■ 連載

Haskell プログラミング

文字列間の距離—モナドを使って—

尾上 能之 (東京大学情報基盤センター)
onoue@ecc.u-tokyo.ac.jp

edit distance

今回も前回同様，動的計画法 (Dynamic Programming) で問題を解く例を紹介する．対象とするのは文字列間の距離を調べる編集距離 (edit distance) 問題である．

編集距離の定義はいくつか知られているが，ここでは与えられた 2 つの文字列に対し，一方の文字列を他方に変換するために必要な編集操作の回数の最小値として定義される Levenshtein 距離 [4] を求める．操作として通常，以下の 3 つが用いられる．

挿入 (insertion) 文字を挿入する
削除 (deletion) 文字を削除する
置換 (substitution) 別の文字に置き換える

例） pascal　距離 = 4
　　!　!! v　(! が置換，v が挿入）
　　haskell

この距離は，スペルチェッカのように 2 つの文字列の近さを判別する処理や，DNA 塩基配列間の距離を測るためにも用いられることが知られている．再帰を用いると，距離は以下の要素の中の最小値として求めることができる．

1. 先頭文字が同じ場合 : 残りの部分の距離
2. 先頭文字が異なる場合 :
   (a) 先頭文字に置換を適用したものとし，残りの部分の距離プラス 1
   (b) 先頭文字を挿入したものとし，残りの部分の距離プラス 1
   (c) 先頭文字を削除したものとし，残りの部分の距離プラス 1

これをプログラムにすると以下のようになる．引数を 2 つ組にして渡しているのは，後に前号で扱ったメモ化の手法を適用し効率を改善するので，そのための準備である．

```
ed :: (String, String) -> Int
ed ([],       []) = 0
ed (xs@(_:_), []) = length xs          -- xs@(_:_)  xs が [] 以外の場合を示すパターン
ed ([], ys@(_:_)) = length ys          --           上とオーバーラップを避ける
ed (xxs@(x:xs), yys@(y:ys)) =
  let a = ed (xs,  ys)
      b = ed (xxs, ys)
      c = ed (xs, yys)
  in minimum [ (if x==y then 0 else 1) + a, 1 + b, 1 + c ]
```

しかしこの定義では，残りの部分に対して計算した距離を保存していないため同じ計算を繰り返すことになり，効率は指数オーダと大変悪い．そこで前号でも扱ったメモ化を用いるが，今回はメモ化の表を維持管理するための枠組みとしてモナドを用いることにする．

モナドについて

モナド (Monad) とは，Haskell の既存の型を拡張するために用意された統一的な枠組みのことで，これを用いると純粋な関数型言語で扱いにくい入出力や状態の保存・更新などの処理を陽に意識せず簡単にできるメリットがある．

まずはじめに簡単な例として，Maybe モナドの使い方を見てみよう．Maybe モナドとは結果がない可能性のある計算を表すもので，たとえばデータベース検索の結果を表すときなどに用いられる．そのための型定義は以下のようになる．

```
data Maybe a = Nothing | Just a

-- キーを用いた連想リストの検索
lookup            :: Eq a => a -> [(a,b)] -> Maybe b
lookup k []        = Nothing            -- 見つからなかった場合
lookup k ((x,y):xys)
      | k==x       = Just y             -- 見つかった場合
      | otherwise = lookup k xys
```

関数 lookup は k をキーとして連想リストを検索し，見つかった場合は x(==k) に対応する y をラップ (wrap) した Just y を，見つからなかった場合は Nothing を返す．

プログラムを書く際，検索結果が見つかったときとそれ以外の場合に応じて，行う処理を分けることが多い．これには通常 case 文などの条件分岐を用いることになり，処理が複雑になればこのような分岐が繰り返し行われ，複数の枝を持つ樹状に分岐が細分化されることになる．

そこで，検索結果が見つからなかった場合の処理を共通化してよい場合，Maybe モナドを用いると全体の処理をすっきり記述することができる．

```
databaseAB :: [(A,B)]
databaseBC :: [(B,C)]
someProcess :: C -> D

{-
-- case 文を用いた自然な定義
chainLookup a =
  case (lookup a databaseAB) of
    Nothing -> Nothing
    Just b  -> case (lookup b databaseBC) of
                 Nothing -> Nothing
                 Just c  -> Just (someProcess c)
-}

-- (>>=), return を用いた同等な定義
chainLookup :: A -> Maybe D
chainLookup a =
  lookup a databaseAB >>= \ b ->              -- 1:
  lookup b databaseBC >>= \ c ->              -- 2:
  return (someProcess c)                      -- 3:
```

ここで出てきた (>>=)(バインドと呼ぶ), return はモナドを扱うときの基本的な演算で，return はモナドを生み出し，(>>=) はモナドとモナドを橋渡しする働きを持つ．なお (>>=) は中置演算子なので単独で表すときは括弧で囲み，中置する場合は括弧をとって用いる．chainLookup の右辺は以下のような構成になっており，

```
(>>=) (lookup a databaseAB)
      (\ b ->
          (>>=) (lookup b databaseBC)
                (\ c ->
                    return (someProcess c)))
```

各行における処理を言葉に読み換えると次のようになる．検索結果が見つからなかった場合は，各 lookup で返される Nothing が，そのまま chainLookup の返り値となる．

1. a をキーにして databaseAB を検索し，見つかった結果を変数 b に束縛．
2. b をキーにして databaseBC を検索し，見つかった結果を変数 c に束縛．
3. c に対し処理 someProcess を行い，結果は return で返す．

(>>=), return は計算をモナドの枠組みで扱うとき共通に現れる処理で，Maybe モナドでは以下のように定義される．なお Haskell においてモナドはクラスとして抽象化されており，Maybe 型のようにモナドの性質を備える型は Monad クラスのインスタンスとして定義される．

```
instance Monad Maybe where
    Just x  >>= k  = k x           -- 1: (>>=) の定義，第一引数が Just の場合
    Nothing >>= k  = Nothing       -- 2: (>>=) の定義，第一引数が Nothing の場合
    return         = Just          -- 3: return の定義
    fail s         = Nothing       -- 4: Monad クラスで用意された定義を上書き
```

モナドの処理を抽象化した Monad クラスの定義は以下のようになる．クラスには，各インスタンスが備えるべきクラスメソッドが定められており，Monad クラスでは return, (>>=), (>>), fail の 4 つのメソッドとなっている．ただし (>>), fail は以下のように標準の定義が用意されているため，Monad クラスのインスタンスを定義するには最低限 (>>=), return を用意すればよいことになる．

```
infixl 1 >>, >>=         -- 結合順位とともに中置演算子であることを指定

class Monad m where
    return :: a -> m a
    (>>=)  :: m a -> (a -> m b) -> m b
    (>>)   :: m a -> m b -> m b
    fail   :: String -> m a

    -- Minimal complete definition: (>>=), return
    p >> q  = p >>= \ _ -> q          -- (>>=) で p の結果が不要な場合
    fail s  = error s                 -- パターンマッチが失敗した場合
```

さきほど定義した chainLookup は，モナドに関する糖衣構文 (syntax sugar) である do 記法を用いると，以下のような手続き型言語の記述に似通った等価な定義に書き換えることができる．

```
-- do 記法
chainLookup :: A -> Maybe D
chainLookup a = do              -- 新たに do を追加
  b <- lookup a databaseAB
  c <- lookup b databaseBC
  return (someProcess c)
```

do 記法から do を用いない等価な定義に書き換えるための変換規則は以下のようになる．

```
a) do { e }                      =  e
b) do { e ; stmts }              =  e >> do { stmts }
c) do { p <- e ; stmts }         =  let ok p = do { stmts }
                                        ok _ = fail "..."
                                    in e >>= ok
```

```
d) do { let decls ; stmts }          =  let decls in do { stmts }

-- 規則 c) において，p が変数でパターンマッチに失敗しない場合
c') do { v <- e ; stmts }            =  e >>= (\ v -> do { stmts })
```

モナドの概念は，圏論と呼ばれる数学の分野をもとに構成されているが，その細かい理論について知らなくてもプログラムを書けるようライブラリとして整備されている．Haskell での有用な処理のいくつかにはモナドを用いることで共通化できる部分が多くあることが示され[3]，その後 Monad クラスに基づいたライブラリが作られることになった．その例として，6 月号で紹介した I/O モナドや次節で見る State モナドなどが挙げられる．最近の Haskell 処理系では階層ライブラリ[2]がサポートされていて，モナドに関するさまざまなライブラリは `import Control.Monad.Maybe` のように階層をピリオドで区切った名前でインポート宣言すれば利用可能となる．

モナドを使う利点として，以下が挙げられている[7]．

モジュラ性　大きな計算をより簡単な計算の組合せとして構築できる．実際に行われる計算と組合せの戦略を分離できる．

柔軟性　モナドを用いず書かれた同等のプログラムより，ずっと汎用的なものとしてプログラムを記述できる．これは，モナドは計算のための戦略をプログラム全体にばらまく代わりに，1 カ所に集中してまとめあげるためである．

分離性　関数プログラムのメイン部から，命令的な計算構造を安全に分離することを可能にする．これは，通常では参照透明性を満たさない I/O などの副作用を持つ処理や状態を持つ処理を，Haskell のような純粋な関数型言語上で扱うのに適している．

モナドにはこのようなメリットがある反面，構造を抽象化している点，処理が表と裏に分かれるので直感的に理解しづらい点などから，Haskell を習得する上での壁となることも多い．しかしモナドを活用することで Haskell で扱える処理は飛躍的に増えるため，Web 上の解説記事[7]も参照しつつ，馴染んでいってほしい．

State モナド

前号では，関数における処理間においてメモ化のための表を受け渡しするために，状態を表す以下のような State 型を定義し活用した．

```
type State s t = s -> (t,s)
```

実はこれとほぼ同等の処理を行うための State モナドが，Haskell のライブラリには用意されている．

```
-- Control.Monad.State より抜粋

newtype State s a = State { runState :: s -> (a, s) }

instance Monad (State s) where
    return a = State $ \s -> (a, s)           -- f $ ... = f ( ... )
    m >>= k  = State $ \s -> let               -- 1:
                 (a, s') = runState m s        -- 2:
                 in runState (k a) s'          -- 3:

evalState :: State s a -> s -> a
evalState m s = fst (runState m s)    -- モナド m に初期状態 s を渡し評価,
                                      -- fst で結果と状態の 2 つ組から結果を返す
```

`newtype` 宣言は，`type` 宣言が既存の型の別名を付けるだけであったのに対し，既存の型から新たな型を生み出すために用いられる．定義右辺の `runState` はフィールドラベルと呼び，関数適用の形で構成子を外し中身を抽出するセレクタ関数としての働きを持つ．

```
runState (State a) = a                -- runState は State の逆関数に相当
```

State モナドでも (>>=), return の処理がモナド上の基本的な演算となる．return は，結果 a を State モナドの枠組みで扱えるよう，状態を受け渡す処理を関数で表し State でラップする．m >>= k の処理は少し複雑なので，順を追って見てみることにしよう．

0.（(>>=) を用いた段階では後に実行されるラムダ式を用意するだけで，実際の処理は行わないことに注意）
1. 外側で evalState が呼ばれると，s にこれまでの状態が束縛されて呼び出される
2. runState でモナド m のラムダ式を取り出し状態 s を渡して計算．結果と新しい状態の 2 つ組（a, s'）を得る
3. 関数 k を結果 a に適用し，次のモナドを生成．それに新しい状態 s' を渡し，計算を続ける．

これにより，State モナドを (>>=) で繋ぎ合わせることによって状態を保存することができるようになる．状態の確認や更新のためには，以下の get や put を用いればよい．

```
instance MonadState s (State s) where
    get   = State $ \s -> (s, s)      -- これまでの状態 s を結果として設定
    put s = State $ \_ -> ((), s)     -- 状態を指定した s に，
                                      -- 結果（通常使わない）は () に設定
```

メモ化した改良版

今回はメモ化に用いる表のデータ構造として，Data.FiniteMap ライブラリに用意されている FiniteMap データ型を用いることにする．なおこのライブラリは，その後継でより効率のよい Data.Map ライブラリに置き換えることが推奨されているので，可能な場合はこちらを利用するとよいであろう[☆1]．

```
import Control.Monad.State      ( State, get, put, evalState )
import Data.FiniteMap           ( FiniteMap, lookupFM, addToFM, emptyFM )
{- 最近の処理系なら以下も可．ただし lookup,insert は引数の順序が変わる
import Prelude                  hiding ( lookup )     -- 名前の衝突を避けるため隠す
import Data.Map                 ( Map, lookup, insert, empty )
-}

type Table k v = FiniteMap k v
type Memo a b = State (Table a b) b

memoise :: Ord a => (a -> Memo a b) -> a -> Memo a b
memoise f x = do                              -- 1:
  table <- get                                -- 2: 表の取得
  case (lookupFM table x) of                  -- 3: 表の検索
    Just y  -> return y                       -- 4: すでに計算済みだった場合
    Nothing -> do fx <- f x                   -- 5: 未済のときは計算
                  table' <- get               -- 6: 表の再取得
                  put (addToFM table' x fx)   -- 7: 表に計算結果を格納
                  return fx                   -- 8: 結果を返す

runM :: (a -> Memo a b) -> a -> b
runM m v = evalState (m v) emptyFM
```

memoise の 6 行目で表を再取得しているのは，5 行目の計算中に再帰の奥の方で表が更新される可能性があるので，最新版を入手する必要があるためである．

メモ化された関数は Monad クラスの型を返すようになっているので，そこから結果を取り出すために関数 runM を用意する．これは対象となる計算を表すモナド m v に対し，表の初期値として emptyFM を渡すことで，内部の計

☆1 このプログラムは http://www.sampou.org/haskell/ipsj/ から取ることができる．

算を開始するものである．ここまででメモ化のための道具が揃ったことになる．

次は関数をメモ化するため，対象となる関数がモナドを扱えるよう

- 値を返すところは return
- 逐次的に処理を続けるところは（>>=）や do 記法

を用いた表現に書き換える．以下にモナドを返す関数 edM を示すので，最初に示した ed と見比べてみて欲しい．

```
edM :: (String, String) -> Memo (String, String) Int
edM ([],       []) = return 0
edM (xs@(_:_), []) = return (length xs)
edM ([], ys@(_:_)) = return (length ys)
edM xys@(_:_, _:_) = memoise edM' xys    -- memoise の追加
  where
    edM' (xxs@(x:xs), yys@(y:ys)) = do
      a <- edM (xs,  ys)             -- edM' ではなくメモ化済みの edM を利用
      b <- edM (xxs, ys)
      c <- edM (xs, yys)
      return (minimum [ (if x==y then 0 else 1) + a, 1 + b, 1 + c ])

ed_memo :: (String, String) -> Int
ed_memo = runM edM
```

以下に実行した結果を示す．今回 Haskell 98 の規格に含まれないライブラリ (依存パラメータを使用) を間接的に用いているため，hugs の起動時に -98 オプションを付ける必要がある．また +s オプションは統計情報を出力するオプションで，ghci 同様起動後に :set +s してもよい．

```
$ hugs -98 +s EditDistance.hs
    (...)
Main> ed ("pascal", "haskell")
4
(631380 reductions, 893871 cells, 1 garbage collection)
Main> ed_memo ("pascal", "haskell")
4
(37787 reductions, 98524 cells)
```

変換操作の表示

これまでのプログラムでは編集距離の値しか求めていなかったため，変換後の文字列を得るために行われた操作の内容については知ることができなかった．そこで少し手を加えて，距離と合わせてその距離を得るための操作 (挿入，削除，置換) の過程を求め表示させることにしよう．まず最初に，前章のプログラムに追加する形でライブラリのインポート宣言とデータ型ならびにインスタンス宣言の追加を行う．

```
-- transpose :: [[a]] -> [[a]]  リストのリストを行列とみなし転置
-- ( 例 . transpose [[1,2,3],[4,5,6]] == [[1,4],[2,5],[3,6]])
import Data.List        ( transpose )

data OpC = Ins Char | Del Char | Subst Char Char | Keep Char
type IntOps = (Int, [OpC])

-- 各操作を，変換前の文字，操作を表す文字，変換後の文字，と長さ 3 の文字列で表現
instance Show OpC where
  show (Ins c)     = ['-', 'v',  c ]
  show (Del c)     = [ c,  '^', '-']
  show (Subst c d) = [ c,  '!',  d ]
  show (Keep c)    = [ c,  ' ',  c ]
```

これまで距離を表す Int (のモナド) を返していた関数 edM を，編集距離と操作リストの 2 つ組を返すように変更する．また最小値をとる処理では，3 つの 2 つ組から距離が最小となるものを返すための関数 min3 を用意しこれを用いている．

```
edM :: (String, String) -> Memo (String, String) IntOps
edM ([],       []) = return (0, [])
edM (xs@(_:_), []) = return (length xs, map Del xs)
edM ([], ys@(_:_)) = return (length ys, map Ins ys)
edM xys@(_:_, _:_) = memoise edM' xys
  where
    edM' (xxs@(x:xs), yys@(y:ys)) = do
      (a, ops_a) <- edM (xs,  ys)
      (b, ops_b) <- edM (xxs, ys)
      (c, ops_c) <- edM (xs, yys)
      return (min3 (if x==y then (a, Keep x:ops_a)
                            else (1+a, Subst x y:ops_a))
                   (1+b, Ins y:ops_b)
                   (1+c, Del x:ops_c))

--  2 つ組の第 1 要素 (fst) だけで順序を決定
min3 :: Ord a => (a,b) -> (a,b) -> (a,b) -> (a,b)
min3 a b c = if a `le` b then a `min2` c else b `min2` c
  where
    (x, _) `le` (y, _) = x <= y
    min2 x y | x `le` y  = x
             | otherwise = y
```

関数 ed_all によって，求めた編集距離とその距離を満たす操作の過程を表示する．ここで transpose を用いることで，そのままではリストで表された操作の過程が上から下へ順に表示されるところを，左から右へと文字列を表示するのに自然な方向へ表示するよう置き換えている．

```
ed_memo :: (String, String) -> IntOps
ed_memo = runM edM

ed_all :: (String, String) -> IO ()
ed_all xys = do
  putStrLn $ show d
  mapM_ putStrLn $ transpose $ map show ops
  where
    (d, ops) = ed_memo xys
```

実行結果は以下のようになる．変換前と後の文字列の間には，上下の文字を対応づける形で操作を表す文字が表示され，! は置換，v は挿入，^ は削除の操作を意味する．

```
Main> ed_all ("pascal", "haskell")
4
pascal-
!  !! v
haskell

(37765 reductions, 99594 cells)
Main> ed_all ("mathematical games", "metamagical themas")
8
mathematical ga-mes
 ! !^  !     !!v !
meta-magical themas

(782392 reductions, 2419388 cells, 2 garbage collections)
```

この編集距離を求める問題は，文字列の長さを n とすると最初の素朴な定義では指数オーダだった処理が，一般の動的計画法を用いたアルゴリズムでは $O(n^2)$ に改善されることが知られている．これは $n \times n$ の行列を作り各文字列を行列の最左列と最上行に配置し，行列の左上すなわち各文字列の先頭から比較を行うことによって，距離で行列を埋めていく作業に相当する．今回 Haskell でメモ化を用いたプログラムでは，手続き型では通常 $O(1)$ となる行列要素の値にアクセスする作業が，$O(\log n)$ かかるメモ化した表の `lookupFM` に置き換えられるので，オーダとしては $O(n^2 \log n)$ となる．

さらに改善する方法として，$n \times n$ の行列全体を埋めるのではなく対角線要素の近傍だけで距離を求める O(ND) アルゴリズム [6] や，O(NP) アルゴリズム [8] がある．この手法を用いるとオーダ $O(n*(1+d))$ (ただし d は編集距離) で求めることが可能となる．また関数型の遅延評価を用いて，シンプルなアルゴリズムから O(ND) アルゴリズムを導出した例 [1] も知られている．

今回は紹介しきれなかったが，Haskell でもモナドを用いることでさまざまな処理が行えるようになる．その一例として，グラディウスというシューティングゲームを Haskell で書き直した Monadius[5] を挙げておこう．ゲームのように状態を扱うリアルタイム処理は通常関数型言語で扱うのに不向きとされるが，その間を取り持ち関数プログラミングの世界を広げるための枠組みがモナドなのである．

参考文献

1) Allison, L.: Lazy Dynamic-programming Can be Eager, Inf. Process. Lett., Vol.43, No.4, pp.207-212 (1992).
2) Haskell Hierarchical Libraries: http://www.haskell.org/ghc/docs/latest/html/libraries/
3) Jones, M. P.: Functional Programming with Overloading and Higher-Order Polymorphism, Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text, London, UK, Springer-Verlag (1995).
4) Levenshtein, V. I.: Binary Codes Capable of Correcting Deletions, Insertions, and Reversals, Soviet Physics - Doklady, Vol.10, No.8, pp.707-710 (Feb. 1966).
5) Monadius: http://www.geocities.jp/takascience/windows/monadius.html
6) Myers, E. W.: An O(ND) Difference Algorithm and Its Variations, Algorithmica, Vol.1, No.2, pp.251-266 (1986).
7) Newbern, J.: All About Monads, —A Comprehensive Guide to the Theory and Practice of Monadic Programming in Haskell: http://www.nomaware.com/monads/html, (訳, モナドのすべて, http://www.sampou.org/haskell/a-a-monads/html/).
8) Wu, S., Manber, U., Myers, G. and Miller, W.: An O(NP) Sequence Comparison Algorithm., Inf. Process. Lett., Vol.35, No.6, pp.317-323 (1990).

（平成 17 年 7 月 15 日受付）