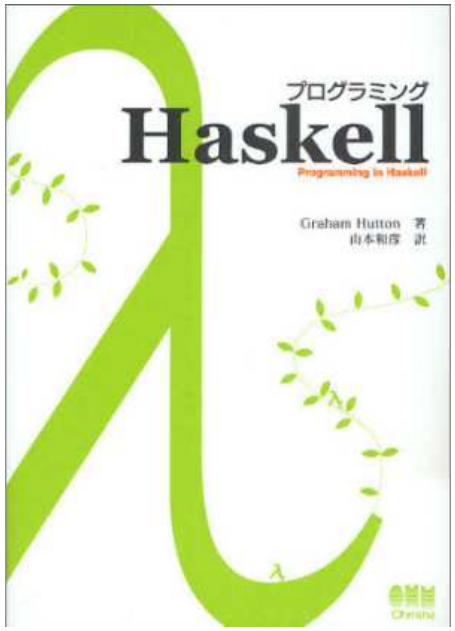
Haskell で
Behavior Driven Development

2012.5.27

山本和彦
@kazu_yamamoto

自己紹介



山本和彦

二児の父

Haskell 歴は4年と少し

Mew、Firemacs、Mighty、ghc-mod の開発者

Ruby/Java ユーザのつぶやき

- HackageDB のあるライブラリを見て

Haskeller は
テストコードを
あんまり書かない！

今日のお題

Haskeller のみなさん
もっとテストコードを
書きましょう

Q) なぜ Haskeller は
あまりテストコードを
書かないのか？

A) コンパイルが通れば
だいたい思い通りに動くから

なぜなら

コンパイルに通れば

引数の数に関する間違い

とか

関数名のタイポ

とか絶対にない

他の静的型付け言語のユーザ曰く

そんなの当たり前じゃない？

Haskeller 曰く

実はもっとすごい

Haskell は型安全なんです

型安全？

何それ？

役に立つ静的型付け

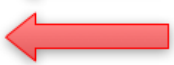
動的型付け言語	型は書かない エラーは実行時に見つかる
一般的な静的型付け言語	型を書く コンパイル時にあまり エラーが見つからない
静的型付け関数型言語	型を書く コンパイル時に多く エラーが見つかる

Haskell ではすべてが式！

```
(defun fibonacci (n)
  (let ((x 1) (y 1) (i 3))
    (while (<= i n)
      (setq y (+ x y))
      (setq x (- y x))
      (setq i (1+ i)))
    y))
```


式を文として利用


式を文として利用


式を文として利用

```
fibonacci :: Int -> Integer
fibonacci n = fib 1 0 1
 where
  fib m x y
    | n == m     = y
    | otherwise = fib (m + 1) y (x + y)
```

Haskell のコンパイルはテスト

- 文と文の型の関係は検査されない
- 式と式の型の関係は検査される
- Haskell ではあらゆる場所が検査される

```
fibonacci :: Int -> Integer
fibonacci n = fib 1 0 1
 where
  fib :: Int->Integer->Integer->Integer
  fib m x y
    | n == m     = y
    | otherwise = fib (m + 1) y (x + y)
```

推論された型

しかも

Haskell には型を台無しするものがない

言外の型変換

unsigned int + int

スーパーな型

何でも表せる型

void *, Object

スーパーなデータ

どんな型にもなれるデータ

NULL, null, nil, None

つまり

Haskell では、
コンパイルが通れば
型に関する間違いがない

大切なことなので二度言いますが

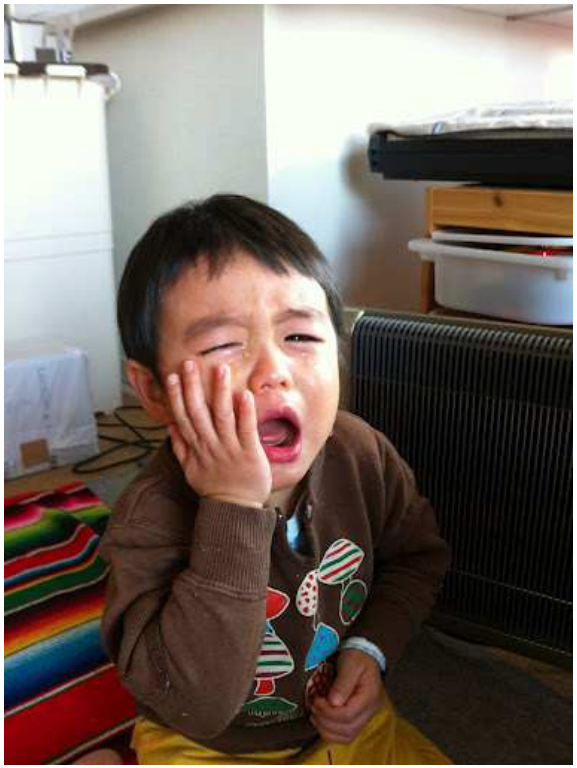
Haskell は
コンパイルが通れば
だいたい思い通りに動く

型安全ばんざーい \^^/

が、しかし

型に関する間違いがないとしても

値に関する間違いは存在する



Haskeller 曰く

HUnit のテストコードを
書くのは面倒だから
QuickCheck に
コーナーケースを見つけて
もらえばいいんじゃない？

QuickCheck って何？

- 関数の性質を記述する

```
prop_doubleSort :: [Int] -> Bool
prop_doubleSort xs = sort xs == sort (sort xs)
```

- テストケースを乱数で生成してくれる

```
> quickCheck prop_doubleSort
+++ OK, passed 100 tests.
```

純粋な関数は
性質を見つけやすい

自分が書いたコードの
性質は分かっているはず

訓練すればそれを表現できる

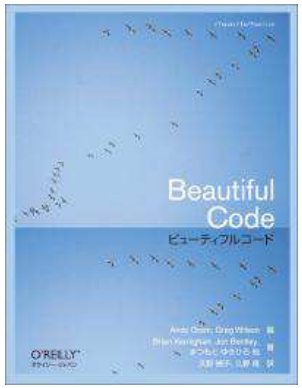
Haskell では型で純粋か分かる

- 純粋な関数
  - QuickCheck がおススメ

```
sort :: Ord a => [a] -> [a]
lookup :: Eq a => a -> [(a, b)] -> Maybe b
delete :: Eq a => a -> [a] -> [a]
```

- 副作用があるかもしれない関数
  - HUnit がおススメ(だった)

```
hGetLine :: Handle -> IO String
writeFile :: FilePath -> String -> IO ()
forkIO :: IO () -> IO ThreadId
```



「ビューティフルコード」
7章 ビューティフル・テスト

二分探索に対するテスト

二分探索のアイディアは
1946年に出された
バグがない実装ができたのは12年後

美しきテストたち

- 線形探索を使いながら二分探索をテストする

スモークテスト

境界テスト

ランダムテスト

突然変異テスト

でも、それって

二分探索が線形探索と同じ

って言ってるだけじゃん！

QuickCheck すごい

- QuickCheck だと、仕様はモデル実装と同じと表現するだけ！

```
prop_model x xs =
  linearSearch x xs == binarySearch x xs
```

QuickCheck がすごいとして

Haskeller には
テストを書きたくなる
仕組みが必要

Simon HENGEL さんは言った

「doctest あれ」

doctest って何？

- Python のドキュメントに利用例を書く仕組み

```
def factorial(n):
  """Return the factorial of n,
  an exact integer >= 0.
  If the result is small enough to fit in an int,
  return an int. Else return a long.

  >>> [factorial(n) for n in range(6)]
  [1, 1, 2, 6, 24, 120]
  >>> factorial(30)
  265252859812191058636308480000000L
  """
  import math
  if not n >= 0:
    raise ValueError("n must be >= 0")
   ...
```

- 利用例を自動的にテストできる

Haskell のドキュメント・ツールは

Haddock

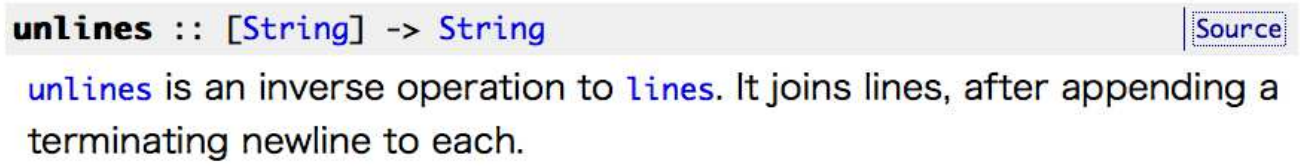
Haddock

- コメントの中にドキュメントを書く
  - 各種マークアップが定義されている

```
-- | 'unlines' is an inverse operation to 'lines'.
-- It joins lines, after appending a terminating
-- newline to each.

unlines :: [String] -> String
unlines [] = []
unlines (l:ls) = l ++ '\n' : unlines ls
```

- Haddock が生成する HTML



コードブロックは使えるか？

- コードブロック用のマークアプは ">"
- Data.Map より

```
-- | /O(1)/. Is the map empty?
--
-- > Data.Map.null (empty)           == True
-- > Data.Map.null (singleton 1 'a') == False

null :: Map k a -> Bool
null Tip              = True
null (Bin _ _ _ _ _) = False
```

- 利用例だと思うと、
  どこが式でどこが結果が分からない
- 性質だと思うと、利用例とは区別したい

マークアップに関する結論

- 利用例用のマークアップが必要
- 性質用のマークアップが必要

利用例用のマークアップ

- Simon HENGEL さんが ">>>" を導入
- 式と結果で利用例を記述する

```
>>> length []
0
```

- 命令シーケンスも書ける

```
>>> writeFile "tmpfile" "Hello"
>>> readFile "tmpfile"
"Hello"
```

- 例外も書ける

```
>>> head []
*** Exception: Prelude.head: empty list
```

- Haddock 2.8 以降で利用可能

doctest の実装

- Haddock コメントから利用例を切り出す
  - 0.5.2 以前は Haddock API を利用
  - 0.6 以降は GHC API を利用
- GHCi で評価して、文字列で結果を比較
  - doctest コマンドとライブラリ関数がある
- 0.6.x 以前は、関数ごとに GHCi を起動していた
- 0.7 以降は、モジュールごとに GHCi 起動する
  - 爆速です きりっ

性質用のマークアップ

- 山本和彦が "prop>" を導入
- パラメータのない性質

```
prop> Data.Map.null empty == True
```

- パラメータのある性質は無名関数で
  - 型は、型シグニチャで補う

```
prop> \xs -> sort xs == sort (sort (xs::[Int]))
```

- 無名関数のプレフィックスは省略したい

```
prop> sort xs == sort (sort (xs::[Int]))
```

- 将来の Haddock で利用可能になる予定

どうやってプレフィックスを補うか？

- haskell-src-exts でもパースできるけど...
- GHCi が教えてくれる！

```
> sort xs == sort (sort (xs::[Int]))
<interactive>:1:6: Not in scope: ‘xs’
<interactive>:1:24: Not in scope: ‘xs’
```

- この例では `"\xs -> "` を補えばよい

という訳で
将来の doctest では
利用例と性質が扱えます

名付けて

doctest による
設計、ドキュメント、自動テストの
三位一体化

あるいは

一粒で三度お美味しい

QuickCheck の余談

- v2.3 以前

```
> quickCheck $ length [] == 0
+++ OK, passed 100 tests.
```

  - 結果はメモ化されるので、True との比較が 100 回

- v2.4？ 以降

```
> quickCheck $ length [] == 0
+++ OK, passed 1 tests.
```

ところで
ドキュメントにふさわしくない
利用例/性質はどうするんですか？

たとえばチケットが切られた
コーナーケース

```
> valid $ deleteMin $ deleteMin $ fromList
    [(i,())|i <- [0,2,5,1,6,4,8,9,7,11,10,3]]
False
```

Trystan SPANGLER さんは言った

「Hspec あれ」

Q) Hspec って何？

A) Ruby の Rspec の Haskell 版
Rspec は BDD の旗手

Q) BDD (Behavior Driven Development)
って何？

A) TDD (Test Driven Development)
のテストコードを設計の言葉で書くこと

仕様書が自動テストとして使える

注意

Haskell のコードは静的なので
「振る舞い」という言葉は不適切かも

Hspec の例

- 先ほどのコーナーケース

```
describe "deleteMin" $ do
 it "maintains the balance even if applied doubly" $
  valid $ deleteMin $ deleteMin $ fromList
   [(i::Int,())|i <- [0,2,5,1,6,4,8,9,7,11,10,3]]

 it ...

 it ...

 it ...
```

IO の setup と teardown

- それ高階関数でできるよ！

```
withDB action = bracket
 (connectSqlite3 "my.db")
 disconnect
 action

describe "database adaptor" $ do
 it "returns 23, when 23 is selected" $
  withDB $ \connection ->
   run connection "select 23;" ‘shouldReturn‘ 23
```

材料が揃ったので Haskell での

Behavior Driven Development

を考えてみる

Haskell で BDD (1)

(1) シグニチャと関数名を書く

```
rev :: [a] -> [a]
rev = undefined
```

- 関数定義は undefined で
  - undefined では、実行時にどこで落ちたのか分からない
  - undefined の代わりに placeholders ライブラリの notImplemented と todo もおススメ
- 型のレベルで設計すること
  - そうすれば型が実装を導いてくれる

Haskell で BDD (2)

(2) Haddock スタイルでユーザ用の仕様を書く

```
-- |
-- 'rev' @xs@ returns the elements of @xs@
-- in reverse order. @xs@ must be finite.
--
-- >>> rev [1,2,3]
-- [3,2,1]
--
-- prop> rev [] == rev []
-- prop> rev (xs++ys) == rev ys ++ rev (xs::[Int])
-- prop> rev (rev xs) == (xs::[Int])

rev :: [a] -> [a]
rev = undefined
```

- 実行して文法間違いがないか確かめる

Haskell で BDD (3)

(3) Hspec スタイルで開発者用の仕様を書く

```
describe "rev" $ do
  it "returns the first element in the last" $
    property $ \xs -> not (null xs) ==>
      head (rev xs) == last (xs :: [Int])

  it "returns the last element in the first" $
    property $ \xs -> not (null xs) ==>
      last (rev xs) == head (xs :: [Int])
```

- 実行して文法間違いがないか確かめる

Haskell で BDD (4)

(4) 実装する

```
rev :: [a] -> [a]
rev = foldl (flip (:)) []
```

- doctest と Hspec のテストがすべて通るまで修正を繰り返す
- Cabal で自動化しておくのがおススメ
  - `cabal test`

```
test-suite doctests
    type:           exitcode-stdio-1.0
    main-is:        doctests.hs
    build-depends: base, doctest

test-suite spec
    type:           exitcode-stdio-1.0
    main-is:        Spec.hs
    build-depends: base, hspec
```

免責

- 僕は原理主義者ではありません
- 実装が簡単なら、まず実装して

```
rev = foldl (flip (:)) []
```

- ghc-mod などで推測したシグニチャを挿入し

```
rev :: [a] -> [a]
rev = foldl (flip (:)) []
```

- 後から例と性質を書いてもいいでしょう

```
-- |
-- >>> rev [1,2,3]
-- [3,2,1]
rev :: [a] -> [a]
rev = foldl (flip (:)) []
```

まとめ

QuickCheck を統合した
doctest と Hspec で
仕様(テストコード)を書き
Cabal でまとめて自動テストしよう！

一粒で三度おいしいよ！

おまけ

Simon HENGEL さんからのお願い
(doctest の作者、Hspec の貢献者)

Hspec に足りない機能があったら
github の issue に登録してね