システム理論演習 II

(5) Conventional Interface・記号処理

増原英彦 *(masuhara@graco.c.u-tokyo.ac.jp)*

1997 年 5 月 16 日

1 Conventional Interface

関数の抽象化は「定義の共通部分をとり出す」ことを行っていた。一見共通性のない関数でも、データを抽象化する —conventional interface を用意する — ことによって一見共通性のない関数定義も、抽象化できる場合が多い。

```
;;; ある木構造の葉の値のうち、奇数のものだけを 2 乗したものの和
(define (sum-odd-squares tree)
  (cond ((null? tree) 0)
        ((not (pair? tree))
         (if (odd? tree) (square tree) 0))
        (else
         (+ (sum-odd-squares (car tree))
            (sum-odd-squares (cdr tree))))))
;;; フィボナッチ数 Fib(k), k < n のうち、偶数のもののリスト
(define (even-fibs n)
  (define (next k)
    (if (> k n)
        '()
        (let ((f (fib k)))
          (if (even? f)
              (cons f (next (+ k 1)))
              (next (+ k 1))))))
  (next 0))
```

図 1: 一見共通性のない関数定義

図 1 の 2 つの関数には、一見共通性はないのだが、より抽象的な計算を考えると、「数の列」を発生させて、それを加工してゆくものだと見ることができる。つまり、

- `sum-odd-squares`
  - 木構造の葉の要素を並べる
  - ふるいにかけて奇数のものだけが残す
  - 各要素を 2 乗する
  - 関数 `+` で結果を集計する (初期値 `0`)
- `even-fibs`
  - 0 から n までの自然数を並べる
  - 各値に対するフィボナッチ数を求める
  - ふるいにかけて偶数のものだけを残す

– 関数 cons で結果を集計する (初期値は空リスト)

この見方は、ちょうど信号処理でやっていることに似ている:

- *enumerator*— 処理すべき「信号」を発生させる
- *filter*— 条件に合う「信号」だけを通過させる
- *map*— 「信号」を加工変形する
- *accumulator*— 流れて来た「信号」を集計する

そこで、「信号」即ち「数の列」をリストで表わすことにして、それに対する conventional interface を定義する。

- 上の *map* 段階は、すでに定義した map が使える。
- *filter*の定義。

```
(define (filter pedicate sequence)
  (cond ((null? sequence) '())
        ((predicate (car sequence))
         (cons (car sequence)
               (filter predicate (cdr sequence))))
        (else (filter predicate (cdr sequence)))))

> (filter odd? (list 1 2 3 4 5))
(1 3 5)
```

- *accumulator*の定義。

```
(define (accumulate op initial sequence)
  (if (null? sequence)
      initial
      (op (car sequence)
          (accumulate op inital (cdr sequnce)))))
```

- *enumerate*の部分は、個別に定義する必要がある。

  まずは「整数の列」を発生させる場合:

```
(define (enumerate-interval low high)
  (if (> low heigh)
      '()
      (cons low (enumerate-interval (+ low 1) high))))
```

  また、木構造の葉の値を並べる場合:

```
(define (enumerate-tree tree)
  (cond ((null? tree) '())
        ((not (pair? tree)) (list tree))
        (else
         (append (enumerate-tree (car tree))
                 (enumerate-tree (cdr tree))))))
```

この conventional interface を使って図 1 の関数を書き直すと、図 2 のようになる。「数の列」に対する加工手順を部品として組み合わせたものになっているのが見てとれるであろう。

数の列を扱うような計算はもちろん、一般的に「列」を加工してゆくような計算は、図 2 に類似した定義になる。

このように、リストとして表現された列は、 conventional interface として働く。これによって計算する “部品” を合成することができ、データ間の依存関係を簡潔に記述できる。

```
;;; ある木構造の葉の値のうち、奇数のものだけを 2 乗したものの和
(define (sum-odd-squares tree)
  (accumulate +
              0
              (map square
                   (filter odd?
                           (enumarate-tree tree)))))
;;; フィボナッチ数 Fib(k), k < n のうち、偶数のもののリスト
(define (even-fibs n)
  (accumulate cons
              '()
              (filter even?
                      (map fib
                           (enumerate-interval 0 n)))))
```

図 2: Conventional Interface を使って書き直した関数定義

課題 5-1: (Conventional interface の使用)

以下の関数を定義せよ。

- $0 \ldots n$ までのフィボナッチ数の 2 乗のリストを求める関数 (`fib-squares n`)
- ある列 (`seq`) の奇数要素について、各数を 2 乗したものの積を求める関数 (`product-squares seq`)

課題 5-2: (accumulate の使用)

リストを操作する関数の多くは `accumulate` によって定義できる。以下の穴を埋めよ。

```
(define (map p sequence)
  (accumulate (lambda (x y) ⟨??⟩) '() sequence))

(define (append seq1 seq2)
  (accumulate cons ⟨??⟩ ⟨??⟩))

(define (length sequence)
  (accumulate ⟨??⟩ 0 sequence))

(define (count-leaves tree)
  (accumulate ⟨??⟩ ⟨??⟩ (map ⟨??⟩ ⟨??⟩)))
```

課題 5-3: (fold)

関数 `accumulate` は、列の右側から値を集計してゆくため、別名 `fold-right` とも呼ばれている。

一方、列の左側から値を集計してゆく関数は `fold-left` と呼ばれる:

```
(define (fold-left op initial sequence)
  (define (iter result rest)
    (if (null? rest)
        result
        (iter (op result (car rest))
              (cdr rest))))
  (iter inital sequence))
```

以下の値を比べよ:

```
(fold-right / 1 (list 1 2 3))
(fold-left / 1 (list 1 2 3))
(fold-right list '() (list 1 2 3))
(fold-left list '() (list 1 2 3))
```

さて、 fold-right と fold-left に同じ引数を与えたとき、結果が同じになる場合と異なる場合がある。結果が同じになるのは、 op がどのような性質を満たす時か? ∎

課題 5-4: (fold その 2)

リストを逆順にする関数を fold-left, fold-right を使って定義するとどうなるか? 以下の穴を埋めよ:

```
(define (reverse sequence)
  (fold-left (lambda (x y) ⟨??⟩) '() sequence))
(define (reverse sequence)
  (fold-right (lambda (x y) ⟨??⟩) '() sequence))
```

∎

2 記号データ

いままでは、原始データとしては数だけを扱ってきた。ここでは新たに記号 (symbols) をを導入する。すでにプログラムを書くときには、変数名などとして記号を使っていたのだが、ここではそれをデータとして扱う。

プログラム中に普通に書かれた記号は、キーワードや変数と解釈される。記号をデータとして扱うためには、引用する (quote)。

'⟨式⟩

のように書くと、⟨式⟩ はプログラムではなくデータだとして扱われる:

```
> (define a 1)
> (define b 2)
> (list a b)
(1 2)
> (list 'a 'b)
(a b)
> (list 'a b)
(a 2)
```

引用は、記号だけでなく複合データでもできる:

```
> (car '(a b c))
a
> (cdr '(a b c))
(b c)
```

空リストを作るための記法 '() は、実は空リストを引用したものだった。

記号のために用意されている原始関数として、 2 つの引数が同じかどうかを調べる eq? がある。 (ここで言う「同じ」というのは言語によってそれぞれ違うので微妙なところである。 Scheme において「2 つの記号が同じである」というのは、同じ文字が同じ順序で並んでいることを言う。)

数式の微分

例として、数式を微分する問題を考える。例えば、 $ax^2 + bx + c$ という式を x で微分して $2ax + b$ を求めるようなことである。

そのためには、まず「数式」という抽象データを作り、それを使う世界で微分を定義するとよい。

数学における微分

まずは数学の世界での微分の定義を示しておく。

$$\frac{\mathrm{d}c}{\mathrm{d}x} = 0 \qquad c\text{ が定数か、 }x\text{ でない変数の場合} \tag{1}$$

$$\frac{\mathrm{d}x}{\mathrm{d}x} = 1 \tag{2}$$

$$\frac{\mathrm{d}(u+v)}{\mathrm{d}x} = \frac{\mathrm{d}u}{\mathrm{d}x} + \frac{\mathrm{d}v}{\mathrm{d}x} \tag{3}$$

$$\frac{\mathrm{d}(uv)}{\mathrm{d}x} = u\left(\frac{\mathrm{d}v}{\mathrm{d}x}\right) + v\left(\frac{\mathrm{d}u}{\mathrm{d}x}\right) \tag{4}$$

u や v というのは、複雑な式でもあるので、この定義は再帰的なものであることに注意。

数式抽象データを使った微分

これを Scheme 上で記述する。まず、以下のような関数が「数式」抽象データのために用意されているとする。

`(variable? e)`	`e` は変数か?
`(same-variable? v1 v2)`	`v1` と `v2` は同じ変数か?
`(sum? e)`	`e` は足し算式か?
`(added e)`	足し算式 `e` の第 1 項
`(augend e)`	足し算式 `e` の第 2 項
`(make-sum a1 a2)`	式 `a1`, `a2` を足した式
`(product? e)`	`e` は掛け算式か?
`(multiplier e)`	掛け算式 `e` の第 1 項
`(multiplicand e)`	掛け算式 `e` の第 2 項
`(make-product m1 m2)`	式 `m1`, `m2` を掛けた式

表 1: 数式抽象データの構築子・選択子

これを使って微分の定義は 3 のようになる。

```
(define (deriv exp var) ;; exp を var で微分する
  (cond ((number? exp) 0) ;; number? は引数が数かどうかを調べる原始関数
        ((variable? exp)
         (if (same-variable? exp var) 1 0))
        ((sum? exp)
         (make-sum (deriv (addend exp) var)
                   (deriv (augend exp) var)))
        ((product? exp)
         (let ((m1 (multiplier exp)) (m2 (multiplicand exp)))
           (make-sum
             (make-product m1 (deriv m2 var))
             (make-product (deriv m1 var) m2))))))
```

図 3: 微分関数の定義

数式の内部表現

次に、数式の内部表現を考える。ここでは $ax + b$ という式を表わすのに、 Scheme の式と同様に `(+ (* a x) b)` のような内部表現を用いることにする。

- 変数は記号で表わされるので、原始関数 `symbol?` を用いて

```
(define (variable? x) (symbol? x))
```

- 2 引数が同じ変数であるためには、 eq? であればよい

```
(define (same-variable? v1 v2)
  (and (variable? v1) (variable v2) (eq? v1 v2)))
```

- 足し算・掛け算は、リストとして表わす

```
(define (make-sum a1 a2) (list '+ a1 a2))
(define (make-product m1 m2) (list '* m1 m2))
```

- 式が足し算・掛け算であるかは、先頭の記号を eq? で調べればよい

```
(define (sum? x)
  (and (pair? x) (eq? (car x) '+)))
(define (product? x)
  (and (pair? x) (eq? (car x) '*)))
```

- 足し算・掛け算の第 1,2 項は、それぞれリストの第 2,3 要素になる

```
(define (addend s) (cadr s)) ;; (cadr s) = (car (cdr s))
(define (augend s) (caddr s)) ;; (caddr s) = (cadr (cdr s))
(define (multiplier p) (cadr p))
(define (multiplicand p) (caddr p))
```

課題 5-5: (deriv)

次の式を実行した結果はどうなるか?

```
(deriv '(+ x 3) 'x)
(deriv '(* x y) 'x)
(deriv '(* (* x y) (+ x 3)) 'x)
```

∎

課題 5-6: (簡約化)

このままだと、

```
> (deriv '(* x y) 'x)
(+ (* x 0) (* 1 y))
```

のようになってしまい、式を簡約化していない。そこで、足し算、掛け算について成り立つ次の規則 $e \cdot 0 = 0, \quad e \cdot 1 = e, \quad e + 0 = e$ や、数どうしの足し算や掛け算は実際に計算できることを利用すると、簡約化された形で式の内部表現を持つように make-sum を変更する。

```
(define (make-sum a1 a2)
  (cond ((=number? a1 0) a2)
        ((=number? a2 0) a1)
        ((and (number? a1) (number? a2)) (+ a1 a2))
        (else (list '+ a1 a2))))
;;; ただし =number? は、式がある数と等しいことを調べる関数
(define (=number? exp num)
  (and (number? exp) (= exp num)))
```

同様にして、簡約化する make-product を変更せよ。さらに、

```
(deriv '(+ x 3) 'x)
(deriv '(* x y) 'x)
(deriv '(* (* x y) (+ x 3)) 'x)
```

の結果はどう変わるか? ∎

課題 5-7: (冪乗)

数式の種類の中に冪乗を加える。微分に関する規則は

$$\frac{\mathrm{d}(u^n)}{\mathrm{d}x} = nu^{n-1}\left(\frac{\mathrm{d}u}{\mathrm{d}x}\right)$$

である。まず、冪乗に関する数式のための構築子・選択子として、 `exponentiation?`, `base`, `exponent`, `make-exponentiation` があるとして、関数 `deriv` の定義を拡張せよ。冪乗の内部表現としては、 e^n という式を `(** e n)` のように表わすことにする。上記の構築子・選択子を定義せよ。さらに、 $e^0 = 1$, $e^1 = e$ といった簡約化の規則を用いるとなおよろしい。

∎

課題 5-8: (多項演算子)

足し算と掛け算に関する数式が、任意個の項がとれるように拡張せよ。例えば、

```
(deriv '(* x y (+ x 3)) 'x)
```

のようなことができるようにするということである。ここでは、内部表現だけを変更して、 `deriv` の定義を変更せずに済むようにせよ。(ヒント: 3 項以上の足し算式に対しては、 `addend` は第 1 項を、 `augend` は第 2 項以降の足し算式を答えるようにするとよい。) ∎

課題 5-9: (中置記法)

この微分プログラムは、 Scheme の式とは無関係であることを示すために、数式の内部表現を「中置記法」(infix notation) つまり、演算子を項と項の間に書くものに変更せよ。

1. 簡単のため、足し算、掛け算の式は項を 2 つしかとらず、式には全部括弧がついていた場合のものを作れ。例えば、 `(x + (3 * (x + (y + 2))))` のようになっている。
2. 普通の数式のように、括弧を省略した場合はどうか。つまり、 `(x + 3 * (x + y +2))` のように書いた場合、掛け算が足し算に優先されるようにせよ。

∎