

「プログラミング言語」

SICP 第4章

～ 超言語的抽象～

その3

五十嵐 淳

igarashi@kuis.kyoto-u.ac.jp

京都大学

June 23, 2015

今日のメニュー

4.2 Variations on a Scheme – Lazy Evaluation

4.2.1 Normal Order and Applicative Order

4.2.2 An Interpreter with Lazy Evaluation

4.2.3 Streams as Lazy Lists

4.2 Scheme 変奏曲 — 遅延評価

- 評価器を改造して定義される言語を改造
- インタプリタを使った新しい機能・言語設計の実験
 - ▶ テスト・デバグが容易
 - ▶ 評価器を実装する言語の機能に「タダ乗り」できる!

4.2.1 Normal Order and Applicative Order¹

関数適用式に関する二つの評価方式 (実行順序)

- normal order: 引数の計算をせずに関数本体の計算を始める (引数が「本当に」必要になったら計算)
 - ▶ 遅延評価 (lazy evaluation), 名前呼び (call-by-name) とも
 - ▶ memoization を伴う遅延評価を特に必要呼び (call-by-need) ということもある
- applicative order: 関数本体の計算を始める前に引数の計算を済ませる
 - ▶ 値呼び (call-by-value) とも

¹ “Normal order” と “Applicative order” については、訳語はおろか、(専門用語として) その意味するところもあまり確定していないように思います。とはいえ、Wikipedia の訳語は日英ともにちょっと...

ふたつの方式の違い

```
(define (try a b)
  (if (= a 0) 1 b))
```

```
(try 0 (/ 1 0))
```

⇒ エラー (Scheme)

⇒ 1 (lazy Scheme)

遅延評価であれば...

こんな関数もプログラマが定義できる!

```
(define (unless cond usual exceptional)
  (if cond exceptional usual))
```

```
(unless (= b 0)
  (/ a b)
  (begin
    (display "exception: returning 0")
    0))
```

しかも, `unless` は特殊形式ではないので, 第一級の値としても使える!

(非) 正格性 — (non-)strictness

評価順序に基づいた定義:

関数 f が第 i 引数に関して (非) 正格

$\stackrel{\text{def}}{\Leftrightarrow}$ 第 i 引数の評価が関数本体の評価が始まる前 (始まった後) に行われる

- applicative order の下では全ての関数は各引数について正格
- normal order の下では全ての定義された関数は各引数について非正格/プリミティブはものによる

参考: 正格性のより標準的な定義

関数 f が第 i 引数に関して正格

^{def}
 $\Leftrightarrow e_i$ が値を持たない (計算が止まらない・エラーで異常終了する) ような式ならば関数適用 ($f \dots e_i \dots$) も値を持たない

- applicative order の下では全ての関数は各引数について正格
- normal order の下でも (前の定義と異なり) 例えば $(\text{lambda } (x) (+ x x))$ は第1引数について正格

非正格な cons

```
> (define (f x) (f x)) ;; 止まらない関数
```

```
ok
```

```
> (define a (cons (f 1) 4))
```

```
ok ;; 定義できる!
```

```
> (cdr a)
```

```
4
```

```
> (length a)
```

```
2
```

```
> (car a) ;; これはさすがに止まらない
```

⇒ ストリーム (3.5 節) を非正格 cons で実装できる!

4.2.2 遅延評価をするインタプリタ

基本仕様

- ユーザ定義関数は引数の評価を全て遅らせる
- プリミティブ関数は引数を全て評価してから呼ぶ
 - ▶ プリミティブの正体である Scheme の関数が正格なんだから仕方ない!

実装方針

- `thunk`(遅延された引数を表すデータ構造)の導入
- 適当なタイミングで `thunk` を計算 (`forcing`)

⇒ 主要な修正は `apply` 周辺

eval の改造

改造前

```
(define (eval exp env)
  (cond ...
    ((application? exp)
     (apply (eval (operator exp) env)
             (list-of-values (operands exp) env)))
    ...))
```

- 青字部分で applicative order を実現

eval の改造

改造後

```
(define (eval exp env)
  (cond ...
    ((application? exp)
     (apply (actual-value (operator exp) env)
             (operands exp) env))
    ...))
```

- apply には引数の式そのままと環境を渡す
- actual-value: 式の (thunk ではない) 値を得るための関数
 - ▶ 式を評価した結果: ふつうの値 or thunk

改造前

```
(define (apply proc args)
  ;; args は評価されたの値 (のリスト)
  (cond ((primitive-procedure? proc)
        (apply-primitive-procedure proc
                                       args))
        ((compound-procedure? proc)
         (eval-sequence
          (procedure-body proc)
          (extend-environment
           (procedure-parameters proc)
           args
           (procedure-environment proc))))
        ...))
```

改造後

```
(define (apply proc args env)
  ;; args は評価前の式 (のリスト)
  (cond ((primitive-procedure? proc)
        (apply-primitive-procedure proc
          (list-of-arg-values args env)))
        ((compound-procedure? proc)
        (eval-sequence
          (procedure-body proc)
          (extend-environment
            (procedure-parameters proc)
            (list-of-delayed-args args env)
            (procedure-environment proc))))
        ...))
```

引数処理関数

- actual-value: (thunk ではない) 値を得る
- delay-it: thunk を生成するための関数 (後述)

```
(define (list-of-arg-values exps env)
  (if (no-operands? exps) '()
      (cons (actual-value (first-operand exps) env)
            (list-of-arg-values
              (rest-operands exps) env))))

(define (list-of-delayed-args exps env)
  (if (no-operands? exps) '()
      (cons (delay-it (first-operand exps) env)
            (list-of-delayed-args
              (rest-operands exps) env))))
```

主要部分の残りの変更

(think ではない) 値が必要なところ

- プリミティブの引数 (前述)
- if の条件部
- 評価結果の表示
 - ▶ プロンプトに入力された式は計算する必要あり
- ...

で actual-value を (eval の代わりに) 使う

thunk の表現

```
(define (delay-it exp env)
  ;; 名前は make-thunk の方がよかったような？
  (list 'thunk exp env))
(define (thunk? obj)
  (tagged-list? obj 'thunk))
(define (thunk-exp thunk) (cadr thunk))
(define (thunk-env thunk) (caddr thunk))
```

actual-value, force-it の実装

相互再帰的

- actual-value: 評価して force
- force-it: thunk なら (中身の式の) 値を actual-value で計算

```
(define (actual-value exp env)
  (force-it (eval exp env)))
```

```
(define (force-it obj)
  (if (thunk? obj)
      (actual-value (thunk-exp obj)
                    (thunk-env obj))
      obj))
```

Memoization の追加

- 一回評価した thunk は結果を覚えておく
- 評価済みの thunk の表現を用意
- 副作用で thunk の中身を「未評価」から「評価済」へ変身させる

```
(define (force-it obj)
  (cond ((thunk? obj)
        (let ((result (actual-value
                        (thunk-exp obj)
                        (thunk-env obj))))
          (set-car! obj 'evaluated-thunk)
          (set-car! (cdr obj) result)
          (set-cdr! (cdr obj) '())
          result))
        ((evaluated-thunk? obj)
         ...)
        (else obj))))
```

```
(define (evaluated-thunk? obj)
  (tagged-list? obj 'evaluated-thunk))
(define (thunk-value evaluated-thunk)
  (cadr evaluated-thunk))

(define (force-it obj)
  (cond ((thunk? obj)
        ...)
        ((evaluated-thunk? obj)
         (thunk-value obj))
        (else obj)))
```

4.2.3 遅延リストとしてのストリーム

ストリームの3つの実装方法:

- ① `eval` を改造して `delay`, `cons-stream` を特殊形式として追加
 - ▶ (遅延評価であろうとなかろうと) やればできる・簡単
 - ▶ 特殊形式は高階関数に渡せない (not first-class)
 - ▶ ストリーム操作関数群を定義する必要あり
- ② 遅延評価を行うプリミティブとして `cons` などを再定義
 - ▶ `apply` は要改造: プリミティブの引数は全て評価することになっているので
- ③ 評価器は改造せずにユーザ定義関数として `cons` などを再定義

ユーザ定義関数としての cons

c.f Section 2.1.3 と Exercise 2.4

```
(define (cons x y)
  (lambda (m) (m x y)))
(define (car z)
  (z (lambda (p q) p)))
(define (cdr z)
  (z (lambda (p q) q)))
```

宿題： 6/30 16:30 締切

- Ex. 4.27, 4.33
- レポートには
 - ▶ 考え方の説明
 - ▶ プログラムリストと考え方の対応
 - ▶ 実行例を示すこと
- レポート (pdf) とプログラムファイルを提出
- 友達に教えてもらったなら、その人の名前を明記
- web は出典を明記 (「同じ」回答は減点)