第18章：序列理论概述(1)

胡振江，张伟

信息学院计算机科学技术系

2021年12月8日

序列理论 (Theory of Lists)

- BMF (Bird Meertens Formalism)

A set of combinators (higher order functions on Lists)	+	A set of rules/laws (properties)

⬇

Calculational Functional Programming
(Constructive Functional Programming)

参考资料： Richard Bird, Lecture Notes on Constructive Functional Programming, Technical Monograph PRG-69, Oxford University, 1988.

复习

- 我们已经简单地讨论了在agda上的
  - 自然数的定义和性质
  - 布尔值的定义和性质
  - 等式推理方法
  - 简单的序列上的函数定义和推理

```
open import Data.Nat.Base as ℕ
         using (ℕ; zero; suc; _+_; _*_ ; _≤_ ; _>_ ; s≤s)
open import Data.Nat.Properties
         using (*-assoc)

open import Data.Bool.Base as Bool
         using (Bool; false; true; not; _∧_; _∨_; if_then_else_)

import Relation.Binary.PropositionalEquality as Eq
open Eq  using (_≡_; refl; trans; sym; cong; cong-app; subst)
open Eq.≡-Reasoning using (begin_; _≡⟨⟩_; step-≡; _∎)
```

函数

- 外延相等

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

函数

- 恒等函数是函数合成的左单位元

```
∘-identity-l : ∀ {A B : Set} (f : A → B) → id ∘ f ≡ f
∘-identity-l {A} f = extensionality lem
  where
    lem : ∀ (x : A) → (id ∘ f) x ≡ f x
    lem x =  begin
               (id ∘ f) x
             ≡⟨⟩
               id (f x)
             ≡⟨⟩
               f x
             ∎
```

函数

- 恒等函数是函数合成的右单位元

```
∘-identity-r : ∀ {A B : Set} (f : A → B) → f ∘ id ≡ f
∘-identity-r {A} f = extensionality lem
  where
    lem : ∀ (x : A) → (f ∘ id) x ≡ f x
    lem x =  begin
               (f ∘ id) x
             ≡⟨⟩
               f (id x)
             ≡⟨⟩
               f x
             ∎
```

函数

- 函数合成是可结合的

```
∘-assoc : ∀ {A B C D : Set} (f : A → B) (g : B → C) (h : C → D)
          → h ∘ (g ∘ f) ≡ (h ∘ g) ∘ f
∘-assoc {A} f g h = extensionality lem
  where
    lem : ∀ (x : A) → (h ∘ (g ∘ f)) x ≡ ((h ∘ g) ∘ f) x
    lem x = begin
            (h ∘ (g ∘ f)) x
          ≡⟨⟩
            h ((g ∘ f) x)
          ≡⟨⟩
            h (g (f x))
          ≡⟨⟩
            (h ∘ g) (f x)
          ≡⟨⟩
            ((h ∘ g) ∘ f) x
          ∎
```

函数

练习：证明下面关于程序合成的性质。

```
∘-assoc' : ∀ {B C D : Set} (g : B → C) (h : C → D)
         → (h ∘_) ∘ (g ∘_) ≡ ((h ∘ g) ∘_)
```

```
∘-assoc' {B} g h = extensionality lem
  where
    lem : ∀ {A : Set} (f : A → B) → ((h ∘_) ∘ (g ∘_)) f ≡ ((h ∘ g) ∘_) f
    lem f = begin
              ((h ∘_) ∘ (g ∘_)) f
            ≡⟨⟩
              (h ∘_) (g ∘ f)
            ≡⟨⟩
              h ∘ (g ∘ f)
            ≡⟨ ∘-assoc f g h ⟩
              (h ∘ g) ∘ f
            ≡⟨⟩
              ((h ∘ g) ∘_) f
            ∎
```

序列 List

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

infixr 5 _::_
...

_ : List ℕ
_ = 0 :: 1 :: 2 :: []
```

序列上的函数定义

序列链接函数

```
_++_ : ∀ {A : Set} → List A → List A → List A
[]       ++ ys  =  ys
(x :: xs) ++ ys  =  x :: (xs ++ ys)

_ : 0 :: 1 :: 2 :: [] ++ 3 :: 4 :: [] ≡ 0 :: 1 :: 2 :: 3 :: 4 :: []
_ =
  begin
    0 :: 1 :: 2 :: [] ++ 3 :: 4 :: []
  ≡⟨⟩
    0 :: (1 :: 2 :: [] ++ 3 :: 4 :: [])
  ≡⟨⟩
    0 :: 1 :: (2 :: [] ++ 3 :: 4 :: [])
  ≡⟨⟩
    0 :: 1 :: 2 :: ([] ++ 3 :: 4 :: [])
  ≡⟨⟩
    0 :: 1 :: 2 :: 3 :: 4 :: []
  ∎
```

序列上的函数定义及其性质

计算序列长度函数

```
# : ∀ {A : Set} → List A → ℕ
# []          =  zero
# (x :: xs)   =  suc (# xs)
```

序列反转函数

```
reverse : ∀ {A : Set} → List A → List A
reverse []          =  []
reverse (x :: xs)   =  reverse xs ++ (x :: [])
```

序列上的函数定义及其性质

```
#-++ : ∀ {A : Set} (xs ys : List A)
  → # (xs ++ ys) ≡ # xs + # ys
#-++ {A} [] ys =
  begin
    # ([] ++ ys)
  ≡⟨⟩
    # ys
  ≡⟨⟩
    # {A} [] + # ys
  ∎
#-++ (x ∷ xs) ys =
  begin
    # ((x ∷ xs) ++ ys)
  ≡⟨⟩
    suc (# (xs ++ ys))
  ≡⟨ cong suc (#-++ xs ys) ⟩
    suc (# xs + # ys)
  ≡⟨⟩
    # (x ∷ xs) + # ys
  ∎
```

高阶函数 map

定义

```
map : ∀ {A B : Set} → (A → B) → List A → List B
map f [] = []
map f (x ∷ xs) = f x ∷ map f xs
```

分配律

```
map-compose : ∀ {A B C : Set} → (f : A → B) → (g : B → C)
              → map g ∘ map f ≡ map (g ∘ f)

map-compose f g = extensionality (map-compose-p f g)
```

高阶函数 map

```
map-compose-p : ∀ {A B C : Set} (f : A → B) (g : B → C) (x : List A) →
              (map g ∘ map f) x ≡ map (g ∘ f) x
map-compose-p f g [] =
  begin
    (map g ∘ map f) []
  ≡⟨⟩
    map g (map f [])
  ≡⟨⟩
    map g []
  ≡⟨⟩
    []
  ≡⟨⟩
    map (g ∘ f) []
  ∎
```

高阶函数 map

```
map-compose-p f g (x :: xs) =
  begin
    (map g ∘ map f) (x :: xs)
  ≡⟨⟩
    map g (map f (x :: xs))
  ≡⟨⟩
    map g (f x :: map f xs)
  ≡⟨⟩
    g (f x) :: map g (map f xs)
  ≡⟨ cong (g (f x) ::_) (map-compose-p f g xs) ⟩
    g (f x) :: map (g ∘ f) xs
  ≡⟨⟩
    (g ∘ f) x :: map (g ∘ f) xs
  ≡⟨⟩
    map (g ∘ f) (x :: xs)
  ∎
```

高阶函数：foldr, foldl

```
foldr : ∀ {A B : Set} → (A → B → B) → B → List A → B
foldr _⊗_ e [] = e
foldr _⊗_ e (x ∷ xs) = x ⊗ foldr _⊗_ e xs

foldl : ∀ {A B : Set} → (B → A → B) → B → List A → B
foldl _⊗_ e [] = e
foldl _⊗_ e (x ∷ xs) = foldl _⊗_ (e ⊗ x) xs
```

高阶函数：scanr, scanl

```
scanr : ∀ {A B : Set} → (A → B → B) → B → List A → List B
scanr _⊗_ e []          =  e ∷ []
scanr _⊗_ e (x ∷ xs) with scanr _⊗_ e xs
... | y ∷ ys = x ⊗ y ∷ (y ∷ ys)
... | []      = []    -- non-excutable branch

scanl :  ∀ {A B : Set} → (B → A → B) → B → List A → List B
scanl _⊗_ e [] = e ∷ []
scanl _⊗_ e (x ∷ xs) = e ∷ scanl _⊗_ (e ⊗ x) xs
```

Monoid 代数结构

```
record IsMonoid {A : Set} (_⊗_ : A → A → A) (e : A) : Set where
  field
    assoc : ∀ (x y z : A) → (x ⊗ y) ⊗ z ≡ x ⊗ (y ⊗ z)
    identityˡ : ∀ (x : A) → e ⊗ x ≡ x
    identityʳ : ∀ (x : A) → x ⊗ e ≡ x
```

```
open IsMonoid

++-monoid : ∀ {A : Set} → IsMonoid {List A} _++_ []
++-monoid =
  record
    { assoc = ++-assoc
    ; identityˡ = ++-identityˡ
    ; identityʳ = ++-identityʳ
    }
```

Monoid 代数结构

```
infix 5 _⇑_

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

postulate
  ⇑-assoc : ∀ (m n p : ℕ) → (m ⇑ n) ⇑ p ≡ m ⇑ (n ⇑ p)
  ⇑-identity-l : ∀ (n : ℕ) → zero ⇑ n ≡ n
  ⇑-identity-r  : ∀ (n : ℕ) → n ⇑ zero ≡ n

⇑-monoid : IsMonoid  _⇑_ zero
⇑-monoid =
  record
    { assoc = ⇑-assoc
    ; identityˡ = ⇑-identity-l
    ; identityʳ = ⇑-identity-r
    }
```

Maximum Segment Sum 的问题描述

```
[-]_ : ∀ {A : Set} (x : A) → List A
[-] x = x ∷ []

inits : ∀ {A : Set} → List A → List (List A)
inits = scanl _++_ [] ∘ map ([-]_)

tails : ∀ {A : Set} → List A → List (List A)
tails = scanr _++_ [] ∘ map ([-]_)

segs : ∀ {A : Set} → List A → List (List A)
segs = foldr _++_ [] ∘ map tails ∘ inits

sum : List ℕ → ℕ
sum = foldr _+_ zero

max : List ℕ → ℕ
max = foldr _↑_ zero

mss : List ℕ → ℕ
mss = max ∘ map sum ∘ segs
```

序列函数的性质证明例1

```
foldr-monoid : ∀ {A : Set} (_⊗_ : A → A → A) (e : A)
  → IsMonoid _⊗_ e
    ------------------
  → ∀ (xs : List A) (y : A) → foldr _⊗_ y xs ≡ foldr _⊗_ e xs ⊗ y
```

序列函数的性质证明例1

```
foldr-monoid _⊗_ e ⊗-monoid [] y =
  begin
    foldr _⊗_ y []
  ≡⟨⟩
    y
  ≡⟨ sym (identityˡ ⊗-monoid y) ⟩
    (e ⊗ y)
  ≡⟨⟩
    foldr _⊗_ e [] ⊗ y
  ∎
```

序列函数的性质证明例1

```
foldr-monoid _⊗_ e ⊗-monoid (x ∷ xs) y =
  begin
    foldr _⊗_ y (x ∷ xs)
  ≡⟨⟩
    x ⊗ (foldr _⊗_ y xs)
  ≡⟨ cong (x ⊗_) (foldr-monoid _⊗_ e ⊗-monoid xs y) ⟩
    x ⊗ (foldr _⊗_ e xs ⊗ y)
  ≡⟨ sym (assoc ⊗-monoid x (foldr _⊗_ e xs) y) ⟩
    (x ⊗ foldr _⊗_ e xs) ⊗ y
  ≡⟨⟩
    foldr _⊗_ e (x ∷ xs) ⊗ y
  ∎
```

序列函数的性质证明例2

```
postulate
  foldr-++ : ∀ {A : Set} (_⊗_ : A → A → A) (e : A) (xs ys : List A) →
    foldr _⊗_ e (xs ++ ys) ≡ foldr _⊗_ (foldr _⊗_ e ys) xs

foldr-monoid-++ : ∀ {A : Set} (_⊗_ : A → A → A) (e : A) → IsMonoid _⊗_ e →
  ∀ (xs ys : List A) → foldr _⊗_ e (xs ++ ys) ≡ foldr _⊗_ e xs ⊗ foldr _⊗_ e ys
foldr-monoid-++ _⊗_ e monoid-⊗ xs ys =
  begin
    foldr _⊗_ e (xs ++ ys)
  ≡⟨ foldr-++ _⊗_ e xs ys ⟩
    foldr _⊗_ (foldr _⊗_ e ys) xs
  ≡⟨ foldr-monoid _⊗_ e monoid-⊗ xs (foldr _⊗_ e ys) ⟩
    foldr _⊗_ e xs ⊗ foldr _⊗_ e ys
  ∎
```

作业

18-1.证明_++_满足结合律，而且[]是它的单位元。

++-assoc : ∀ {A : Set} (xs ys zs : List A)

→ (xs ++ ys) ++ zs ≡ xs ++ (ys ++ zs)

++-identity^l : ∀ {A : Set} (xs : List A) → [] ++ xs ≡ xs

++-identity^r : ∀ {A : Set} (xs : List A) → xs ++ [] ≡ xs

作业

18-2.证明以下性质：

foldr-++ : ∀ {A : Set} (_⊗_ : A → A → A) (e : A) (xs ys : List A) →
foldr _⊗_ e (xs ++ ys) ≡ foldr _⊗_ (foldr _⊗_ e ys) xs

18-3. 证明 reverse ∘ reverse = id, 其中 reverse的定义如下。

```
reverse : ∀ {A : Set} → List A → List A
reverse []          =  []
reverse (x :: xs)  =  reverse xs ++ (x :: [])
```