データ構造の畳み込みと展開

1. はじめに

こんにちは、yingtaiです。今回はデータ構造の畳み込み (folding) について書きたいと思います。

畳み込みは、関数型プログラミングに欠かせない概念と言えるでしょう。特にリストの畳み込み (Pythonのreduce, Rubyのinject, C++のaccumulate, Haskellのfoldr, etc.) は有名です。

今回は畳み込み、およびその逆の展開、あるいは畳み込みと展開の合成、およびリスト以外の畳み込みなどについて簡単に紹介したいと思います。

2. 畳み込み

まず、Haskellのfoldr関数を例に説明します。foldrは引数を3つ取ります——2引数関数、初期値、リストです。

最も簡単な例はsumでしょう。

```
foldr (+) 0 [1,2,3,5,8]
    = 0+(1+(2+(3+(5+8))))
    = 19
```

(+) の代わりに (-) を入れると、こうなります。

```
foldr (-) 0 [1,2,3,5,8]
    = 0-(1-(2-(3-(5-8))))
    = 5
```

foldlはfoldrと畳み込む方向が逆になる関数です。

```
foldl (-) 0 [1,2,3,5,8]
    = ((((0-1)-2)-3)-5)-8
    = -19
```

foldr関数を使って基本的なリスト操作関数を定義してみます。

```
map f = foldr (\x y->f x:y) []

(++) = flip $ foldr (:)

concat = foldr (++) []
```

畳み込みが強力な操作であることが分かると思います。

3. 展開

Haskellにおいて、foldrに対応する展開関数はunfoldrです。

こちらはあまり馴染みがないかも知れません。関数型言語以外でunfoldrに相当する機能をもつ言語はあまりないと思います。

GHCでもfoldrはPreludeに定義されていますが、unfoldrはData.Listに入っています。

Data.List.unfoldrの定義は以下のようになっています。

```
unfoldr      :: (b -> Maybe (a, b)) -> b -> [a]
unfoldr f b  =
  case f b of
   Just (a,new_b) -> a : unfoldr f new_b
   Nothing        -> []
```

ここでのunfoldrは実用性を考えてMaybeが使われていますが、これをぱっと見ただけではfoldrとの対応が見えづらいです。

そこで、unfoldrに別の定義を与えてみます:

```
unfoldr' p f g b
```

```
  = if p b then []
           else f b : unfoldr' p f g (g b)
```

このバージョンでは、unfoldrは4つの引数を与えられています。pは述語で、展開の停止条件を表します。fはリストの先頭を与える関数、gは展開しながら適用していく関数です。bは展開する種となる値です。いくつかこれを使った面白い例を見てみます。

コラッツ数列は簡単です——条件関数で次の数を決め、1になったら停止します。

```
collatz = unfoldr' (== 1) id cond
    where cond x | odd x      = x*3 + 1
                 | otherwise = x`div`2

*Main> collatz 11
[11,34,17,52,26,13,40,20,10,5,16,8,4,2]
```

Data.List.unfoldrで同様に書くと以下のようになります。

```
collatz' = unfoldr f
  where
    f 1 = Nothing
    f x | odd x      = Just (x, x*3 + 1)
        | otherwise = Just (x, x`div`2)
```

(簡単のために上のような定義にしましたが、これでは出力する数列に最後の1が含まれません。もっとも、修正は簡単です。)

同様の方法で、漸化式で表せる数列がunfoldrで生成できることが分かると思います。分かりやすい例ではフィボナッチ数列とか。

```
fibonacci = unfoldr' (const False) fst (\(x,y)-
>(y,x+y)) (1,1)

*Main> take 10 fibonacci
[1,1,2,3,5,8,13,21,34,55]
```

ソートもできます。選択ソート。

```
ssort :: Ord a => [a] -> [a]
ssort = unfoldr' (==[]) minimum (\xs->delete (minimum
xs) xs)

*Main> ssort [1,8,3,5,2]
[1,2,3,5,8]
```

最小要素を除き、残りのリストで同様の操作を繰り返します。簡単ですね!

unfoldrを使うと、再帰の本質的な部分だけを記述することができます。

4. 畳み込みと展開の合成

畳み込みと展開があるならば、畳み込みと展開の合成もあります。

展開の後に畳み込むというパターンはhylomorphismと呼ばれます (これは一般化した圏論の概念です。同様にfoldは一般にcatamorphism, unfoldはanamorphismと呼ばれます) 。

```
hylo f e p g h = foldr f e . unfoldr' p g h
```

例えば、これで階乗を定義することができます。

```
fact = hylo (*) 1 (==0) id pred
```

なお、hyloはfoldr, unfoldrを使わずに書き換えることが可能です。これはhyloがリストの中間データを持つ必要がないことを意味します。

```
hylo f e p g h b
  = if p b then e
           else f (g b) (hylo f e p g h (h b))
```

5. 自然数と畳み込み

リスト以外のデータ構造にも、畳み込み/展開は応用できます。
例えば、自然数は次のようなデータ型で定義できますが:

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

これはリスト構造のより単純な (値を含まない) バージョンとして考えることができます。

```
-- cf.
data List a = Nil | Cons a (List a)
```

では、Nat上で畳み込みと展開を定義してみます。

```
foldN z s Zero     = z
foldN z s (Succ n) = s (foldN z s n)

unfoldN p f x
  = if p x then Zero
           else Succ (unfoldN p f (f x))
```

これらを使ってNat上での演算を定義できます。

```
predN Zero     = Nothing
predN (Succ n) = Just n

addN x = foldN x Succ
mulN x y = foldN Zero (addN y) x
subN x = foldN (Just x) (>>= predN)
divN x y = fromJust $ predN $
 unfoldN isNothing (>>= (flip subN y)) (Just x)
```

除算が少し汚いのは気にしない方針で...。

また、foldNの引数の順序を変えると、関数をn回適用するイテレータを定義できます。

```
iter n f x = foldN x f n
```

実は、ここで (iter n) はチャーチ数になっています。

```
iter (Succ (Succ Zero)) f x
  = foldN x f (Succ (Succ Zero))
  = f (f x)
```

6. まとめ

ここまで読んでいただきありがとうございました。尺の都合で触れられませんでしたが、一般に、畳み込みと展開は再帰的データ型上の操作として定義できます。例えばHaskellの代数的データ型で表現された木構造はfoldableです。

畳み込みと展開の面白さを少しでも感じていただけたなら幸いです。

参考文献:

『関数型プログラミングの楽しみ』
Jeremy Gibbons & Oege de Moor 編
山下 伸夫 訳