

第23章：序列理论概述 (in Agda)

胡振江，张伟

信息学院计算机科学技术系

2023年12月13日

序列理论 (Theory of Lists)

- BMF (Bird Meertens Formalism)



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  $\mathbb{N}$ 
  using ( $\mathbb{N}$ ; zero; suc;  $_+_$ ;  $_*_$  ;  $_ \leq _$  ;  $_ > _$  ;  $s \leq s$ )
open import Data.Nat.Properties
  using ( $*-assoc$ )

open import Data.Bool.Base as Bool
  using (Bool; false; true; not;  $_ \wedge _$ ;  $_ \vee _$ ; if_then_else_)

import Relation.Binary.PropositionalEquality as Eq
open Eq using ( $_ \equiv _$ ; refl; trans; sym; cong; cong-app; subst)
open Eq. $\equiv$ -Reasoning using (begin_;  $_ \equiv \{ \}_$ ; step- $\equiv$ ;  $_ \blacksquare$ )
```

函数

- 外延相等

```
postulate
```

```
  extensionality :  $\forall \{A B : \text{Set}\} \{f g : A \rightarrow B\}$ 
```

```
  →  $(\forall (x : A) \rightarrow f\ x \equiv g\ x)$ 
```

```
  -----  
  →  $f \equiv g$ 
```

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

```
_++_ :  $\forall \{A : \text{Set}\} \rightarrow \text{List } A \rightarrow \text{List } A \rightarrow \text{List } A$   
[] ++ ys = ys  
(x :: xs) ++ ys = x :: (xs ++ ys)  
  
_ : 0 :: 1 :: 2 :: [] ++ 3 :: 4 :: []  $\equiv$  0 :: 1 :: 2 :: 3 :: 4 :: []  
=  
begin  
  0 :: 1 :: 2 :: [] ++ 3 :: 4 :: []  
 $\equiv$  {}  
  0 :: (1 :: 2 :: [] ++ 3 :: 4 :: [])  
 $\equiv$  {}  
  0 :: 1 :: (2 :: [] ++ 3 :: 4 :: [])  
 $\equiv$  {}  
  0 :: 1 :: 2 :: ([] ++ 3 :: 4 :: [])  
 $\equiv$  {}  
  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

```
# :  $\forall \{A : \text{Set}\} \rightarrow \text{List } A \rightarrow \mathbb{N}$   
# [] = zero  
# (x :: xs) = suc (# xs)
```

序列反转函数

```
reverse :  $\forall \{A : \text{Set}\} \rightarrow \text{List } A \rightarrow \text{List } A$   
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 :  $\forall \{A B : \text{Set}\} \rightarrow (A \rightarrow B) \rightarrow \text{List } A \rightarrow \text{List } B$   
map f [] = []  
map f (x :: xs) = f x :: map f xs
```

分配律

```
map-compose :  $\forall \{A B C : \text{Set}\} \rightarrow (f : A \rightarrow B) \rightarrow (g : B \rightarrow C)$   
               $\rightarrow \text{map } g \circ \text{map } f \equiv \text{map } (g \circ f)$   
map-compose f g = extensionality (map-compose-p f g)
```

高阶函数 map

```
map-compose-p :  $\forall \{A B C : \text{Set}\} (f : A \rightarrow B) (g : B \rightarrow C) (x : \text{List } A) \rightarrow$   
| | | | | (map g  $\circ$  map f) x  $\equiv$  map (g  $\circ$  f) x  
map-compose-p f g [] =  
| begin  
| (map g  $\circ$  map f) []  
 $\equiv$  ( $\langle$ )  
| map g (map f [])  
 $\equiv$  ( $\langle$ )  
| map g []  
 $\equiv$  ( $\langle$ )  
| []  
 $\equiv$  ( $\langle$ )  
| map (g  $\circ$  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
```


Hom: Map/Reduce

```
reduce : ∀{A : Set} → (_⊕_ : A → A → A) → (e : A)
        → IsMonoid _⊕_ e → List A → A
reduce _⊕_ e _ [] = e
reduce _⊕_ e p (x :: xs) = x ⊕ reduce _⊕_ e p xs
```

```
hom : ∀{A B : Set} (_⊕_ : B → B → B) (e : B)
      (p : IsMonoid _⊕_ e) → (A → B)
      → List A → B
hom _⊕_ e p f = reduce _⊕_ e p ∘ map f
```

Promotion Laws

```
map-promotion :  
  ∀{A B : Set} (f : A → B)  
  → map f ∘ flatten ≡ flatten ∘ map (map f)  
  
reduce-promotion :  
  ∀{A : Set} (_⊕_ : A → A → A) (e : A)  
  (p : IsMonoid _⊕_ e)  
  → reduce _⊕_ e p ∘ flatten  
  ≡ reduce _⊕_ e p ∘ map (reduce _⊕_ e p)
```

```
flatten : ∀{A : Set} → List (List A) → List A  
flatten = foldr _++_ []
```

Hom-Hom Fusion

```

hom-hom : ∀{A B C : Set} (_⊕_ : C → C → C) (e : C)
  (p : IsMonoid _⊕_ e)(g : A → List B) (f : B → C)
  → hom _⊕_ e p f ∘ hom _++_ [] _g
    ≡ hom _⊕_ e p (hom _⊕_ e p f ∘ g)
  
```

$$\begin{aligned}
 & \oplus / \cdot f^* \cdot ++ / \cdot g^* \\
 = & \quad \{ \text{map promotion} \} \\
 & \oplus / \cdot ++ / \cdot f^* * \cdot g^* \\
 = & \quad \{ \text{reduce promotion} \} \\
 & \oplus / \cdot (\oplus /) * \cdot f^* * \cdot g^* \\
 = & \quad \{ \text{map distribution} \} \\
 & \oplus / \cdot (\oplus / \cdot f^* \cdot g)^*
 \end{aligned}$$

记号: $\text{reduce } \oplus _ e _ = \oplus /$
 $\text{map } f = f^*$

Accumulation Lemma

```
acc-lemma : ∀{A : Set} (_⊕_ : A → A → A) (e : A)  
→ scanl _⊕_ e ≡ map (foldl _⊕_ e) ∘ inits
```

$O(n)$

$O(n^2)$

$$(\oplus \# e) = (\oplus \# e) * \cdot inits$$

Honor's Rule

```
R-Dist : ∀{A : Set} (_⊕_ : A → A → A) (_⊗_ : A → A → A) → Set
R-Dist {A} _⊕_ _⊗_ = ∀ (a b c : A)
  → (a ⊕ b) ⊗ c ≡ (a ⊗ c) ⊕ (b ⊕ c)
```

```
horner-rule : ∀{A : Set} (_⊕_ : A → A → A) (e-⊕ : A)
  (_⊗_ : A → A → A) (e-⊗ : A)
  → (p : IsMonoid _⊕_ e-⊕)
  → (q : IsMonoid _⊗_ e-⊗)
  → (rdist : R-Dist _⊕_ _⊗_)
  -----
  → reduce _⊕_ e-⊕ p ∘ map (reduce _⊗_ e-⊗ q) ∘ tails
  ≡ foldl (λ a b → (a ⊗ b) ⊕ e-⊕) e-⊕
```

$$\oplus / \cdot \otimes / * \cdot \text{tails} = \odot \dashv e$$

where

$$e = id_{\otimes}$$

$$a \odot b = (a \otimes b) \oplus e$$

高效mss的程序推导

$$\begin{aligned} & mss \\ = & \quad \{ \text{definition of } mss \} \\ & \uparrow / \cdot + / * \cdot \text{segs} \\ = & \quad \{ \text{definition of segs} \} \\ & \uparrow / \cdot + / * \cdot ++ / \cdot \text{tails} * \cdot \text{inits} \\ = & \quad \{ \text{map and reduce promotion} \} \\ & \uparrow / \cdot (\uparrow / \cdot + / * \cdot \text{tails}) * \cdot \text{inits} \\ = & \quad \{ \text{Horner's rule with } a \odot b = (a + b) \uparrow 0 \} \\ & \uparrow / \cdot \odot \not\rightarrow_0 * \cdot \text{inits} \\ = & \quad \{ \text{accumulation lemma} \} \\ & \uparrow / \cdot \odot \not\rightarrow_0 \end{aligned}$$

作业

23-1. 证明 `_++_` 满足结合律，而且 `[]` 是它的单位元。

`++-assoc` : $\forall \{A : \text{Set}\} (xs\ ys\ zs : \text{List } A)$

$\rightarrow (xs\ ++\ ys)\ ++\ zs \equiv xs\ ++\ (ys\ ++\ zs)$

`++-identityl` : $\forall \{A : \text{Set}\} (xs : \text{List } A) \rightarrow []\ ++\ xs \equiv xs$

`++-identityr` : $\forall \{A : \text{Set}\} (xs : \text{List } A) \rightarrow xs\ ++\ [] \equiv xs$

作业

23-2. 证明以下性质:

$$\text{foldr} \text{++} : \forall \{A : \text{Set}\} (_ \otimes _ : A \rightarrow A \rightarrow A) (e : A) (xs\ ys : \text{List } A) \rightarrow \\ \text{foldr } _ \otimes _ e (xs \text{ ++ } ys) \equiv \text{foldr } _ \otimes _ (\text{foldr } _ \otimes _ e\ ys) xs$$

23-3. 证明 $\text{reverse} \circ \text{reverse} = \text{id}$, 其中 reverse 的定义如下。

$$\text{reverse} : \forall \{A : \text{Set}\} \rightarrow \text{List } A \rightarrow \text{List } A$$

$$\text{reverse } [] = []$$

$$\text{reverse } (x :: xs) = \text{reverse } xs \text{ ++ } (x :: [])$$

作业

23-4. 利用agda实现mss的程序推导。