:s.t : U}$$

$$\text{Uni}\quad \frac{\Gamma \vdash U_0 : U_1}{\Gamma \vdash U_n : U_{n+1}}$$

$$\text{Con}\quad \frac{\Gamma \vdash s : t \qquad \Gamma \vdash t' : U}{\Gamma \vdash s : t'}\quad t \approx t'$$

$$\text{Sub}\quad \frac{\Gamma \vdash s : t}{\Gamma \vdash s : t'}\quad t < t'$$

The **subtyping relation** used in the last rule is defined inductively by two rules.

$$\frac{m < n}{U_m < U_n} \qquad \frac{t < t'}{\forall x:s.t < \forall x:s.t'}$$

Judgements of the form

- $\Gamma \vdash U_0 : U_1$ may be read as Γ **is well-formed**.
- $\Gamma \vdash s : U$ may be read as s **is well-typed in** Γ.

Note that well-formed contexts declare a variable at most once and assign types that are well-defined in the preceding context.

Here are the most important properties of the typing relation.

- **Propagation** If $\Gamma \vdash s : t$, then $\Gamma \vdash t : U$ for some universe U.
- **Type preservation** Reduction preserves typings.
- **Termination** Reduction terminates on all well-typed terms.
- **Decidability** The typing relation is decidable.

Exercise 8.3.1 Suppose $\emptyset \vdash s : t$ and s is normal. Find out whether s can be a variable or an application.