■ 連載

Haskell プログラミング

記憶（memo）する関数

山下 伸夫（（株）タイムインターメディア）
nobsun@sampou.org

例題：両替問題（Count Change Problem）

簡単な問題から考える．

問題　50 セント，25 セント，10 セント，5 セント，1 セントがあるとして 1 ドルの両替には何通りあるか [1)]．

硬貨の種類（額面）のリストが与えられたとして，指定した金額の両替は何通りあるかを計算する関数を定義すると以下のとおり．

```
type Amount = Integer
type Coin   = Integer
type Count  = Integer

-- 金額と貨幣（額面）のリストから，両替の場合の数へ
cc :: Amount -> [Coin] -> Count
cc 0 _  = 1                    -- 金額がちょうど 0 なら，両替は 1 通り
cc _ [] = 0                    -- 両替に使う貨幣がなければ，両替は 0 通り
cc a ccs@(c:cs)
  | a < 0     = 0              -- 金額が 0 より少なければ，両替は 0 通り
  | otherwise = cc (a-c) ccs -- 最初の種類の貨幣額面を引いた金額の両替の場合の数
              + cc a     cs  -- 最初の種類の貨幣以外を使う両替の場合の数
```

ccs@(c:cs) はアズパターンという．こう書くことで，パターン c:cs に ccs という名前をつけることができる．

上の定義をファイル cc.hs に保存して[☆1] 実行してみる．

```
% ghci -v0 cc.hs
*Main> :set +s
*Main> cc 100 [50,25,10,5,1]
292
(0.05 secs, 1264564 bytes)
```

-v0 は対話モードのインタープリタ ghci のバナー出力を抑制するオプションである．:set +s は指定された式を評価後，その計算にかかった時間（およびメモリマネージャが確保したメモリ量）を表示させるためのコマンドである．

最初の問題の解は 292 通りである．**図 -1** に，cc を使って 11 セントの両替の場合の数を求める計算が展開する様

[☆1] この記事で書いたプログラムは http://www.sampou.org/haskell/ipsj/ から取ることができる．

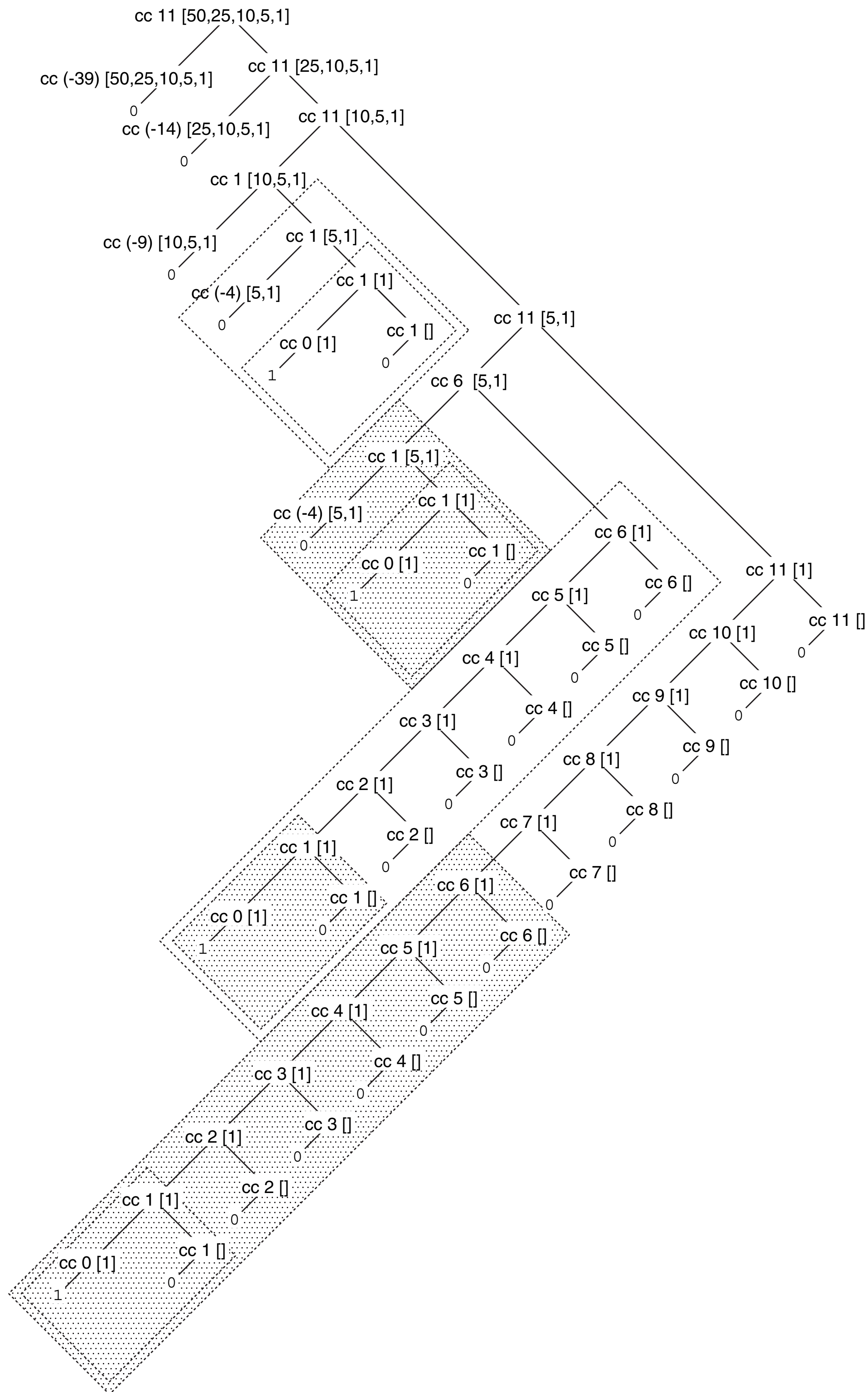

図-1　11セントの両替（影部分は冗長な計算）

子を示す．このプログラムでは，cc 11 [10,5,1] を計算するには，cc 1 [10,5,1] と cc 11 [5,1] を計算し，cc 11 [5,1] を計算するには cc 6 [5,1] と cc 11 [1] を計算する．このように計算が枝分かれするような再帰構造のことを樹状再帰 (tree recursion) という．上の両替問題を解くプログラムの定義は解を直截に表現しているが，計算の効率は悪い．たとえば，日本の通貨で千円札の両替の場合の数を求めると

```
*Main> cc 1000 [500,100,50,10,5,1]
248908
(312.03 secs, 1071624992 bytes)
```

5 分以上かかってしまった．1 万円札の両替はとても計算してみる気にはならない．図 -1 を見ても分かるように，このプログラムの定義では冗長な計算をしていてあまりにも効率が悪い．実際，両替金額に対して指数オーダの計算になる．

メモ化 (memoisation)

そのまま実行すると冗長な計算を繰り返す効率の悪いプログラムを改善する工夫の 1 つがメモ化である．アイディアの基本は計算結果を引数をキーにして表 (table) に蓄えるというもので，関数をある引数に対して適用するとき，まずその引数をキーに表を引くことで冗長な計算を回避する．メモ化はもとの理解しやすい樹状再帰構造を保存したままプログラムを改良できる点が優れている．

メモ版 memocc

cc のメモ化を試みる．Haskell プログラミングの定跡どおり，まず型で考える．memocc は計算結果を蓄える表を使うので，その表も引数で渡す．また，表は計算ごとにエントリが追加される可能性があり，エントリが追加されて賢くなった表を次の計算で使いたいので，memocc は計算結果と表のペアを返す．表の型を Table とすると，memocc の型は，

```
memocc :: Amount -> [Coin] -> Table -> (Count, Table)
```

となる．定義本体は，

```
memocc 0 _  tbl = (1,tbl)                                      --  1:
memocc _ [] tbl = (0,tbl)                                      --  2:
memocc a ccs@(c:cs) tbl                                        --  3:
  | a < 0     = (0,tbl)                                        --  4:
  | otherwise = case lookupTable (a,ccs) tbl of                --  5: 表を引く
                  (v:_) -> (v,tbl)                             --  6: 表に登録済の場合
                  []    -> let                                 --  7: 表に未登録の場合
                              (cnt1,tbl1) = memocc (a-c) cs  tbl   --  8: 左の枝
                              (cnt2,tbl2) = memocc a     ccs tbl1  --  9: 右の枝
                              cnt3 = cnt1 + cnt2                   -- 10: 左右の結果の合成
                              tbl3 = insertTable (a,ccs) cnt3 tbl2 -- 11: 表に新たなエントリ追加
                           in (cnt3,tbl3)                      -- 12: 返り値
```

cc の樹状再帰構造が保たれているので，memocc の定義を読むのは難しくない．定義本体の 1 行目から 4 行目までは cc の定義とほぼ同じなので説明は不要だろう．5 行目以降がメモ化された部分である．5 行目は引数のペアをキーにして表を索いている．lookupTable は指定したキーで表を索き，表の中で同じキーを持つ値のリストを返し，同じキーを持つ値がなければ空リストを返す関数である．6 行目は表に登録済の場合で，この時はその登録されている値と表をペアにしたもの返す．7 行目以降は表には未登録の場合である．与えられた表を使って計算した樹状再帰の左の枝の結果と更新された表でそれぞれ cnt1，tbl1 を束縛する（8 行目）．左の枝の計算により更新された表 tbl1 を使って計算した樹状再帰の右の枝の結果と更新された表でそれぞれ cnt2，tbl2 を束縛する（9 行目）．左右の枝の結果を合成し（10 行目），その結果 cnt3 を最初の引数のペア (a,ccs) をキーとして表 tbl2 に登録する（11 行目）．最後に結果 cnt3 と表 tbl3 をペアにして返す（12 行目）．

表とそれに対する操作

次に表 (Table) を具体的に定義する．表を表現する方法はいろいろあるが，ここでは降順にならんだ連想リスト (association list) を使う．連想リストはキーと対応する値のペアのリストで表現する．この場合キーとなるのは memocc の元々の 2 つの引数（金額と貨幣（額面）のリスト）である．2 つの引数に対応して 2 階層になった表にすることもできるが，単純な構造のほうが分かりやすいので，ペアにまとめる．キーが (Amount,[Coin])，対応する値が Count とする．Table の定義は，

```
type Table = [(Key,Value)]
type Key   = (Amount, [Coin])
type Value = Count
emptyTable :: Table
emptyTable = []
```

と書ける．表に対する操作は

```
lookupTable :: Key -> Table -> [Value]         -- 表を引く
lookupTable key []           = []
lookupTable key ((k,v):tbl) | key >  k  = []
                            | key == k  = [v]
                            | key <  k  = lookupTable key tbl

insertTable :: Key -> Value -> Table -> Table  -- 表にエントリを追加する
insertTable k v tbl = case break ((k >) . fst) tbl of
                        (xs,ys) -> xs ++ (k,v):ys
```

とする．

実行例

memocc の定義は表の受け渡しが表に（引数と返り値に）現れているが，表は計算の補助具にすぎない．余分なものを隠すために，evalMemoCC というドライバを定義する．

```
evalMemoCC :: Amount -> [Coin] -> Count
evalMemoCC amount coins = fst (memocc amount coins emptyTable)
```

コードをファイル memocc.hs に保存して，ghci にロードして実行する．

```
% ghci -v0 memocc.hs
*Memo> :set +s
*Memo> evalMemoCC 1000 [500,100,50,10,5,1]
248908
(0.03 secs, 1009500 bytes)
*Memo> evalMemoCC 10000 [5000,2000,1000,500,100,50,10,5,1]
24597373438
(0.32 secs, 6256204 bytes)
```

先程の素朴な実装 cc では試す気にならない 1 万円札の両替もすぐに結果が出る．

再利用のための工夫

メモ化を使って，「両替問題」の素朴な樹状再帰構造の解を効率のよいプログラムに書き換えることに成功した．しかし，このプログラムは「両替問題」を解くことにしか利用できない．このプログラムをもう少し抽象化してメモ化部分を再利用できるよう高階関数として取り出す．

表のパラメータ化

いろいろな問題にメモ化手法を使うのなら，利用する表のエントリ（キーと対応する値のペア）も固定のものではなく，柔軟に対応できなければならない．そこで，表 Table をキー (Key) と対応する値 (Value) をパラメータとす

る型に変更する．

```
type Table k v = [(k,v)]
emptyTable :: Table k v
```

それに伴い，表に対する操作関数の型は，それぞれ，

```
lookupTable :: Ord k => k -> Table k v -> [v]
insertTable :: Ord k => k -> v -> Table k v -> Table k v
```

となる．型宣言にある Ord k => は文脈 (Context) という．この文脈は k が Ord クラスのインスタンスであることを前提条件に定義されている型であることを示す．この文脈が必要なのは，表を引く際および表に新しいエントリを登録する際にキーを比較しているからである．最初の Table の定義ではキーが Ord クラスのインスタンスである Integer に固定されていたので，文脈を書く必要がなかった．型の表現は変更されたが，操作関数の実装はなにも変更の必要はない．このように表の型をキーの型と値の型とでパラメータ化しておくと，表に関する述語や操作関数そのままで，キーや値が異なる表 (Table) を扱うことができて再利用しやすくなる．

memocc では cc の 2 つの引数を 1 つの引数にまとめる．これで memocc の引数の型と利用する表のキーの型が一致し分かりやすくなる．

```
memocc :: (Amount,[Coin]) -> Table (Amount,[Coin]) Count
                          -> (Count, Table (Amount,[Coin]) Count)
```

実装部分は型に合わせて，引数の部分と内部で自分自身を再帰的に呼んでいる部分を変更する．

```
memocc (0,_ ) tbl = (1,tbl)
memocc (_,[]) tbl = (0,tbl)
memocc arg@(a,ccs@(c:cs)) tbl
 | a < 0     = (0,tbl)
 | otherwise = case lookupTable arg tbl of
                 (v:_) -> (v,tbl)
                 []    -> let
                            (cnt1,tbl1) = memocc (a-c,ccs) tbl
                            (cnt2,tbl2) = memocc (a  , cs) tbl1
                            cnt3 = cnt1 + cnt2
                            tbl3 = insertTable arg cnt3 tbl2
                          in (cnt3,tbl3)
```

ドライバも修正する．

```
evalMemoCC :: Amount -> [Coin] -> Count
evalMemoCC amount coins = fst (memocc (amount,coins) emptyTable)
```

メモ化高階関数

引数をまとめた memocc の型をよく観察するとまだ抽象化できそうな共通部分が見える．memocc の型は，

```
a -> Table a b -> (b, Table a b)
```

というパターンの型であり，これは

```
a -> (Table a b -> (b, Table a b))
```

と同じ意味である．次に

```
Table a b -> (b, Table a b)
```

の意味を考える．これは，「b 型」の値を使うかわりに，「表 Table a b 型の値をもらって，b 型の値と表 Table a b 型の値とのペアを返す関数の型」の値を使うという意味である．このことは，関心のあるデータ (t 型) とは別に，変化する状態 (s 型) を次へと伝えていく仕組みとして一般化できる．

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

withState :: t -> State s t
withState x = \ state -> (x,state)

bindState :: State s t -> (t -> State s u) -> State s u
bindState sx sf s0 = let (x,s1) = sx s0 in sf x s1

evalState :: State s t -> s -> t
evalState s s0 = fst (s s0)
```

withStateは，関心のあるデータ（t型）から，状態（s型）をもらって関心のあるデータと状態とのペアを返す仕組み（State s t型）への変換関数である．bindStateは状態を受渡す機構である．evalStateはState s t型のデータ（実は関数）に初期状態を与えて計算を始めて，結果を得るための関数である．

次に通常の1引数関数a -> bおよび2引数関数a -> b -> cをState sの世界の関数State s a -> State s bおよびState s a -> State s b -> State s cにそれぞれ変換する高階関数を定義する．

```
fun1WithState :: (a -> b) -> State s a -> State s b
fun1WithState f sx = bindState sx (\ x -> withState (f x))
fun2WithState :: (a -> b -> c) -> State s a -> State s b -> State s c
fun2WithState f sx sy = bindState sx (\ x ->
                        bindState sy (\ y ->
                        withState (f x y)))
```

a -> Table a b -> (b, Table a b)のパターンを書き直すとa -> State (Table a b) bとなる．すなわち

```
a -> b
```

という関数のメモ版は

```
a -> State (Table a b) b
```

と書き表すことができる．いよいよメモ化の核心部分，「関数適用時に，表を引く，表に追加する」を高階関数memoiseに抽象化する．

```
memoise :: Ord a => (a -> State (Table a b) b) -> a -> State (Table a b) b
memoise f x tbl = case lookupTable x tbl of
                    y:_ -> (y,tbl)
                    []  -> let (y,tbl') = f x tbl
                           in  (y,insertTable x y tbl')
```

さらにwithState，bindState，memoiseを使ってmemoccを書き直すと，

```
memocc (0,_ ) = withState 1
memocc (_,[]) = withState 0
memocc arg@(a,_)
  | a < 0      = withState 0
  | otherwise  = memoise (\ (a,ccs@(c:cs)) ->
                            bindState memocc (a-c,ccs) (\ cnt1 ->
                            bindState memocc (a  ,cs ) (\ cnt2 ->
                            withState (cnt1 + cnt2)))
                         ) arg
```

高階関数fun2WithStateを使うとmemoccはさらに簡潔に書ける．

```
memocc (0,_ ) = withState 1
memocc (_,[]) = withState 0
memocc arg@(a,_)
  | a < 0     = withState 0
  | otherwise = memoise (\ (a,ccs@(c:cs)) -> memocc (a-c,ccs) `add` memocc (a,cs)) arg
   where add = fun2WithState (+)
```

ここまで来たらもうひとふんばりしよう．上では，fun2WithState で型 Count 上の演算子 (+) を変換して State (Table (Amount,[Coin]) Count 上の演算子 add に変換した．実は型 State (Table a b) b が Num クラスのインスタンスであると宣言してあれば，明示的な変換の必要はない☆2．

```
instance Num b => Eq (State (Table a b) b) where
  sx == sy    = (evalState sx emptyTable) == (evalState sy emptyTable)

instance Num b => Show (State (Table a b) b) where
  show sx     = show (evalState sx emptyTable)

instance Num b => Num (State (Table a b) b) where
  (+)         = fun2WithState (+)
  (-)         = fun2WithState (-)
  (*)         = fun2WithState (*)
  negate      = fun1WithState negate
  abs         = fun1WithState abs
  signum      = fun1WithState signum
  fromInteger = withState . fromInteger
```

Num クラスは同時に Eq クラスと Show クラスのインスタンスでデータ型に対して規定されるクラス[2] なので，Memo a b も Eq クラスと Show クラスのインスタンスとして宣言する必要がある．

できあがった memocc の定義と，memocc の定義に似せて書き換えたメモ化する前の cc の定義を比較する．State (Table (Amount,[Coin]) Count) Count が Num クラスのインスタンスなので withState 1 あるいは withState 0 と書ける場所では単に 1 あるいは 0 と書くことができる．また，fun2WithState (+) と書ける場所では，(+) と書くことができる．

```
memocc (0,_ ) = 1
memocc (_,[]) = 0
memocc arg@(a,_)
 | a < 0      = 0
 | otherwise  = memoise (\ (a,ccs@(c:cs)) -> memocc (a-c,ccs) + memocc (a,cs)) arg

cc (0,_)  = 1
cc (_,[]) = 0
cc arg@(a,_)
 | a < 0      = 0
 | otherwise  = ($) (\ (a,ccs@(c:cs)) -> cc (a-c,ccs) + cc (a,cs)) arg
```

$ は関数適用演算子で f $ x および ($) f x はともに f x と書くのと同等である．これが memoise と対応している．すなわち，memoise は特殊な関数適用と考えることができる．また，a -> b のメモ版

```
type Memo a b = a -> State (Table a b) b
```

と定義すると，memocc の型は，

```
Memo (Amount,[Coin]) Count
```

と書ける．一方，上の cc の型は (Amount,[Coin]) -> Count であるが，これは

```
(->) (Amount,[Coin]) Count
```

と解釈できる．すなわち，Memo と (->) とが対応し，型の上でも，memoise は特殊な関数適用であることが理解できる．

☆2 Haskell 98 の仕様では，型の別名に対するインスタンス宣言は許されていない．ここ以降のコードを試すには，hugs を -98 オプション付きで起動するか，ghci を -fglasgow-exts フラグ付きで起動する必要がある．

動的計画法への応用

メモ化は，動的計画法 (Dynamic Programming) と呼ばれるプログラミング技法の 1 つである．上で定義したメモ化に関する高階関数を使って，典型的な動的計画法の対象となる問題を解いてみよう．

最長共通部分系列問題

与えられた系列の部分系列とは，与えられた系列からいくつかの要素（0 個でもよい）を取り除いた系列のことをいう．ここでは系列を文字列と読み替えておく．

問題　与えられた 2 つの文字列の最長共通部分文字列を求めよ．

まず，樹状再帰定義．これは問題を素直に記述したものである．

```
lcs :: String -> String -> (Integer,String)
lcs "" _  = (0,"")
lcs _ ""  = (0,"")
lcs xxs@(x:xs) yys@(y:ys)
 = if x == y
      then cons x (lcs xs ys)
      else maxlen (lcs xs yys) (lcs xxs ys)

cons :: Char -> (Integer,String) -> (Integer,String)
cons x (lx,xs) = (lx+1,x:xs)

maxlen :: (Integer,String) -> (Integer,String) -> (Integer,String)
maxlen xs@(lx,_) ys@(ly,_)= if lx > ly then xs else ys
```

この定義の意味は直截で分かりやすく，説明を要しない．しかし，実行効率は非常に悪い．

```
*Main> lcs "ascii" "asian"
(3,"asi")
(0.01 secs, 269776 bytes)
*Main> lcs "algorithm" "assertions"
(3,"ari")
(0.04 secs, 443880 bytes)
*Main> lcs "implementations" "constructively"
(4,"ntti")
(60.73 secs, 386384600 bytes)
```

最後の `lcs "implementations" "constructively"` になると少し待たなければならない．これを `memoise` および一連の高階関数を利用してメモ化する．

```
memolcs :: Memo (String,String) (Integer,String)
memolcs ("",_) = withState (0,"")
memolcs (_,"") = withState (0,"")
memolcs xxsyys
 = memoise (\ (xxs@(x:xs),yys@(y:ys))
            -> if x == y
                  then consS (withState x) (memolcs (xs,ys))
                  else maxlenS (memolcs (xs,yys)) (memolcs (xxs,ys))
           ) xxsyys
   where consS   = fun2WithState cons
         maxlenS = fun2WithState maxlen

evalMemoLCS :: String -> String -> (Integer,String)
evalMemoLCS xs ys = evalState (memolcs (xs,ys)) emptyTable
```

先程と同じ問題で実行すると以下のとおり

```
*Main> evalMemoLCS "ascii" "asian"
(3,"asi")
(0.00 secs, 272800 bytes)
*Main> evalMemoLCS "algorithm" "assertions"
(3,"ari")
(0.02 secs, 562352 bytes)
*Main> evalMemoLCS "implementations" "constructively"
"ntti"
(0.12 secs, 2193968bytes)
```

今度はすぐに答えが返ってくる．

メモ化によって，指数オーダの計算手間のかかるプログラムを多項式オーダ，場合によっては線型オーダにまで改善できることもある．ただし，メモ化による改良が最善のものでないことも多い．ここで手にしたメモ化高階関数の有用性は，自明あるいは簡単に理解できる再帰定義の構造を保存したまま効率を劇的に改善できる点にある．動的計画法について触れているアルゴリズムの教科書（たとえば，“Introduction to Algorithms”[3)]）にはいくつもの問題が載っているので，これらに適用してプログラムしてみてほしい．そうすれば高階関数の便利さが実感できるはずだ．

参考文献

1）Abelson, H., Sussman, G. J. and Sussman, J.: Structure and Interpretation of Computer Programs - 2nd ed., MIT Press (1996).
2）Jones, S. P.: 6.3. Standard Haskell Classes, Haskell 98 Language and Libraries The Revised Report, Cambridge Univ. (2003).
3）Cormen, T. H., Leiserson, C. E. and Rivest, R. L.: Introduction to Algorithms 2nd ed., MIT Press (2001).

（平成17年6月22日受付）