布尔类型

提到布尔类型，人们往往会马上联想到“真”、“假”、“与”、“或”和“非”这些概念。这些概念又常常和程序语言中的布尔表达式或者命题逻辑的语义理论联系起来。许多程序语言中都可以使用布尔表达式以表示判定条件。这些判定条件又可以被进一步用于控制后续的程序执行。例如，下面是一段 C 程序：

```
unsigned int abs(int x) {
  if (x > 0) {
    return (unsigned int) x;
  }
  else {
    return - (unsigned int) x
  }
}
```

其中的布尔表达式 `x > 0` 在变量 `x` 为正时为真，在 `x` 为零或为负时为假。这个布尔表达式在上面 C 程序中控制着后续执行哪一个程序分支。程序语言中往往会包含“与”、“或”和“非”这些布尔运算，允许我们将多个简单判定条件合在一起构成一个复合判定条件。通常情况下，这些布尔表达式的结构与相应数学命题逻辑组合非常相似，例如 C 表达式中的

```
x > 0 && x < 10
```

为“真”当且仅当变量 `x` 的值在 0 到 10 之间，这类似于我们在 Coq 中写

```
x > 0 /\ x < 10
```

然而，在这一章中我们会看到，Coq 标准库中在 Coq 命题类型 `Prop` 之外额外定义了布尔类型 `bool`，其能够通过相似的结构表达上面这些数值比较和逻辑运算，但是与以 `Prop` 为类型的 Coq 命题起到不同的作用。

1 布尔类型的定义

在 Coq 中，布尔类型 `bool` 是用归纳类型定义的，它只有两个值 `true` 和 `false`。下面 Coq 代码可以用于查看 `bool` 类型在 Coq 中的定义。

```
Print bool.
```

查询结果如下：

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

2 第一个布尔函数：判定自然数相等

下面定义两个自然数的相等判定函数：

```
Fixpoint eqb (n m: nat): bool :=
  match n, m with
  | O, O => true
  | O, S _ => false
  | S _, O => false
  | S n', S m' => eqb n' m'
  end.
```

那么这个 `eqb` 函数与之前我们通常写的等号有什么区别呢？首先是类型不同。假设 `n` 与 `m` 是两个自然数，那么 `eqb n m` 的类型是 `bool` 布尔类型，而 `n = m` 是一个命题，它的类型是 `Prop` 。在 Coq 中可以用以下代码做类型检查：

```
Section TypeChecking.

Variable n: nat.
Variable m: nat.
Check eqb n m.
Check n = m.

End TypeChecking.
```

查询结果如下：

```
eqb n m: bool
n = m: Prop
```

而 `forall` 和 `exists` 都只能作用在 `Prop` 类型上。因此，下面用等号书写的是一个合法的 Coq 命题，它可以通过 Coq 类型检查。

```
Check forall n: nat, n = n.
```

相反，下面用 `eqb` 书写的式子是不合法的：

```
Fail Check (forall n, eqb n n).
```

除了类型上的区别之外，两者表示的含义也有细微的不同。Coq 中等号所表示的命题的成立条件是等号两侧的数学对象完全相同，这就是数学意义上的相等。而 `eqb` 则定义了一种判定整数相等与否的计算方法，这就像在编程语言中（例如 C 语言中）用布尔表达式判定两个整数是否相等其实也是一种计算，硬件上能直接处理的整数往往是 64 位整数、32 位整数等，比较两个 32 位整数是否相等时会把两数逐位比对从而判断它们是否相等。

当然，`eqb` 这一基于计算的自然数相等判定结果理应与数学意义上的相等吻合。这可以用下面定理概括：`eqb n m` 为真当且仅当 `n = m` 。

```
Theorem eqb_true_iff:
  forall n m: nat,
    eqb n m = true <-> n = m.
Proof.
  (** 要证明该定理，只需要对_[n]_归纳即可，不过需要注意，要归纳证明该命题对于一切
  _[m]_成立，而不是对于某一个_[m]_成立。 *)
  intros n.
  induction n; intros; destruct m; simpl.
  + tauto.
  + intuition congruence.
  + intuition congruence.
  + rewrite (IHn m).
    intuition.
Qed.
```

类似的，也可以证明 `eqb n m` 为假当且仅当 `n <> m` 。证明方法是类似的。

```
Theorem eqb_false_iff:
  forall n m: nat,
    eqb n m = false <-> n <> m.
Proof.
  intros n.
  induction n; intros; destruct m; simpl.
  + intuition congruence.
  + intuition congruence.
  + intuition congruence.
  + rewrite (IHn m).
    intuition congruence.
Qed.
```

当然，这个定理也可以利用排中律和前面的 `eqb_true_iff` 来证明，而不用再次使用归纳法。

```
Theorem eqb_false_iff_alter:
  forall n m: nat,
    eqb n m = false <-> n <> m.
Proof.
  intros.
  pose proof eqb_true_iff n m as [? ?].
  (** 由排中律可知，要么_[n = m]_要么_[n <> m]_。 *)
  pose proof classic (n = m).
  (** _[eqb n m]_要么为真要么为假，可以用_[destruct]_指令分类讨论。 *)
  destruct (eqb n m).
  + intuition congruence.
  + intuition congruence.
Qed.
```

3 布尔运算

基于 `bool` 的类型定义，也很容易定义布尔类型上的“且”、“或”和“非”。

```
Definition andb (b1 b2: bool): bool :=
  match b1 with
  | true => b2
  | false => false
  end.
```

```
Definition orb (b1 b2: bool): bool :=
  match b1 with
  | true => true
  | false => b2
  end.
```

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

不难证明 `andb` 、 `orb` 与 `negb` 为真的充分必要条件如下。

```
Theorem andb_true_iff:
  forall b1 b2: bool,
    andb b1 b2 = true <-> b1 = true /\ b2 = true.
Proof.
  intros.
  destruct b1; simpl.
  + tauto.
  + intuition congruence.
Qed.
```

```
Theorem orb_true_iff:
  forall b1 b2: bool,
    orb b1 b2 = true <-> b1 = true \/ b2 = true.
Proof.
  intros.
  destruct b1; simpl.
  + tauto.
  + intuition congruence.
Qed.
```

```
Theorem negb_true_iff:
  forall b: bool,
    negb b = true <-> b = false.
Proof.
  intros.
  destruct b; simpl.
  + intuition congruence.
  + tauto.
Qed.
```

习题 1.

请证明 `andb` 运算结果为假的充分必要条件如下。

```
Theorem andb_false_iff:
  forall b1 b2: bool,
    andb b1 b2 = false <-> b1 = false \/ b2 = false.
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```

请证明 `orb` 运算结果为假的充分必要条件如下。

```
Theorem orb_false_iff:
  forall b1 b2: bool,
    orb b1 b2 = false <-> b1 = false /\ b2 = false.
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```

请证明 `negb` 运算结果为假的充分必要条件如下。

```
Theorem negb_false_iff:
  forall b: bool,
    negb b = false <-> b = true.
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```

4 自然数的奇偶判定

除了判断自然数相等之外，还有许多性质可以写作布尔函数。例如，我们可以如下判断一个自然数的奇偶性。

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

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

这两个函数都是通过“跨两步”递归的方式定义的，它们将 `S (S n)` 的奇偶性计算归约到了 `n` 的奇偶性判定。

```
Lemma even_succ_succ:
  forall n, even (S (S n)) = even n.
Proof. intros. simpl. reflexivity. Qed.
```

```
Lemma odd_succ_succ:
  forall n, odd (S (S n)) = odd n.
Proof. intros. simpl. reflexivity. Qed.
```

不难证明，`S n` 的奇偶性总是与 `n` 相反。

```
Lemma even_odd_succ:
  forall n, even (S n) = odd n /\ odd (S n) = even n.
Proof.
  intros.
  induction n.
  + simpl.
    tauto.
  + rewrite even_succ_succ, odd_succ_succ.
    destruct IHn as [IHn_even IHn_odd].
    rewrite IHn_even, IHn_odd.
    tauto.
Qed.
```

```
Lemma even_succ:
  forall n, even (S n) = odd n.
Proof. intros. pose proof even_odd_succ n. tauto. Qed.
```

```
Lemma odd_succ:
  forall n, odd (S n) = even n.
Proof. intros. pose proof even_odd_succ n. tauto. Qed.
```

我们可以证明，根据 `even` 函数被判定为偶数的自然数，都可以写成某个数的两倍，反之亦然。这个证明我们分两步完成。首先证明必要性，这只需要对 `n` 归纳证明即可。

```
Theorem even_complete:
  forall n, even (2 * n) = true.
Proof.
  intros.
  induction n.
  + simpl.
    reflexivity.
  + (** 归纳步骤只需证明：
        - IH: even (2 * n) = true
        - 结论：even (2 * S n) = true
        同样，这只需如下变换即可完成证明：
            2 * S n = 2 * n + 2 = 2 + 2 * n = S (S (2 * n)) *)
    rewrite Nat.mul_succ_r, Nat.add_comm.
    unfold add.
    rewrite even_succ_succ.
    apply IHn.
Qed.
```

再证明充分性，这个证明需要用到跨步归纳法。

```
Theorem even_sound:
  forall n, even n = true -> exists m, n = 2 * m.
Proof.
  intros.
  (** 这里我们使用_[Nat.div2]_来辅助证明。*)
  (** 前面我们介绍过，它与_[even]_类似是“跨两步”递归定义的。*)
  exists (Nat.div2 n).
  revert H.
  (** 要用跨步归纳法，就要像下面这样归纳证明原命题对_[n]_成立，也对_[S n]_成立。*)
  assert ((even n = true -> n = 2 * Nat.div2 n) /\
          (even (S n) = true -> S n = 2 * Nat.div2 (S n))); [| tauto].
  induction n.
  + simpl.
    intuition congruence.
  + split; [tauto |].
    destruct IHn as [IHn _].
    (** 归纳步骤只需证明：
        - IHn: even n = true -> n = 2 * Nat.div2 n
        - 结论：even (S (S n)) = true -> S (S n) = 2 * Nat.div2 (S (S n)) *)
    intros.
    simpl in H.
    specialize (IHn H).
    clear H.
    (** 利用_[even]_的定义，现在只需证明：
        - IHn: n = 2 * Nat.div2 n
        - 结论：S (S n) = 2 * Nat.div2 (S (S n))
        这只需如下变换即可完成证明：
            2 * Nat.div2 (S (S n))
            = 2 * S Nat.div2 n
            = 2 * Nat.div2 n + 2
            = 2 + 2 * Nat.div2 n
            = S (S (2 * Nat.div2 n)) *)
    simpl Nat.div2.
    rewrite Nat.mul_succ_r, Nat.add_comm.
    rewrite <- IHn.
    simpl.
    reflexivity.
Qed.
```

最后，将上面两条合并起来即得到充分必要条件。

```
Theorem even_spec:
  forall n, even n = true <-> exists m, n = 2 * m.
Proof.
  intros.
  split.
  + apply even_sound.
  + intros [m ?].
    subst n.
    apply even_complete.
Qed.
```

类似的，也可以证明 `odd n = true` 当且仅当 `n` 可以写成 `S (2 * m)` 的形式。

```
Theorem odd_spec:
  forall n, odd n = true <-> exists m, n = S (2 * m).
(* 证明详见Coq源代码。 *)
```