メモ化の技法について

奥居　哲

1　はじめに

問題を部分問題に分解する**分割統治**はアルゴリズム設計における基本的な考え方である．元の問題をサイズのより小さい同種の部分問題に分解することで簡潔な**再帰的解法**が得られる．ただ，分解の方法が線形でない場合には同じ部分問題が繰り返し出現することがあり，これが冗長な再計算の原因となる．そこで再計算を回避するために部分問題を（n 次元の）表として配置し，サイズの小さい問題から順に系統的に計算するのが**動的計画法**である．

部分問題毎に計算結果を保存し再計算時にはその結果を参照する**メモ化**（memoization）の技法も広い意味で動的計画法に含められることがある．不要な部分問題の値が一度も計算されないことは，メモ化を用いる動的計画法の優れた点であると言える．

本稿では，Haskell のような遅延評価に基づく関数型言語におけるメモ化の技法について簡単にまとめる．

2　遅延評価によるメモ化の実現

破壊的代入を許さない純粋な**関数型言語**では，表の項目の更新を必要とするメモ化は一般に実現が難しいものと思われがちである．[1] では表の更新を副作用と捉え**状態モナド**を用いて実現している．

実際には，Haskell のような**遅延評価**に基づく関数型言語におけるメモ化の実現は容易である．関数に与えられる実引数の計算は，それが必要な場合にだけ，しかも高々一度だけ行なわれる．これを利用すれば，必要な部分問題の計算だけを高々一度だけ行なうことが可能である．例えば Ford の **packrat parsing** は，バックトラックに基づく**再帰下降型構文解析**における再計算の回避にこの方法を用いている [2]．

以下ではフィボナッチ数列の例を用いてこの方法を説明する．フィボナッチ数列の各項を計算する関数は Haskell で

```
fib :: Integer -> Integer
fib 0 = 1
fib 1 = 1
fib n = fib (n - 1) + fib (n - 2)
```

のように定義できるが，二分再帰ゆえに部分問題の数が指数的に増加する．一方，`fib` n を計算するには n 個の部分問題 `fib` $(n-1), \ldots,$ `fib 0` を計算するだけでよいので，ほとんどの計算は同じ部分問題の再計算に費やされていることになる．そこでメモ化を導入する．

基本的なアイデアは，メモ化したい関数の定義域のデータ型を拡張し，部分問題の計算結果を含めることである．関数 `fib` の定義域は自然数なので以下のように帰納的に定義できる．

```
data Nat = Zero | Succ Nat
```

これに従い，もとの関数は以下のように書き直せる．

```
fib :: Nat -> Integer
fib Zero            = 1
fib (Succ Zero)     = 1
fib (Succ (Succ n)) = fib (Succ n) + fib n
```

次に定義域のデータ構造にメモ化される値を含める．

```
data MNat a = Zero | Succ a (MNat a)
```

ここで `a` は，メモ化される値の型を意味する型変数である．`a` は部分問題の計算結果に対応しているので，`(MNat a)` を引数に取るコンストラクタ `Succ` に付加されるが `Zero` には不要である．

その上で，再帰呼出しを行なう代わりに定義域に含まれるメモ化された値を参照するように関数 `fib` を変更すると以下のようになる．

```
fib :: MNat Integer -> Integer
fib Zero                  = 1
fib (Succ m Zero)         = m
fib (Succ m1 (Succ m2 _)) = m1 + m2
```

最後に与えれた自然数に対して `MNat Integer` 型のメモ化データ構造を作成する補助関数 `tabulate` が必要である．

```
tabulate 0 = Zero
tabluate n = Succ (fib t) t
```

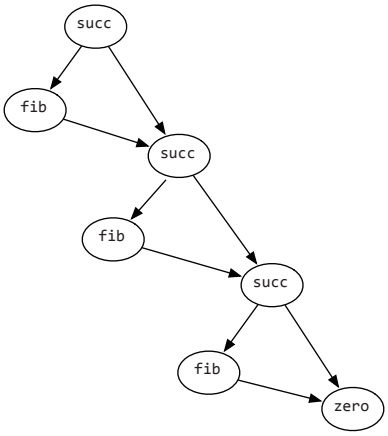

図 1　tabulate 3 によって生成されるデータ構造（簡略化して図示）

```
where t = tabulate (n - 1)
```

図 2 は，tabulate 3 によって生成されるデータ構造を簡略化して図示したものである．メモ値が格納される場所に部分問題の計算を表す式が配置されていることが分かる．また，このデータ構造は非循環グラフであり各部分問題が重複して出現することはない．仮に上記の tabulate の定義の代わりに以下のようにしたならば，部分問題が指数的に重複して出現する木構造が得られ，当然ながらメモ化の役割を果たさなくなる．

```
tabulate' 0 = Zero
tabluate' n = Succ (fib $ tabulate' $ n - 1)
                   (tabulate' $ n - 1)
```

以上のもとで，fib.tabulate は元の fib と同じ値を返すが，その計算効率は著しく異なる．元の fib では事実上計算できない入力の値（例えば 10000）にたいしてもメモ化されたほうではミリ秒のオーダで計算が終了する．

データ構造 MNat a は Zero を Nil に，Succ を Cons と読み変えれば，リストでも実現できる．さらには配列を用いるほうが，この問題の場合は最も簡単である．しかしながら，配列を用いる方法は部分問題が n 次元の表として配置できることが明確な問題でないと適用が難しい．一方で先に述べた方法は，任意の代数的データ構造に対して系統的に（機械的に）適用できるので汎用性がある．

3　トライ木の一般化に基づく技法

前節で述べた方法では与えられた入力毎にメモ化のデータ構造を作成した．これに対して，すべての入力値をメモ化できる無限データ構造を作成する方法がある．以下は無限リストを用いてフィボナッチ関数をメモ化する良く知られた方法である（原典は [5]）．

```
fibs = [fib n | n <- [1..]]
fib 0 = 1
fib 1 = 1
fib n = fibs !! (n - 1) + fibs !! (n - 2)
```

無限リスト fibs が，フィボナッチ数列のすべての項に対応する無限長のメモ値表の働きをしている．

[3] では，以下のような型の同型規則を用いて関数の定義域のデータ構造からメモ化のためのデータ構造を系統的に導出するエレガントな方法を与えている．

$$1 \to T \cong T$$
$$(S_1 + S_2) \to T \cong (S_1 \to T) \times (S_2 \to T)$$
$$(S_1 \times S_2) \to T \cong S_1 \to (S_2 \to T)$$

例えば，自然数の帰納的定義からは上記のような無限リスト構造が導出され，文字列の帰納的定義からはトライ木（の一般化）が得られる．

本節で述べた方法は，メモ値表を一旦つくれば，メモ化したい関数の定義域全体をカバーする点で前節の方法より優れている．

一方，本節の方法では部分問題の検索があまり効率的ではない．例えば上のフィボナッチの例ではリストの線形探索が行なわれる．この例に関しては，部分問題を無限木上に配置して検索を高速化する工夫 [4] があるが，汎用性は低い．前節の方法では，部分問題が定義域のデータの再帰的構造にしたがって配置されているので，動的計画法を実現する際の部分問題の検索効率に優れている．

参考文献

[1] Brown, D. Cook, W.R.: Function Inheritance: Monadic Memoization Mixins, Proc. Brazilian Symposium on Programming Lnaguage (2009).

[2] Ford, B.:Packrat parsing:simple, powerful, lazy, linear time, Proc. ICFP'02, pp.36–47 (2002).

[3] Hinze, R.:Memo Functions, Polytypically!, Proc. 2nd Workshop on Generic Programming, pp.17–32 (2000).

[4] Memoization, http://www.haskell.org/haskellwiki/Memoization/ (2013).

[5] Turner, D.A.:The semantic elegance of applicative languages, Proc. FPCA'81, pp.85–92 (1981).