第17章：Agda的基本概念(2)

胡振江，张伟
信息学院计算机科学技术系

2021年12月1日

北京大學
PEKING UNIVERSITY

不等式关系，等式判定关系

```
data _/=_ : Nat -> Nat -> Set where
  z/=s : {n : Nat} -> zero /= suc n
  s/=z : {n : Nat} -> suc n /= zero
  s/=s : {m n : Nat} -> m /= n -> suc m /= suc n

data Equal? (n m : Nat) : Set where
  eq : n == m -> Equal? n m
  neq : n /= m -> Equal? n m
```

练习：定义函数equal?，判定两个自然数是否相等：

equal? : (n m : Nat) -> Equal? n m

（注：这个函数实际上是给定了一个判定证明）

```
equal? : (n m : Nat) -> Equal? n m
equal? zero zero = eq refl
equal? zero (suc m) = neq z/=s
equal? (suc n) zero = neq s/=z
equal? (suc n) (suc m) with equal? n m
equal? (suc n) (suc .n) | eq refl = eq refl
equal? (suc n) (suc m)  | neq p = neq (s/=s p)
```

子序列关系

```
infix 20 _⊆_

data _⊆_ {A : Set} : List A -> List A -> Set where
stop : [] ⊆ []
drop : forall {xs y ys} -> xs ⊆ ys -> xs ⊆ y :: ys
keep : forall {x xs ys} -> xs ⊆ ys -> x :: xs ⊆ x :: ys
```

```
lem-filter : {A : Set}(p : A -> Bool)(xs : List A)
             -> filter p xs ⊆ xs
lem-filter p [] = stop
lem-filter p (x :: xs) with p x
... | true = keep (lem-filter p xs)
... | false = drop (lem-filter p xs)
```

... 代表 with 前面的部分 `lem-filter p (x :: xs)`

等式理论

```
infix 4 _≡_
data _≡_ {A : Set} (x : A) : A → Set where
    refl : x ≡ x
```

```
sym : ∀ {A : Set} {x y : A}
    → x ≡ y
      -----
    → y ≡ x
sym refl = refl
```

```
trans : ∀ {A : Set} {x y z : A}
    → x ≡ y
    → y ≡ z
     ------
    → x ≡ z
```

```
cong : ∀ {A B : Set} (f : A → B) {x y : A}
    → x ≡ y
      ----------
    → f x ≡ f y
```

```
cong-app : ∀ {A B : Set} {f g : A → B}
    → f ≡ g
      ---------------------
    → ∀ (x : A) → f x ≡ g x
```

```
subst : ∀ {A : Set} {x y : A} (P : A → Set)
    → x ≡ y
      ----------
    → P x → P y
```

等式推理

```
infix     1 begin_
Infixr    2 _≡⟨⟩_ _≡⟨_⟩_
infix     3 _∎
```

```
begin_ : ∀ {x y : A}
    → x ≡ y
      -----
    → x ≡ y
begin x≡y = x≡y
```

推理开始

```
_≡⟨⟩_ : ∀ (x : A) {y : A}
    → x ≡ y
      -----
    → x ≡ y
x ≡⟨⟩ x≡y = x≡y
```

从x新开始
经过等式变换

```
_≡⟨_⟩_ : ∀ (x : A) {y z : A}
    → x ≡ y
    → y ≡ z
      -----
    → x ≡ z
x ≡⟨ x≡y ⟩ y≡z = trans x≡y y≡z
```

从x开始经过
等式传递变换

```
_∎ : ∀ (x : A)
      -----
    → x ≡ x
x ∎ = refl
```

证明结束

```
trans′ : ∀ {A : Set} {x y z : A}
    → x ≡ y
    → y ≡ z
      -----
    → x ≡ z
trans′ {A} {x} {y} {z} x≡y y≡z =
  begin
    x
  ≡⟨ x≡y ⟩
    y
  ≡⟨ y≡z ⟩
    z
  ∎
```

```
trans′ : ∀ {A : Set} {x y z : A}
    → x ≡ y
    → y ≡ z
      -----
    → x ≡ z
trans′ {A} {x} {y} {z} x≡y y≡z =
  begin
     x
  ≡⟨ x≡y ⟩
     y
  ≡⟨ y≡z ⟩
     z
  ∎
```

自然数上的证明例

```
data ℕ : Set where
    zero : ℕ
    suc : ℕ → ℕ

_+_ : ℕ → ℕ → ℕ
zero + n = n
(suc m) + n = suc (m + n)

postulate
    +-identity : ∀ (m : ℕ) → m + zero ≡ m
    +-suc : ∀ (m n : ℕ) → m + suc n ≡ suc (m + n)
```

如何通过等式变换证明加法的交换性?

```
+-comm : ∀ (m n : ℕ) → m + n ≡ n + m
```

```
+-comm m zero =
 begin
   m + zero
 ≡⟨ +-identity m ⟩
   m
 ≡⟨⟩
   zero + m
 ∎
+-comm m (suc n) =
 begin
   m + suc n
 ≡⟨ +-suc m n ⟩
   suc (m + n)
 ≡⟨ cong suc (+-comm m n) ⟩
   suc (n + m)
 ≡⟨⟩
  suc n + m
 ∎
```

序列上的证明例

```
data List (A : Set) : Set where
    [] : List A
    _::_ : A -> List A -> List A

foldr : {A B : Set} -> (A -> B -> B) -> B -> List A -> B
foldr f e [] = e
foldr f e (x :: xs) = f x (foldr f e xs)

_=♭_ : {A B : Set} -> (A -> B) -> (A -> B) -> Set
f =♭ g = ∀ x -> f x ≡ g x
```

如何证明以下性质？

```
foldr-universal : ∀ {A B} (h : List A -> B) f e
    -> (h [] ≡ e)
    -> (∀ x xs -> h (x :: xs) ≡ f x (h xs))
       ------------------------------------
    -> h =♭ foldr f e
```

```
foldr-universal : ∀ {A B} (h : List A -> B) f e
   -> (h [] ≡ e)
   -> (∀ x xs -> h (x :: xs) ≡ f x (h xs))
      ---------------------------------------
   -> h ≐ foldr f e
foldr-universal h f e base step [] =
   h []
 ≡⟨ base ⟩
   foldr f e []
 ∎
foldr-universal h f e base step (x :: xs) =
   h (x :: xs)
 ≡⟨ step x xs ⟩
   f x (h xs)
 ≡⟨ cong (f x) (foldr-universal h f e base step xs) ⟩
   f x (foldr f e xs)
 ≡⟨⟩
   foldr f e (x :: xs)
 ∎
```

作业

17-4.证明序列的包含关系_⊆_是自反的而且传递的。

17-5. 利用等式推理证明下面两个性质：

```
+-identity : ∀ (m : ℕ) → m + zero ≡ m

+-suc : ∀ (m n : ℕ) → m + suc n ≡ suc (m + n)
```