Coq 归纳类型

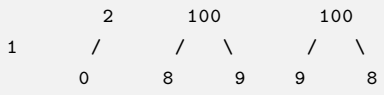
1 关于等式的证明

Coq 证明脚本 1. 利用等式做证明的 rewrite 指令. 如果 `H` 是具有形式 `a = b` 定理或证明前提，`rewrite H` 可以将待证明结论中的 `a` 替换成 `b`，`rewrite H in H0` 也可以将前提 `H0` 中的 `a` 替换成 `b`。

2 用 Coq 归纳类型定义二叉树

下面 Coq 代码定义了节点上有整数标号的二叉树。

```
Inductive tree: Type :=
| Leaf: tree
| Node (l: tree) (v: Z) (r: tree): tree.
```



Coq 中，我们往往可以使用递归函数定义归纳类型元素的性质。Coq 中定义递归函数时使用的关键字是 `Fixpoint`。下面两个定义通过递归定义了二叉树的高度和节点个数。

```
Fixpoint tree_height (t: tree): Z :=
  match t with
  | Leaf => 0
  | Node l v r => Z.max (tree_height l) (tree_height r) + 1
  end.
```

```
Fixpoint tree_size (t: tree): Z :=
  match t with
  | Leaf => 0
  | Node l v r => tree_size l + tree_size r + 1
  end.
```

Coq 中也可以定义树到树的函数。下面的 `tree_reverse` 函数把二叉树进行了左右翻转。

```
Fixpoint tree_reverse (t: tree): tree :=
  match t with
  | Leaf => Leaf
  | Node l v r => Node (tree_reverse r) v (tree_reverse l)
  end.
```

归纳类型有几条基本性质。(1) 归纳定义规定了一种分类方法，以 `tree` 类型为例，一棵二叉树 `t` 要么是 `Leaf` ，要么具有形式 `Node l v r` ；(2) 以上的分类之间是互斥的，即无论 `l` 、`v` 与 `r` 取什么值，`Leaf` 与 `Node l v r` 都不会相等；(3) `Node` 这样的构造子是函数也是单射。这三条性质对应了 Coq 中的三条证明指令：`destruct` 、`discriminate` 与 `injection` 。

3 结构归纳法

数学归纳法说的是：如果我们要证明某性质 `P` 对于任意自然数 `n` 都成立，那么我可以将证明分为如下两步：

- 奠基步骤：证明 `P 0` 成立；
- 归纳步骤：证明对于任意自然数 `n` ，如果 `P n` 成立，那么 `P (n + 1)` 也成立。

对二叉树的归纳证明与上面的数学归纳法稍有不同。具体而言，如果我们要证明某性质 `P` 对于一切二叉树 `t` 都成立，那么我们只需要证明以下两个结论：

- 奠基步骤：证明 `P Leaf` 成立；
- 归纳步骤：证明对于任意二叉树 `l` `r` 以及任意整数标签 `n` ，如果 `P l` 与 `P r` 都成立，那么 `P (Node l n r)` 也成立。

这样的证明方法就成为结构归纳法。

4 加强归纳

下面证明 `tree_reverse` 是一个单射。这个引理的 Coq 证明需要我们特别关注：真正需要归纳证明的结论是什么？如果选择对 `t1` 做结构归纳，那么究竟是归纳证明对于任意 `t2` 上述性质成立，还是归纳证明对于某“特定”的 `t2` 上述性质成立？如果我们按照之前的 Coq 证明习惯，用 `intros` 与 `induction t1` 两条指令开始证明，那就表示用归纳法证明一条关于“特定” `t2` 的性质。

```
Lemma tree_reverse_inj: forall t1 t2,
  tree_reverse t1 = tree_reverse t2 ->
  t1 = t2.
Proof.
  intros.
  induction t1 as [| t11 IHt11 v1 t12 IHt12].
  + destruct t2 as [| t21 v2 t22].
    (** 奠基步骤的证明可以通过对_[t2]_的分类讨论完成。*)
    - (** 如果_[t2]_是空树，那么结论是显然的。*)
      reflexivity.
    - (** 如果_[t2]_是非空树，那么前提_[H]_就能导出矛盾。*)
      simpl in H.
      (** 化简后，前提为：
          - Leaf = Node (tree_reverse t22) v (tree_reverse t21)
          这能直接推出矛盾。*)
      discriminate H.
      (** 当然，在这个证明中，由于之后的_[discriminate]_指令也会完成自动化简，前面
          的一条_[simpl]_指令其实是可以省略的。*)
  + (** 进入归纳步骤的证明时，证明已经无法继续。此时证明目标中的前提与结论为：
      - H: tree_reverse (Node t11 v1 t12) = tree_reverse t2
      - IHt11: tree_reverse t11 = tree_reverse t2 ->
               t11 = t2
      - IHt12: tree_reverse t12 = tree_reverse t2 ->
               t12 = t2
      - 结论：Node t11 v t12 = t2
      我们需要使用的归纳假设并非关于原_[t2]_值的性质。*)
Abort.
```

正确的证明方法是用归纳法证明一条对于一切 `t2` 的结论。

```
Lemma tree_reverse_inj: forall t1 t2,
  tree_reverse t1 = tree_reverse t2 ->
  t1 = t2.
Proof.
  intros t1.
  (** 上面这条_[intros 1t]_指令可以精确地将_[t1]_放到证明目标的前提中，同时却将
      _[t2]_保留在待证明目标的结论中。*)
  induction t1 as [| t11 IHt11 v1 t12 IHt12].
  + (** 现在的奠基步骤需要证明，对于任意_[t2]_，
        - 如果_[tree_reverse Leaf = tree_reverse t2]_
        - 那么_[Leaf = t2]_ *)
    simpl. intros.
    destruct t2 as [| t21 v2 t22].
    - reflexivity.
    - discriminate H.
  + (** 现在的归纳步骤中，归纳假设为，
        - IHt11:
            forall t2: tree,
              tree_reverse t11 = tree_reverse t2 ->
              t11 = t2
        - IHt12:
            forall t2: tree,
              tree_reverse t12 = tree_reverse t2 ->
              t12 = t2 *)
    simpl. intros.
    (** 接下去对_[t2]_分类讨论，排除掉_[t2]_为空树的情况。*)
    destruct t2 as [| t21 v2 t22].
    - discriminate H.
    - injection H as H2 Hv H1.
      (** 现在，由原先的前提_[tree_reverse t1 = tree_reverse t2]_我们就知道：
          - H1: tree_reverse t11 = tree_reverse t21
          - Hv: v1 = v2
          - H2: tree_reverse t12 = tree_reverse t22
          下面就只需要使用归纳假设就可以证明_[t1 = t2]_了，即
          - Node t11 v1 t12 = Node t21 v2 t22。*)
      rewrite (IHt11 t21 H1).
      rewrite (IHt12 t22 H2).
      rewrite Hv.
      reflexivity.
Qed.
```