Haskell プログラミング

～ 純粋関数型言語への誘い～

山本和彦
(株)インターネットイニシアティブ
kazu@iij.ad.jp

毎年少なくとも一つの言語を学習する

「Blub 言語」のパラドックス

- Paul Graham
  - 「普通のやつらの上を行け」
  - http://practical-scheme.net/trans/beating-the-averages-j.html

プログラミング言語はそれぞれが、
機械語から最も力のある言語までの連続した
抽象度のスペクトルのどこかに位置するのだ。

このプログラマ氏がパワーのスペクトルを見下ろしている時、
彼にはそうしているという自覚がある。
「Blub」よりも力の弱い言語は、明らかに力が弱い。
彼が慣れ親しんだ機能が無いからだ。
しかし、このプログラマ氏が反対の方向に目を転じた時、
彼は自分が見上げているのだということに気付かないのだ。
彼が目にするのは、変てこりんな言語ばかり。
多分、それらは「Blub」と同じくらいパワフルなんだろうけど、
どういうわけかふわふわしたおまけがいろいろついているんだ、
と思うだろう。
彼にとっては「Blub」で十分なのだ。
何故なら彼は「Blub」で考えているから。

「なぜ Haskell は重要か」

- Sebastian Sylvan
  - http://www.haskell.org/haskellwiki/Why_Haskell_matters

すべての言語を比較するには、パワーのスペクトルの頂点に
立つ必要があるという結論がおのずと導かれることは
言うまでもない。
私の考えでは、定義によりと言ってもいいぐらいの理由から、
命令型よりも関数型言語の方がパワーのスペクトルの頂点に
近い。
...
もしあなたが、パワーのスペクトルを高所から
見下ろすために、関数型言語を学ぼうと思うなら、
Haskell は最高の選択肢だ。

「Javaスクールの危険」

- Joel Spolsky 氏
  - http://local.joelonsoftware.com/mediawiki/index.php/Javaスクールの危険

私の知るSchemeとHaskellとCのポインタが使える人はみな、
Javaを使い始めて2日で経験5年のJavaプログラマより
いいコードを書くようになる。

しかしそのことが平均的な頭の鈍い人事部の連中には
理解できないのだ。

Haskell とは何か？

- Haskell 98 Language and Libraries
  The Revised Report

  Haskell は、遅延評価 関数型言語に関する長年の研究の結晶かつ頂点である

- 関数型言語
  - 関数を中心としたプログラミング
  - 関数は値のように扱える
- 純粋
  - 状態を持たない
  - 再代入や副作用はない
- Glasgow Haskell Compiler
  - コンパイラー
  - インタープリター
  - スクリプト

状態マシンにどっぷり
浸かっている人の疑問

(あなたのことですよ)

状態がなくて
プログラムなんか書けるの？

Haskell の答え

8割のプログラムは
発想を変えると書ける

残りの2割はモナドで書く

素朴な疑問

再代入がないと
どうやってループするの？

発想の転換(1)

- 数え上げることが本質なら、単にリストを作る

```
for ($i=0; $i<10; $i++) { print "$i\n"; }
```

→ `mapM_ print [0..9]`

- メモリーの節約が本質なら、気にしない

```
$i = 0;
while (<>) { $i++ }
print "$i\n"
```

→ `print . length . lines =<< getContents`

発想の転換(2)

- 繰り返しが本質なら再帰で書く

```
for ($i = 1; $i <= $n; $i++) {
  $ret = $ret * $i;
}
```

→

  - 単純な再帰版

```
fact n = if n == 1
         then 1
         else n * fact (n-1)
```

  - パターンマッチ版

```
fact 1 = 1
fact n = n * fact (n-1)
```

命令型言語では

低レベルな How を記述する

関数型言語では

高レベルな What を記述する

What って何？

定番の答え(1)

- クイックソート

```
qsort []     = []

qsort (p:ps) = qsort less ++ [p] ++ qsort more
    where less = filter (<p)  ps
          more = filter (>=p) ps

qsort [5,7,3,8,1,9,2,4,0,6]
→ qsort [3,1,2,4,0] ++ [5] ++ qsort [7,8,9,6]
```

- リストの内包表記版

```
qsort (p:ps) =  qsort [x|x<-ps,x<p]
             ++ [p]
             ++ qsort [x|x<-ps,x>=p]
```

- 注意
  - 新たなリストが次々と作られている
  - ソートされているリストに対しては $O(N^2)$

定番の答え(2)

- エラトステネスのふるい

```
primes = sieve [2..]

sieve (p:ps) = p : sieve ps’
    where
      notMultipleP n = n ‘mod‘ p /= 0
      ps’ = filter notMultipleP ps

seieve [2,3,4,5,6,7,8,9,10,11,12,13,14,15,16 …]
→ 2:seive [3,5,7,9,11,13,15 …]
→ 2:3:seive [5,7,11,13 …]
```

- リストの内包表記版

```
sieve (p:ps) =
  p : sieve [n|n <- ps, n ‘mod‘ p /= 0]
```

「なぜ関数プログラミングは重要か」

- John Hughes
  - http://www.sampou.org/haskell/article/whyfp.html

関数プログラミングは「～ではない」ということ
(代入はない、副作用はない、制御フローはない)を
多く語っているが、
「～である」ということはそれほど語っていない。
関数プログラマはどちらかというと人生の楽しみを否定し、
それにより高潔な者になることを望む
中世の修道僧のように思える。
もっと物質的な利益に興味のある者にとっては
これらの「利点」はあまりピンとくるものではない。

関数型プログラミング言語は
二つの新しいタイプの糊を供給する。
すなわち、高階関数と遅延評価である。

高階関数とは？

- 引数に関数を取る関数

```
even 2
→ True
filter even [0..9]
→ [0,2,4,6,8]

map even [0..5]
→ [True,False,True,False,True,False]
```

- Haskell の関数は、関数を返す関数

```
oddlist = filter odd

oddlist [0..9]
→ [1,3,5,7,9]
```

遅延評価とは？

- Haskell では関数呼び出しが遅延評価になる
- 値は必要になってから計算される
  - 必要なければ計算されない

```
2 ‘div‘ 0
→ *** Exception: divide by zero

func x y = x
func 1 (2 ‘div‘ 0)
→ 1
```

- Unix のパイプみたいな機能

```
% grep keyword file | wc -l

length $ filter even [1..100]
```

- 参照透明性
  - 関数には状態がないので、計算の順序から解放されている
  - 逆に計算の順番を指定するのは難しい
    - 出力で問題になる

遅延評価の神髄

- 遅延評価によって、終了条件の判定から解放される
- 例) `replicate`

```
replicate '*' 5 → "*****"
```

- 再帰版

```
replicate c 0 = []
replicate c n = c : replicate c (n-1)
```

- 遅延評価版

```
repeat '*'
→ ['*','*','*','*','*', …]

take 5 ['*','*','*','*','*', …]
→ ['*','*','*','*','*']
→ "*****"

replicate c n = take n $ repeat c
```

Haskell は生産性が高い?

- 関数は、状態を持たないたった一つの式からなる
  - 複数の式・文・ブロックを順に並べることはできない
- その関数はテストが容易
  - 小さいから
  - 状態を持たないから
  - 対話環境があるから
- 正しいと確信できる小さな関数を糊でつなぐ
  - 単純に関数を呼び出す → 遅延評価
  - 関数に関数を渡す → 高階関数

行を分解するプログラム(1)

- 仕様

```
decomp ':' "  foo :  bar : baz  "
→ ["foo","bar","baz"]
```

- 標準関数 break

```
break (==':') " foo : bar : baz "
→ (" foo ", ": bar : baz ")
```

- 分割

```
split c cs = case break (==c) cs of
                (w,"")  -> [w]
                (w,_:cs') -> w:split c cs'
```

  - _ には、':'などの c がマッチし、これを捨てる

```
split ':' "  foo :  bar : baz  "
→ ["  foo ","  bar "," baz  "]
```

行を分解するプログラム(2)

- 前後の空白を削る

```
dropWhile isSpace "  foo "
→ "foo "

break isSpace "foo "
→ ("foo"," ")

fst ("foo"," ")
→ "foo"

trim cs =
   fst $ break isSpace $ dropWhile isSpace cs

trim "  foo "
→ "foo"
```

- つなぎ合わせる

```
decomp c cs = map trim $ split c cs

decomp ':' "  foo :  bar : baz  "
→ ["foo","bar","baz"]
```

つまり、純粋関数型言語では、

関数がモジュール

モジュール性は
モジュール以上の意味がある。

～なぜ関数プログラミングは重要か～

強力な代数データ型

- 真偽値
```
data Bool = True | False
```
- 失敗するかもしれない計算
```
data Maybe a = Just a | Nothing
```
- エラーになるかもしれない計算
```
data Either a b = Left a | Right b
```
- 同じようにユーザも定義できる
- これはリテラル
  - 表示も入力も、このまんま！

Haskell の代数データ型は
すご過ぎて

おもちゃに見える

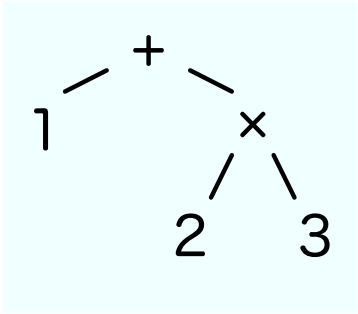
再帰的なデータ型の定義

- 数式を表現する再帰的な木構造

```
data Op = Plus | Multi deriving Show
data Exp = Leaf Int | Node Op Exp Exp
           deriving Show
```

- リテラルの例

```
Node Plus (Leaf 1) (Node Multi (Leaf 2) (Leaf 3))
```



再帰的なデータ型の走査

- 条件にあった部分をリストに変換する高階関数

```
flatten f l@(Leaf a) = f l
flatten f n@(Node o e1 e2) =  flatten f e1
                           ++ f n
                           ++ flatten f e2
```

- 条件を指定する関数

```
op (Leaf _) = []
op (Node o _ _) = [o]
```

- 木の中の演算子を得る

```
flatten op $ Node Plus (Leaf 1) (Node Multi (Leaf 2) (Leaf 3))
→ [Plus,Multi]
```

Haskell のパターン

- 代数データ型を定義する
- そのデータ型を走査する高階関数を書く
- 高階関数に条件を指定し、必要な部分をリストにする
- リストになれば、何でもあり
  - `filter`
  - `map`

厳格な Haskell

- あるレベルには、同じ型しか許されない
  - `if` には必ず `then` と `else` が必要
  - `then` と `else` が異なる型を返してはいけない
- リストの中は、同じ型の要素のみ
  - 異なる型を混ぜることはできない

  ○ `[0,1,2,3]`

  × `[False,True,2,3]`
- 強い静的型付け
  - (void *) などはない
  - 暗黙のキャストはない
  - コンパイラを通れば、型に関する間違いはない
- 型推論
  - 関数の型を全部書く必要はない
  - 分っている部分から、分らない部分を推論してくれる
  - 多相型も推論する

型推論と型検査

- あらかじめ定義されている型
```
putStrLn :: String -> IO ()
(++) :: [a] -> [a] -> [a]
```
- 型検査で怒られる例
```
main = putStrLn "Hello, " ++ "World!\n"
```
- 関数適用は強い結合力を持つから
```
(putStrLn "Hello, ") ++ "World!\n"
```
- 推論された ++ の型は
```
(++) :: IO() -> [Char] -> [Char]
```
- 正しくは
```
main = putStrLn $ "Hello, " ++ "World!\n"
```

コンパイルが通ったら
型に関する間違いはない

関数は小さく
誤りが入り込む余地は少ないので
コンパイルが通ったら
大抵の場合思い通りに動く

Haskell のコンパイラーは
プログラミングの偉大な先生

これまでで、できないこと

- 実行の順番を決める
- 副作用
  - 入出力
  - いちいち引数として渡さなくてもよいグローバル変数のようなものが欲しい

これらを解決するのが「モナド」

モナド

- 純粋関数型の枠組みで、副作用をエミュレートする
  - 実行の順番を決める
  - 暗黙の引数を実現する
- モナドとは、2つの関数を定義したデータ型
  - `return`
  - `>>=`
    - bind と読む
- モナドの肝は >>=
  - Haskell のもう一つの糊
  - cat コマンド

```
getContents :: IO String
putStr :: String -> IO ()

main = getContents >>= \cs -> putStr cs
```

    - IO と IO をくっつけて、より大きな IO を作る
    - 入力の後に出力される

do 表記

- >>= は読みにくい

```
main = getContents >>= \cs -> putStr cs
```

- 読みやすい do 表記が使える

```
main = do cs <- getContents
          putStr cs
```

- これは命令型の再発明では？
  - ある意味 YES
  - でも、Haskell の世界は壊していない
    - 再代入はない、状態はない
- Haskell の最も素敵な応用は Parsec
  - モナドパーサー

パーサーの問題点

- 正規表現
  - 再帰は表現できない
  - Perl の正規表現は Perl ではない
  - Perl コンパイラの恩恵は受けられない
  - 動かしてみるまで、正しいか分らない
  - 仕事は副作用でする ($1 など)
- Yacc/lex
  - 再帰は表現できる
  - Yacc/lex は C ではない
  - C コンパイラの機能の恩恵は受けられない

Parsec

- モナドパーサー
  - 小さなパーサーを >>= でつなぎ合わせる
  - パーサーそれぞれに仕事を定義できる
  - 完全に Haskell なので、Haskell コンパイラの恩恵が受けられる
  - 型推論と型検査

数値を解析するパーサー

- [+-]?[1-9][0-9]*

```
plus  = do char '+'
           return id

minus = do char '-'
           return negate

nOneOf xs = do c <- oneOf xs
               return $ digitToInt c --文字から数値

number = do pm <- plus <|> minus <|> return id
            d  <- nOneOf "123456789"
            ds <- many(nOneOf "0123456789")
            return $ pm $ foldl toNum 0 (d:ds)
    where
      toNum x y = x * 10 + y
```