课后阅读

1 Coq 标准库中定义的序列：list

在 Coq 中，`list X`表示一列`X`类型的元素，在标准库中，这一类型是通过 Coq 归纳类型定义的。下面介绍它的定义方式，重要函数与性质。此处为了演示这些函数的定义方式以及从定义出发构建各个性质的具体步骤重新按照标准库定义了`list`，下面的所有定义和性质都可以从标准库中找到。

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

这里，`nil`表示，这是一个空列；`cons`表示一个非空列由一个元素（头元素）和另外一列元素（其余元素）构成，因此`list`可以看做用归纳类型表示的树结构的退化情况。下面是两个整数列表`list Z`的例子。

```
Check (cons Z 3 (nil Z)).
Check (cons Z 2 (cons Z 1 (nil Z))).
```

Coq 中也可以定义整数列表的列表，`list (list Z)`。

```
Check (cons (list Z) (cons Z 2 (cons Z 1 (nil Z))) (nil (list Z))).
```

我们可以利用 Coq 的隐参数机制与`Arguments`指令，让我们省略`list`定义中的类型参数。

```
Arguments nil {X}.
Arguments cons {X} _ _.
```

例如，我们可以重写上面这些例子:

```
Check (cons 3 nil).
Check (cons 2 (cons 1 nil)).
Check (cons (cons 2 (cons 1 nil)) nil).
```

Coq 标准库还提供了一些`Notation`让`list`相关的描述变得更简单。

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

下面用同一个整数序列的三种表示方法来演示利用这些`Notation`的使用方法，大家不需要完全理解上面这些`Notation`声明的细节。

```
Definition mylist1 := 1 :: (2 :: (3 :: nil)).
Definition mylist2 := 1 :: 2 :: 3 :: nil.
Definition mylist3 := [1; 2; 3].
```

下面介绍一些常用的关于`list`函数。

函数`app`表示序列的连接。

```
Fixpoint app {X: Type} (l1 l2: list X): list X :=
  match l1 with
  | nil      => l2
  | cons h t => cons h (app t l2)
  end.
```

在 Coq 中一般可以用`++`表示`app`。

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

函数`rev`表示对序列取反。

```
Fixpoint rev {A} (l: list A) : list A :=
  match l with
  | nil    => nil
  | h :: t => rev t ++ [h]
  end.
```

下面是一些例子关于`app`和`rev`的例子。

```
Example test_app1: [1; 2; 3] ++ [4; 5] = [1; 2; 3; 4; 5].
Proof. reflexivity.  Qed.
```

```
Example test_app2: [2] ++ [3] ++ [] ++ [4; 5] = [2; 3; 4; 5].
Proof. reflexivity.  Qed.
```

```
Example test_app3: [1; 2; 3] ++ nil = [1; 2; 3].
Proof. reflexivity.  Qed.
```

```
Example test_rev:  rev [1; 2; 3] = [3; 2; 1].
Proof. reflexivity.  Qed.
```

下面是一些关于`app`和`rev`的重要性质，它们的证明留作习题。

习题 1.

```
Theorem app_assoc: forall A (l1 l2 l3: list A),
  (l1 ++ l2) ++ l3 = l1 ++ (l2 ++ l3).
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```

习题 2.

```
Theorem app_nil_r : forall A (l : list A),
  l ++ [] = l.
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```

习题 3.

```
Theorem rev_app_distr: forall A (l1 l2 : list A),
  rev (l1 ++ l2) = rev l2 ++ rev l1.
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```

习题 4.

```
Theorem rev_involutive : forall A (l : list A),
  rev (rev l) = l.
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```

如果熟悉函数式编程，不难发现，上面的`rev`定义尽管在数学是简洁明确的，但是其计算效率是比较低的。相对而言，利用下面的`rev_append`函数进行计算则效率较高。

```
Fixpoint rev_append {A} (l l': list A): list A :=
  match l with
  | []     => l'
  | h :: t => rev_append t (h :: l')
  end.
```

下面分两步证明`rev`与`rev_append`之间的相关性。

习题 5.

```
Lemma rev_append_rev: forall A (l l': list A),
  rev_append l l' = rev l ++ l'.
(* 请在此处填入你的证明，以_[Qed]_结束。 *)

Theorem rev_alt: forall A (l: list A),
  rev l = rev_append l [].
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```

下面的`map`函数表示对一个`list`中的所有元素统一做映射。

```
Fixpoint map {X Y: Type} (f: X -> Y) (l: list X): (list Y) :=
  match l with
  | []     => []
  | h :: t => f h :: map f t
  end.
```

下面是一些例子：

```
Example test_map1: map (fun x => x - 2) [7; 5; 7] = [5; 3; 5].
Proof. reflexivity. Qed.
```

```
Example test_map2: map (fun x => x * x) [2; 1; 5] = [4; 1; 25].
Proof. reflexivity. Qed.
```

```
Example test_map3: map (fun x => [x]) [0; 1; 2; 3] = [[0]; [1]; [2]; [3]].
Proof. reflexivity. Qed.
```

最后两个`map`性质的证明留作习题。

习题 6.

```
Theorem map_app: forall X Y (f: X -> Y) (l l': list X),
  map f (l ++ l') = map f l ++ map f l'.
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```

习题 7.

```
Theorem map_rev : forall X Y (f: X -> Y) (l: list X),
  map f (rev l) = rev (map f l).
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```

2 Coq 标准库中定义的自然数：nat

在 Coq 中，许多数学上的集合可以用归纳类型定义。例如，Coq 中自然数的定义就是最简单的归纳类型之一。

下面 Coq 代码可以用于查看`nat`在 Coq 中的定义。

```
Print nat.
```

查询结果如下。

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

可以看到，自然数集合的归纳定义可以看做`list`进一步退化的结果。下面我们在 Coq 中定义自然数的加法，并且也试着证明一条基本性质：加法交换律。

由于 Coq 的标准库中已经定义了自然数以及自然数的加法。我们开辟一个`NatDemo`来开发我们自己的定义与证明。以免与 Coq 标准库的定义相混淆。

```
Module NatDemo.
```

先定义自然数`nat`。

```
Inductive nat :=
| O: nat
| S (n: nat): nat.
```

再定义自然数加法。

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

下面证明加法交换律。

```
Theorem add_comm: forall n m,
  add n m = add m n.
Proof.
  intros.
  induction n.
```

证明到此处，我们发现我们需要首先证明`n + 0 = n`这条性质，我们先终止交换律的证明，而先证明这条引理。

```
Abort.
```

```
Lemma add_0_r: forall n, add n 0 = n.
Proof.
  intros.
  induction n; simpl.
  + reflexivity.
  + rewrite IHn.
    reflexivity.
Qed.
```

```
Theorem add_comm: forall n m,
  add n m = add m n.
Proof.
  intros.
  induction n; simpl.
  + rewrite add_0_r.
    reflexivity.
  +
```

证明到此处，我们发现我们需要还需要证明关于`m + (S n)`相关的性质。

```
Abort.
```

```
Lemma add_S_r: forall n m,
  add n (S m) = S (add n m).
Proof.
  intros.
  induction n; simpl.
  + reflexivity.
  + rewrite IHn.
    reflexivity.
Qed.
```

现在已经可以在 Coq 中完成加法交换律的证明了。

```
Theorem add_comm: forall n m,
  add n m = add m n.
Proof.
  intros.
  induction n; simpl.
  + rewrite add_0_r.
    reflexivity.
  + rewrite add_S_r.
    rewrite IHn.
    reflexivity.
Qed.
```

```
End NatDemo.
```

上面证明的加法交换律在 Coq 标准库中已有证明，其定理名称是`Nat.add_comm`。

```
Check Nat.add_comm.
```