

工学部専門科目

「プログラミング言語」 SICP
第3章 ~ Modularity, Objects, State ~
その6

五十嵐 淳

igarashi@kuis.kyoto-u.ac.jp

京都大学 大学院情報学研究科
通信情報システム専攻

May 29, 2013

今日のメニュー

3.5 ストリーム

3.5.3 ストリームの活用

3.5.4 ストリームと遅延評価

3.5.5 関数プログラムのモジュラリティとオブジェクト のモジュラリティ

3.5.3: ストリームの活用

- 繰り返し計算をストリーム化
- ペアの無限ストリーム
- シグナル(信号)としてのストリーム

繰り返し計算のストリーム化

1.1.7: Newton 法による平方根の繰り返し計算

```
(define (improve guess x) ;; よりよい近似値  
  (average guess (/ x guess)))
```

```
(define (sqrt-iter guess x)  
  (if (good-enough? guess x)  
      guess  
      (sqrt-iter (improve guess x)  
                  x)))
```

繰り返し計算のストリーム化

近似値のストリームとして表現

```
(define (sqrt-stream x)
  (define guesses
    (cons-stream
      1.0
      (stream-map (lambda (guess)
                    (improve guess x))
                  guesses)))
  guesses)
```

arctan のテイラー展開による円周率

$$\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \dots$$

数列 s_n から部分和 $\sum_{i=0}^n s_i$ を求める

```
(define (partial-sums s)
  (cons-stream
    (stream-car s)
    (stream-map
      + (stream-cdr s) (partial-sums s))))
```

```
(define (pi-summands n)
  (cons-stream
    (/ 1.0 n)
    (stream-map - (pi-summands (+ n 2)))))
```

```
(define pi-stream
  (scale-stream
    (partial-sums (pi-summands 1))
    4))
```

収束の加速: Aitken¹ の Δ^2 法

収束が遅い交代級数 S_n に対し,

$$S_{n+1} - \frac{(S_{n+1} - S_n)^2}{S_{n-1} - 2S_n + S_{n+1}}$$

の数列を考えると早く収束する

¹教科書は Euler となっているが...

```

(define (euler-transform s)
  (let ((s0 (stream-ref s 0))           ; Sn-1
        (s1 (stream-ref s 1))           ; Sn
        (s2 (stream-ref s 2)))          ; Sn+1
    (cons-stream
      (- s2 (/ (square (- s2 s1))
                (+ s0 (* -2 s1) s2)))
      (euler-transform (stream-cdr s))))

```

加速法を何度も適用する

- Δ^2 法を n 回適用した数列が n 番目に来ているような **ストリームのストリーム** (タプロー) を作る

```
(define (make-tableau transform s)
  (cons-stream s
    (make-tableau transform
      (transform s))))
```

- そして先頭要素だけ集める

```
(define (accelerated-sequence transform s)
  (stream-map stream-car
    (make-tableau transform s)))
```

ペアの無限ストリーム

和が素数になるようなペア (i, j) (ただし $i \leq j$) の列挙 (c.f. 2.2.3):

```
(stream-filter
  (lambda (pair)
    (prime? (+ (car pair) (cadr pair))))
  int-pairs)
```

さて int-pairs はどうやって書く？

- (i, j) (ただし $i \leq j$) の全ての組み合わせが「いつか」登場するような列
 - ▶ 「いつか」...その要素が現れる「番目」の自然数がある

一般化して考える

S_0, S_1, S_2, \dots と T_0, T_1, T_2, \dots に対して,

(S_0, T_0)	(S_0, T_1)	(S_0, T_2)	\dots
(S_1, T_0)	(S_1, T_1)	(S_1, T_2)	\dots
(S_2, T_0)	(S_2, T_1)	(S_2, T_2)	\dots
\vdots	\vdots	\vdots	\ddots

の対角線上・対角線より上を取りつくす (漏れなく番号をふる) 方法?

再帰的な構造

(S_0, T_0)	(S_0, T_1)	(S_0, T_2)	(S_0, T_3)	...
(S_1, T_0)	(S_1, T_1)	(S_1, T_2)	(S_1, T_3)	...
(S_2, T_0)	(S_2, T_1)	(S_2, T_2)	(S_2, T_3)	...
(S_3, T_0)	(S_3, T_1)	(S_3, T_2)	(S_3, T_3)	...
\vdots	\vdots	\vdots	\vdots	\ddots

- 左上 (S_0, T_0) はそれぞれのストリームの先頭
- 右上の領域は T_1, T_2, \dots から作れる
- 右下の領域は全体と「相似」 \implies 再帰!
- 右上と右下をどう組み合わせる？

```

(define (pairs s t)
  (cons-stream
    (list (stream-car s) (stream-car t)) ;; 左上
    (???? ;; ふたつのストリームを組み合わせる
      ;; 右上
      (stream-map
        (lambda (x) (list (stream-car s) x))
        (stream-cdr t))
      ;; 左下
      (pairs (stream-cdr s) (stream-cdr t))
    )))

```

ふたつのストリームを互い違いに組み合わせる

以下の `interleave` を前ページ ???? 部分で使えば OK!

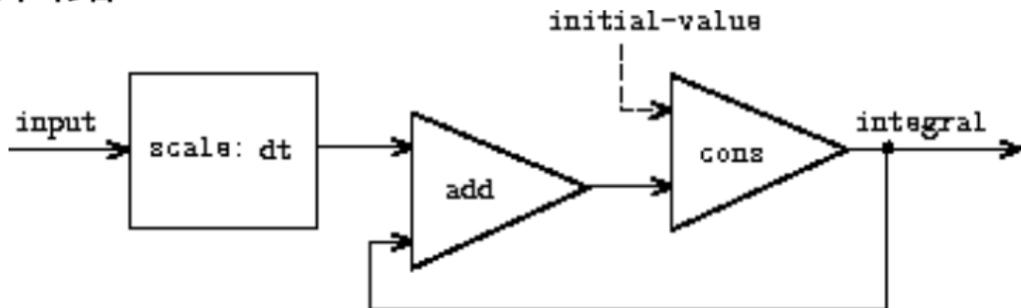
```
(define (interleave s1 s2)
  (if (stream-null? s1)
      s2
      (cons-stream
        (stream-car s1)
        (interleave s2 (stream-cdr s1)))))
```

- `stream-append` (p.340) じゃだめ! (どうして?)

シグナルとしてのストリーム

回路に流れる信号を各 (離散) 時刻での信号値の列で表現する

- 積分回路



刻み dt の信号 x_0, x_1, x_n, \dots に対する, 時刻 i までの信号値の和 (ただし C は定数)

$$S_i = C + \sum_{j=1}^i x_j dt$$

```
(define (integral integrand initial-value dt)
  (define int
    (cons-stream
      initial-value ;; 定数 C の値
      (add-streams (scale-stream integrand dt)
                    int)))
  int)
```

- 再帰と回路の「フィードバックループ」の対応
- 再帰的参照は `cons-stream` の第二引数の中にだけしか書かれていないおかげ

3.5.4 ストリームと遅延評価

微分方程式 $dy/dt = f(y)$ を数値的に積分して解く:

```
(define (solve f y0 dt)
  (define y ;; ほぼ integral と同じ
    (cons-stream
      y0
      (add-streams (scale-stream dy dt) y)))
  (define dy (map-stream f y))
  y)
```

- y 内の y , dy の (再帰的) 参照は `cons-stream` の第二引数の中
- dy の定義中の y の参照は定義が上から順なら OK(処理系に依るかも)

integral を使って書き直す

```
(define (solve f y0 dt)
  (define y (integral dy y0 dt))
  (define dy (stream-map f y))
  y)
```

- これは**ダメ!**
 - ▶ dy の再帰的参照が (特殊形式ではない) ふつうの関数呼出しの引数に現れている
 - ▶ dy の計算の遅延効果が失われた

手で delay を挿入

delay は引数式を評価せずに値を返す特殊形式:

```
(define (solve f y0 dt)
  (define y (integral (delay dy) y0 dt))
  (define dy (stream-map f y))
  y)
```

ただし, integral も要改造 ;-(

```
(define (integral integrand initial-value dt)
  (define int
    (cons-stream initial-value
      (... (force integrand) ...)))
  int)
```

正規順評価

部品化を追求

遅延すべき場所の発見・delay, force の挿入

ひとつの手続きについて, 遅延の有無でバリエーションを作る?

メンテが大変 (組合せの数の爆発)

いっそのこと全引数を自動的に遅延させてしまえば?

⇒ 1.1.5 の正規順 (normal order) 評価

式が評価されるタイミングがわかりづらく set!
などと相性が悪い (4.2 節)

3.5.5 関数プログラムのモジュラリティ とオブジェクトのモジュラリティ

状態を活用したモンテカルロ法プログラムをストリームで書き直す

状態を使った乱数

```
(define rand
  (let ((x random-init))
    (lambda ()
      (set! x (rand-update x))
      x)))
```

ストリームを使った乱数列

```
(define random-numbers
  (cons-stream
    random-init
    (stream-map rand-update random-numbers)))
```

Cesàro 法

乱数列 s_i に対し, s_i, s_{i+1} が互いに素かどうか (#t/#f) を並べた列を生成

```
(define cesaro-stream
  (map-successive-pairs
    (lambda (r1 r2) (= (gcd r1 r2) 1))
    random-numbers))
```

```
(define (map-successive-pairs f s)
  ;; s の要素ふたつずつに f を適用した結果の列
  (cons-stream
    (f (stream-car s) (stream-car (stream-cdr s)))
    (map-successive-pairs f (stream-cdr (stream-cdr s)))))
```

i 回目の試行までの「成功率」を集計した列

```
(define (monte-carlo experiment-stream passed failed)
  (define (next passed failed)
    (cons-stream
      (/ passed (+ passed failed))
      (monte-carlo
        (stream-cdr experiment-stream) passed failed)))
  (if (stream-car experiment-stream)
      (next (+ passed 1) failed)
      (next passed (+ failed 1))))

(define pi
  (stream-map (lambda (p) (sqrt (/ 6 p)))
              (monte-carlo cesaro-stream 0 0)))
```

状態変化は全く使っていない!

関数プログラミング的時間概念

- ストリーム = 各イベントにおける状態値の列
- 銀行口座をストリーム関数化:
 - ▶ 入力: 残高操作の履歴の列
 - ▶ 出力: 残高の推移の列(預金通帳みたいな感じか?)

```
(define (stream-withdraw balance amount-stream)
  (cons-stream
    balance
    (stream-withdraw
      (- balance (stream-car amount-stream))
      (stream-cdr amount-stream))))
```

- 状態変数によるモデリング
直観的

- ★ 我々の時間に沿ったシステムの時間のモデリング

- × 複雑

- ★ プログラム実行のモデル
- ★ エイリアシング
- ★ 状態変化がどういう順序で起こるかを考慮する必要あり

- ストリームによる状態モデリング
(Scheme レベルでは) 状態がない
遅延評価による複雑さ

ストリームには本当に問題がないのか？

- 共同口座をストリーム化
 - ▶ 入力 1: Peter 側の残高操作履歴
 - ▶ 入力 2: Paul 側の残高操作履歴
 - ▶ 出力: 残高の履歴
- 入力 1 と 2 をどうやって混ぜればいいのか!?
 - ▶ Peter と Paul の時間の「すりあわせ」問題

第3章のまとめ

状態変化を伴う部品を使ったシステム記述について

- Modularity(モジュール性)...システム全体を、より小さく、独立して開発/保守可能な部分システム(モジュール)に「自然に」分割可能であること
- Objects(オブジェクト)...(時間を経て変化する)内部状態を持つシステム部品
- State...状態

3章で扱うふたつの「世界観」

- システム as 時間の経過に応じて挙動を変える「オブジェクト」の集まり
- システム as 情報の流れ(ストリーム) (3.5 節)

それぞれの世界観が必要とする言語機構

- 「状態変化」を扱うためのプログラミング機構
 - ▶ 代入に基づく計算から環境に基づく計算へ
 - ▶ identity・エイリアスの問題
- 「ストリーム」を扱うための機構
 - ▶ 対象システムの時間・計算機上の事象順序の分離
 - ▶ 遅延評価