純粋関数型言語 Haskellの紹介

～ 制約プログラミングのススメ ～

山本和彦

なぜバグは
なくならないのか？

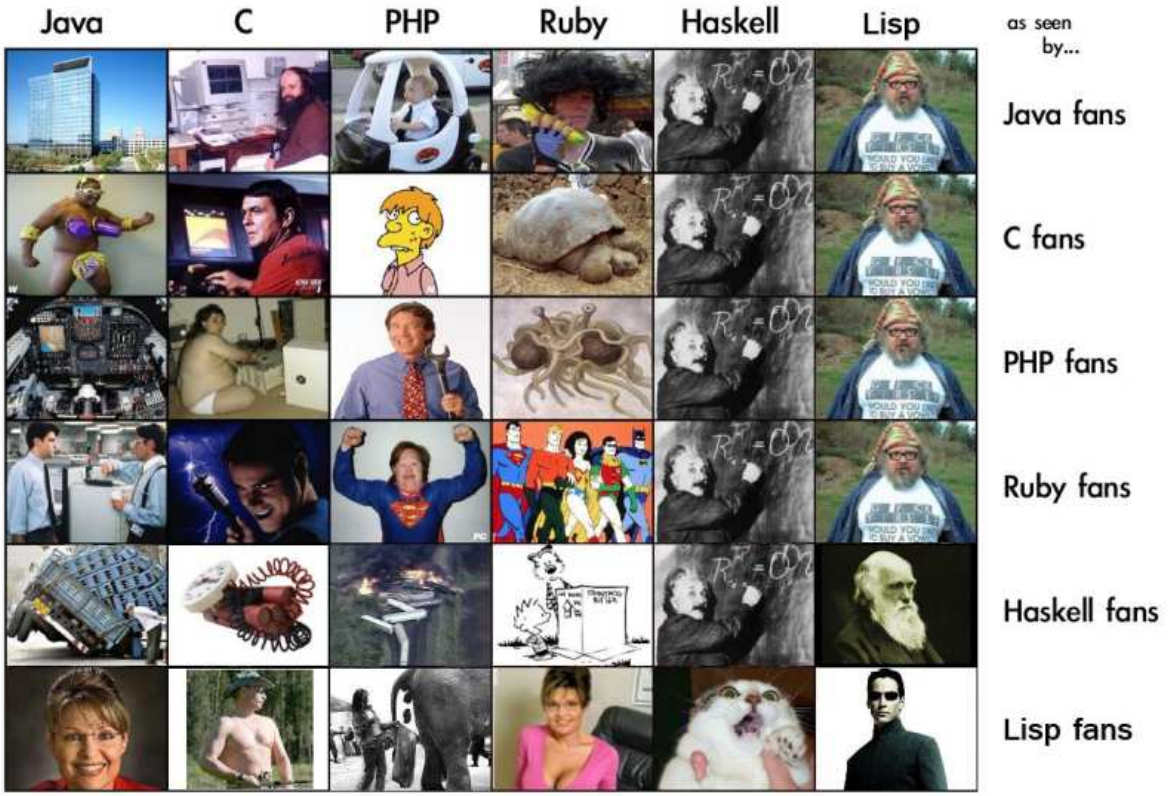
それはあまりにも自由に
プログラムを書くから

制約の下で
プログラムを書いて
バグをなくそう

今日の内容

- Haskell による制約プログラミングのススメ
  - 型に守られたプログラミング
- 当たり前だと思っていることを諦めると
  - 命令文の列挙
  - 破壊的代入(再代入)
  - 型のいい加減な利用
- 無理だと思っていることが手に入る
  - プログラムの品質
  - プログラムの生産性
    - デバッグで時間を無駄にしない
  - プログラムの保守性
    - 後から容易に変更できる
  - テストケースの自動生成
- 今回は説明しない機能
  - 遅延評価
  - 並列プログラミング

他人の言語はどう見えているか？



Haskell は数学、ただし Lisper だけは、
「かわいいけれど手に負えない」と思っている

Haskell は数学

=

Haskell の文法は制約されている

Haskell の文法

- Haskell の文法で記述できること
  - データ定義
  - 関数定義
  - 関数適用
  - 関数合成
- ないもの
  - ループの構文
  - 命令文の列挙
  - 破壊的代入
  - ...

関数適用

- 単純な関数

```
qsort [8,2,5,1]
→ [1,2,5,8]
```

- 二項演算子

```
"Hello, " ++ "world!"
→ "Hello, world!"
```

- 二引数の関数と二項演算子

```
1 + 2          div 8 2
(+) 1 2        8 ‘div‘ 2
→ 3            → 4
```

- 高階関数
  - 引数に関数を取る関数

```
map even [1,2,3,4]
→ [False,True,False,True]
```

関数合成

- 単純な関数合成

```
not . even
```

- バインド演算子による関数合成

```
getLine >>= putStrLn
```

関数定義

- 関数はたった一つの式からなる

```
main = putStrLn "Hello, world!"

average x y = (x + y) / 2
```

- if 文

```
collatz n = if even n
            then n ‘div‘ 2
            else n * 3 + 1
```

- 入り口での分岐と局所的な定義

```
subs [] = [[]]
subs (x:xs) = yss ++ map (x:) yss
  where
    yss = subs xs
```

- 関数合成

```
odd = not . even   -- odd n = not (even n)

getPutLine = getLine >>= putStrLn
```

命令的な記述はできない

- 命令文を列挙することはできない！

```
inp = getLine; -- 誤り
putStrLn inp;
```

- バインド演算子による関数合成 (再掲)

```
getLine >>= putStrLn -- 正しい
```

そんなので
プログラムが書けるの？

C++ で実装したクイックソート

```
template <typename T>
void qsort (T *result, T *list, int n)
{
    if (n == 0) return;
    T *smallerList, *largerList;
    smallerList = new T[n];
    largerList = new T[n];
    T pivot = list[0];
    int numSmaller=0, numLarger=0;
    for (int i = 1; i < n; i++)
        if (list[i] < pivot)
            smallerList[numSmaller++] = list[i];
        else
            largerList[numLarger++] = list[i];
    qsort(smallerList,smallerList,numSmaller);
    qsort(largerList,largerList,numLarger);
    int pos = 0;
    for ( int i = 0; i < numSmaller; i++)
        result[pos++] = smallerList[i];
    result[pos++] = pivot;
    for ( int i = 0; i < numLarger; i++)
        result[pos++] = largerList[i];
    delete [] smallerList;
    delete [] largerList;
};
```

Haskell で実装したクイックソート

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

- ポイント
  - 簡潔
  - How ではなく what を記述
  - 多相性 (リストの中身はなんでもよい)
- 対話的にテスト

```
> qsort [8,2,5,1]
[1,2,5,8]
```

注意

コードを理解しようとは
思わないで下さい

Haskell には
破壊的な代入はない

```
x = x + 1
```

に納得できなかったころを
思い出そう！

すなわち
関数は状態を持たない

状態がなくて
プログラムが書けるの？

たとえばループは？

状態のないプログラミングの例(1)

- Perl での数え上げ

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

- 数え上げることが本質なら、単にリストを作る

```
mapM_ print [0..9]
```

状態のないプログラミングの例(2)

- Perl でファイルの行数を数える
  - メモリーを節約するために一行ずつ読む

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

- メモリーの節約が本質なら、気にしない
  - Haskell ではファイル全体を読む

```
getContents >>= print . length . lines
```

状態のないプログラミングの例(3)

- Perl で階乗を計算する際はループを利用

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

- 繰り返しが本質なら再帰を利用

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

状態のないプログラミング

- 発想を変えると状態は不要になる
  - 状態が必要な問題を挙げることが難しくなる
- 状態は引数で渡す

関数プログラミングのこころ

- 状態を持たない小さな関数を書く
  - 関数はたった一つの仕事をする
- プログラムは小さな関数をつなぎ合わせて作る
  - 関数適用
  - 高階関数
  - (.) による関数合成
  - (>>=)による関数合成
- Haskell の関数は性質がよくテストが容易

テストケースの自動生成

QuickCheck

テストの自動化

- 世の中では
  手で書いたテストケースを自動的に走らせること
  をテストの自動化と言っているようだ

- Haskell はテストケースを自動生成する
  - QuickCheck では性質を記述する

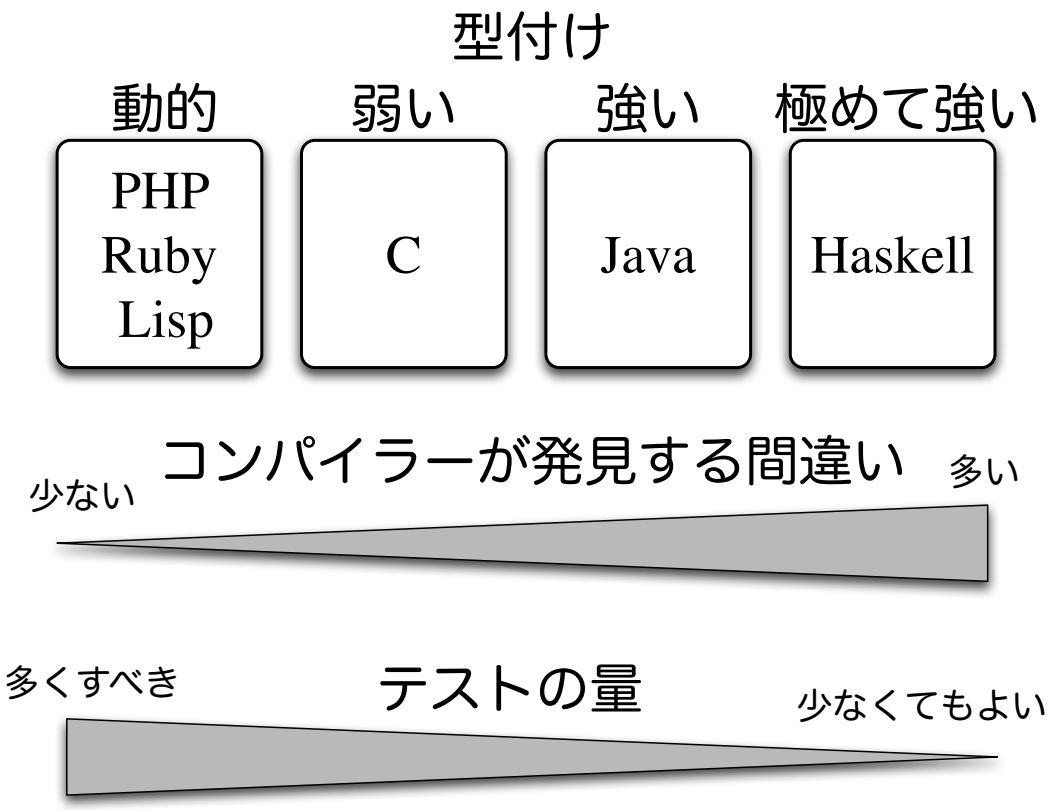
```
prop_ordered xs = ordered (qsort xs)
  where
    ordered []       = True
    ordered [x]      = True
    ordered (x:y:xs) = x <= y && ordered (y:xs)

> quickCheck prop_ordered
OK, passed 100 tests.
```

Haskell の型システム

静的で極めて強い型付け

型付けとテストの関係


型付け
動的
弱い
強い
極めて強い
PHP
Ruby
Lisp
C
Java
Haskell
コンパイラーが発見する間違い
少ない
多い
多くすべき
テストの量
少なくてもよい

コンパイラーに通った
Haskell のコードは概ね正しい

他の言語では経験できない感覚

型は関数の要約された説明

- 単純な関数

```
isDigit :: Char -> Bool
```

- 型変数を取る関数
  - 型変数 a にはどんな型も当てはまる
    - 注意) String は [Char]

```
(++) :: [a] -> [a] -> [a]
```

- 型クラス制約を持つ型変数を取る関数
  - 型変数 a には Int, Float, Double など

```
(+) :: (Num a) => a -> a -> a
```

  - 型変数 a には Int, Char, String など

```
qsort :: Ord a => [a] -> [a]
```

- 関数を取る関数

```
map :: (a -> b) -> [a] -> [b]
```

- IO を返す関数

```
getLine :: IO String
putStrLn :: String -> IO ()
```

if 文の制約

- if 文には必ず then と else の両方を書く
- then と else の返す型は同じ

```
collatz :: Int -> Int
collatz n = if even n
            then n `div` 2 -- これは Int
            else n * 3 + 1 -- これも Int
```

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!"
```
- 関数適用は演算子より強い結合力を持つから
```
main = (putStrLn "Hello, ") ++ "world!"
```
- 推論された ++ の型は
```
(++) :: IO() -> [Char] -> [Char]
```
- 正しくは
```
main = putStrLn ("Hello, " ++ "world!")
```

プログラムの生産性

コンパイラーが
多くの誤りを見つけてくれる

プログラムの保守性

誤った変更は
コンパイラーが禁止する

再帰で書く意味

- 再帰の末端では型検査を受ける

```
lines :: String -> [String]
lines "" = []
lines s  = let (l, s') = break (== '\n') s
           in l : case s' of
                    []      -> []
                    (_:s'') -> lines s''
```

型の定義

- ユーザ定義型は data を使って定義
```
data Tree a = Leaf a | Node (Tree a) (Tree a)
```
- 組み込み型の多くは、
ユーザ定義型と同じように定義されている
- 真偽値
```
data Bool = False | True
```
- 失敗するかもしれない型
```
data Maybe a = Nothing | Just a
```
- IO
```
data IO a = ...
```
- Haskell では、個々の型自体も安全
  - Maybe、IO

Maybe の意味

- On programming language design
  http://d.hatena.ne.jp/kmaebashi/20091223/p1#20091223f2
- NULL の仲間はバグから逃れられない
  - C の NULL, Java の null, Python の None, Lisp の nil, ...
  - NULL と正しい値は同じようにアクセスできるから、NULL を踏むことは避けられない
- Haskell の Maybe は安全

```
data Maybe a = Nothing | Just a
```

  - 正しい値にアクセスするには Just を外さないといけない

```
case mx of
  Nothing -> 0
  Just x  -> x + 1
```

  - 達人ならこう書く

```
maybe 0 (+1) mx
```

IO の性質

- getLine は実行ごとに異なる値を返す

```
% cat a.hs
main = getLine >>= putStrLn

% runghc a.hs
foo  ← ユーザーの入力
foo

% runghc a.hs
bar  ← ユーザーの入力
bar
```

- 明らかに純粋ではない！

IO の意味

- 純粋でない関数には IO という印が付く
- 純粋関数と IO の分離
  - IO から純粋関数を呼ぶことはできる

```
main = getContents >>= \inp ->
       putStrLn (count inp)
count :: String -> String
count = show . length . lines
```

  - 純粋関数から IO を呼ぶことはできない
- IO は IO とのみ連結できる

```
getLine  ::            IO String
putStrLn :: String -> IO ()
main     ::            IO ()
main = getLine >>= putStrLn
```

- 実行順序
  - IO は記述通りに実行される
  - 純粋関数の実行は記述通りではない (遅延評価)

構文糖衣 do

- do を使うと命令型言語のように見える

```
main = do           -- ::             IO ()
  inp <- getLine    -- ::             IO String
  putStrLn inp      -- :: String -> IO ()
```

- バインド演算子に直すと

```
main =
   getLine >>= \inp ->
   putStrLn inp
```

- 補助変数を削除すると

```
main = getLine >>= putStrLn
```

- 初心者のうちは do を使おう

Haskell は
世界で最も素晴らしい
命令型言語だ

Haskell の父
Simon PEYTON JONES

関数パーサー

Parsec

正規表現を使ったパーサーは
保守しにくいのに気づいた？

Parsec を使って
15分で Perl6 の完全なパーサーを
書く方法を勉強しましょう

Pugs の作者
Audrey Tang

正規表現 vs 関数パーサー

- 正規表現は字句解析用
  - 構文解析に用いると難解なコードになる
  - 保守は困難
- 関数パーサーは字句解析＋構文解析用
  - BNF を素直に実装できる
- CSV ファイルを例にとって説明

CSV の ABNF

- RFC 4180

```
file    = record *(CRLF record) [CRLF]
record  = field *(COMMA field)
field   = (escaped / non-escaped)
escaped = DQUOTE
          *(TEXTDATA / COMMA / CR / LF / 2DQUOTE)
          DQUOTE
non-escaped = *TEXTDATA
TEXTDATA = %x20-21 / %x23-2B / %x2D-7E
```

  - カンマを許すためにダブルクォートがある
  - ダブルクォートの中のダブルクォートは、
    ダブルクォートを重ねて書く

正規表現を使ったパーサー

- 詳説 正規表現 第一版より

```
sub parse_csv_line {
    my $text = shift;
    my @fields = ();
    while ($text =~ m/"([^"\\]*(\\.[^"\\]*)*)",?|([^,]+),?|,/g) {
        push(@fields, defined($1) ? $1 : defined($3) ? $3 : '');
    }
    push(@fields, '') if $text =~ m/,$/;
    return \@fields;
}

my @csv;
while (<>) {
    chomp;
    my $ret = parse_csv_line($_);
    push(@csv, $ret);
}
```

- 問題点
  - このコードでは「ダブルクォートの中のダブルクォート」はうまく扱えない
    - 修正されたコードは第二版以降を参照
  - ファイルの読み込みと構文解析を分離できない

Parsec で作る関数パーサー(1)

```
file    :: Parser [[String]]
file    = record ‘sepEndBy1‘ crlf

record  :: Parser [String]
record  = field ‘sepBy1‘ comma

field   :: Parser String
field   = escaped <|> nonEscaped

escaped :: Parser String
escaped = do
  dquote
  txt <- many (textdata <|> comma
           <|> cr       <|> lf
           <|> try (dquote >> dquote))
  dquote
  return txt

nonEscaped :: Parser String
nonEscaped = many textdata
```

Parsec で作る関数パーサー(2)

```
textdata :: Parser Char
textdata = oneOf (" !"
                 ++ ['#'..'+']
                 ++ ['-'..'~'])

comma    :: Parser Char
comma    = char ','

crlf     :: Parser Char
crlf     = cr >> lf

lf       :: Parser Char
lf       = char '\x0a'

cr       :: Parser Char
cr       = char '\x0d'

dquote   :: Parser Char
dquote   = char '"'
```

型に守られた関数パーサー

- Parser は Parser とのみ連結できる

```
dquote ::              Parser Char
many   :: Parser a -> Parser [a]
return :: a         -> Parser a

escaped :: Parser String
escaped = do
  dquote
  txt <- many (textdata <|> comma
           <|> cr       <|> lf
           <|> try (dquote >> dquote))
  dquote
  return txt
```

型を考えることがプログラミング

```
file   :: Parser [[String]]
file   = undefined

record :: Parser [String]
record = undefined

field  :: Parser String
field  = undefined
```

まとめ

- 制約プログラミング
  - 関数定義、関数適用、関数合成のみ
  - 関数は一つの式
  - 静的な極めて強い型付け
  - 純粋関数と IO の分離
  - 同じ型を連結する >>=
- 効用
  - プログラムの品質
  - プログラムの生産性
  - プログラムの保守性
  - テストケースの自動生成