Chapter 1

From Inductive Types

```
Require Import Coq.Setoids.Setoid.
```

1. Define an inductive type *truth* with three constructors, *Yes*, *No*, and *Maybe*. *Yes* stands for certain truth, *No* for certain falsehood, and *Maybe* for an unknown situation. Define "not," "and," and "or" for this replacement boolean algebra. Prove that your implementation of "and" is commutative and distributes over your implementation of "or."

```
Module EX1.
Inductive truth : Type := Yes | No | Maybe.

Definition not (a : truth) : truth :=
  match a with
  | Yes ⇒ No
  | No ⇒ Yes
  | Maybe ⇒ Maybe
  end.

Check not Yes.

Definition and (a b : truth) : truth :=
match a with
| Yes ⇒ b
| No ⇒ match b with
          | Maybe ⇒ Maybe
          | _ ⇒ No
          end
| Maybe ⇒ Maybe
end.

Definition or (a b : truth) : truth :=
  match a with
  | Yes ⇒ match b with
```

```
            | Maybe ⇒ Maybe
            | _ ⇒ Yes
            end
    | No ⇒ b
    | Maybe ⇒ Maybe
    end.
Lemma and_comm : ∀ (a b : truth), and a b = and b a.
  intros; destruct a, b; auto. Qed.
Lemma or_comm : ∀ (a b : truth), or a b = or b a.
  intros; destruct a, b; auto. Qed.
Lemma or_distr : ∀ (a b c : truth), or (and a b) c = and (or a c) (or b c).
  intros; destruct a, b, c; auto. Qed.
End EX1.
```

2. Define an inductive type *slist* that implements lists with support for constant-time concatenation. This type should be polymorphic in a choice of type for data values in lists. The type *slist* should have three constructors, for empty lists, singleton lists, and concatenation. Define a function *flatten* that converts *slists* to *lists*. (You will want to run `Require Import List.` to bring list definitions into scope.) Finally, prove that *flatten* distributes over concatenation, where the two sides of your quantified equality will use the *slist* and *list* versions of concatenation, as appropriate. Recall from Chapter 2 that the infix operator ++ is syntactic sugar for the *list* concatenation function *app*.

```
Module EX2.
Require Import List.
Set Implicit Arguments.
Inductive slist (X : Type) : Type:=
| s_nil : slist X
| s_singleton : X → slist X
| s_cons : slist X → slist X → slist X.
Fixpoint flattern (X : Type) (sl : slist X) : list X :=
  match sl with
  | @s_nil _ ⇒ nil
  | s_singleton a ⇒ a::nil
  | s_cons sl1 sl2 ⇒ (flattern sl1) ++ (flattern sl2)
  end.
Fixpoint s_app (X : Type) (s1 s2 : slist X) : slist X:=
  match s1 with
  | @s_nil _⇒ s2
  | s_singleton a' as a ⇒ s_cons a s2
  | s_cons a s1' ⇒ s_cons a (s_app s1' s2)
  end.
```

```
Lemma flattern_distr : ∀ (X : Type) (a b : slist X), flattern (s_app a b) = (flattern a)
++ (flattern b).
  induction a; intuition.
  - simpl. rewrite ← app_assoc. rewrite ← (IHa2 b). reflexivity. Qed.
End EX2.
```

3. Modify the first example language of Chapter 2 to include variables, where variables are represented with *nat*. Extend the syntax and semantics of expressions to accommodate the change. Your new *expDenote* function should take as a new extra first argument a value of type *var* → *nat*, where *var* is a synonym for naturals-as-variables, and the function assigns a value to each variable. Define a constant folding function which does a bottom-up pass over an expression, at each stage replacing every binary operation on constants with an equivalent constant. Prove that constant folding preserves the meanings of expressions.

```
Module EX3.
Inductive binop : Set := Plus | Times.
Inductive var := vvar : nat→ var.
Inductive exp : Set :=
| Const : nat → exp
| Binop : binop → exp → exp → exp
| Var : var → exp.

Definition binopDenote (b : binop) :=
match b with
| Plus ⇒ plus
| Times ⇒ mult
end.

Fixpoint expDenote (ass : var→ nat) (e : exp) : nat :=
  match e with
  | Const n ⇒ n
  | Binop b e1 e2 ⇒ (binopDenote b) (expDenote ass e1) (expDenote ass e2)
  | Var v ⇒ ass v
  end.

Fixpoint const_fold (e : exp) : exp:=
  match e with
  | Const n ⇒ e
  | Var v ⇒ e
  | Binop b e1 e2 ⇒ match e1, e2 with
                     | Const n1, Const n2 ⇒ Const ((binopDenote b) n1 n2)
                     | _, _ ⇒ Binop b (const_fold e1) (const_fold e2)
                     end
  end.
```

```
Lemma const_fold_correct : ∀ (e : exp) (ass : var → nat), expDenote ass e = expDenote
ass (const_fold e).
  induction e; intuition; destruct b; induction e1, e2;
    auto; simpl; f_equal; simpl in *; auto. Qed.
End EX3.
```

4. Reimplement the second example language of Chapter 2 to use mutually inductive types instead of dependent types. That is, define two separate (non-dependent) inductive types *nat_exp* and *bool_exp* for expressions of the two different types, rather than a single indexed type. To keep things simple, you may consider only the binary operators that take naturals as operands. Add natural number variables to the language, as in the last exercise, and add an "if" expression form taking as arguments one boolean expression and two natural number expressions. Define semantics and constant-folding functions for this new language. Your constant folding should simplify not just binary operations (returning naturals or booleans) with known arguments, but also "if" expressions with known values for their test expressions but possibly undetermined "then" and "else" cases. Prove that constant-folding a natural number expression preserves its meaning.

```
Module EX4.
  Require Import Arith.
  Inductive nbinop : Set := NPlus | NTimes.
  Inductive bbinop : Set := TEq | TLt.

  Inductive var : Set := vvar : nat → var.
  Inductive bool_exp : Set:=
  | BEq : nat → nat → bool_exp
  | BLt : nat → nat → bool_exp
  | BConst : bool → bool_exp.

  Inductive nat_exp : Set :=
  | NConst : nat → nat_exp
  | NBinop : nbinop → nat_exp → nat_exp → nat_exp
  | NVar : var → nat_exp
  | NIf : bool_exp → nat_exp → nat_exp → nat_exp.

  Definition bbinopDenote bb :=
  match bb with
  | TEq ⇒ beq_nat
  | TLt ⇒ Nat.leb
  end.

  Definition nbinopDenote nb :=
  match nb with
  | NPlus ⇒ plus
  | NTimes ⇒ mult
```

```
end.
Fixpoint bexpDenote (e : bool_exp) : bool :=
  match e with
  | BEq n1 n2 ⇒ beq_nat n1 n2
  | BLt n1 n2 ⇒ Nat.leb n1 n2
  | BConst b ⇒ b
  end.
Fixpoint nexpDenote (ass : var → nat) (e : nat_exp) : nat :=
  match e with
  | NConst n1 ⇒ n1
  | NBinop b e1 e2 ⇒ (nbinopDenote b) (nexpDenote ass e1) (nexpDenote ass e2)
  | NVar v ⇒ ass v
  | NIf b e1 e2 ⇒ if (bexpDenote b) then (nexpDenote ass e1) else (nexpDenote
ass e2)
  end.
Fixpoint fold_const (e : nat_exp) : nat_exp :=
  match e with
  | NBinop b e1 e2 ⇒ match e1, e2 with
                     | NConst n1, NConst n2 ⇒ NConst ((nbinopDenote b) n1 n2)
                     | _, _ ⇒ NBinop b (fold_const e1) (fold_const e2)
                     end
  | NIf b e1 e2 ⇒ if (bexpDenote b) then (fold_const e1) else (fold_const e2)
  | _ ⇒ e
  end.
Lemma fold_const_correct : ∀ (e : nat_exp) (ass : var→ nat),
    nexpDenote ass e = nexpDenote ass (fold_const e).
  induction e; intuition.
  - destruct n; induction e1, e2; auto; simpl; f_equal; simpl in *; auto.
  - destruct b; try destruct b; auto; simpl; intuition; destruct (n <=? n0);
      destruct (n =? n0); auto. Qed.
End EX4.
```

5. Define mutually inductive types of even and odd natural numbers, such that any natural number is isomorphic to a value of one of the two types. (This problem does not ask you to prove that correspondence, though some interpretations of the task may be interesting exercises.) Write a function that computes the sum of two even numbers, such that the function type guarantees that the output is even as well. Prove that this function is commutative.

```
Module EX5.
End EX5.
```