

Beautiful Programming Language and Beautiful Testing

山本和彦

IIJイノベーションインスティテュート

純粋関数型プログラミング言語 Haskell に対して、三つのテスト基盤を統合したテスト手法を提案する。ライブラリのユーザに見せるべき振る舞いの記述には、Python から移植した doctest を用いる。doctest では、利用例、設計、および自動テストが実現できる。ユーザに見せるべきでない振る舞いの記述には、Ruby の rspec をヒントに実装した hspec を使用する。hspec では、設計および自動テストが実現できる。Haskell では、純粋なコードと副作用のあるコードが型システムにより明確に分離される。純粋なコードは、性質をテストしやすいことが知られている。我々は、性質テストツールである QuickCheck を doctest と hspec に統合した。

1 発表内容

次ページ以降のスライドを参照。

2 質疑応答

- ・ Q) オーダーのテストはタイムプロファイル取るのではダメか？
- ・ A) 本気でやるには、ある部分を何回と通ったかなどを数える必要がありますが、Haskell ではやりにくいです。
- ・ Twitter で見つけたよい答え) N を変えて実行時間の比を取るとよい。
- ・ Q) 型クラスを作るときにはその仕様を考えているので一緒に hspec を作ったりするという想定ですか？
- ・ A) そうです
- ・ Q) このテストはどういう形で実行するのか？
- ・ A) Haskell のパッケージ管理ソフト cabal のテスト機能を使って実行することを想定しています。

Beautiful Programming Language and Beautiful Testing



山本和彦

1

今日の話題

Haskell erto テストを書かせることを考える

Haskell erto
人向けのスライドです

2

Haskell ertoへの注意

今日話すことの一部は
リリースされていません
(Doctestの QuickCheck 対応の部分)

githubにはあります

3

動機

Haskell ertoはテストを書かない！

4

テストがあるかの調査

- Haskell Platform 2011.4.0.0
 - Glasgow Haskell Compiler + 2個のパッケージ
 - 19個のコアパッケージを除く
- 8個がテストを持つ
 - ただしテストケースは貧弱
- 5個が利用例を記述
 - 利用例をテストに活用してはいない
- 10個はテストも利用例もなし

5

Q) Haskell ertoはどうして
テストを書かないのか？

A) コンパイルが通れば
だいたい思い通りに動くから

6

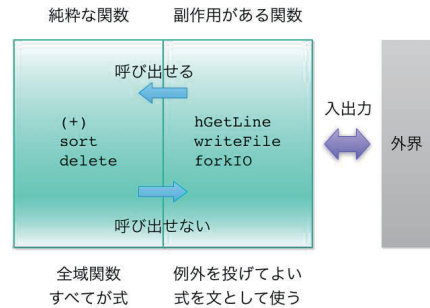
Q) どうして思い通りに
動くことが多いのか？

A) Haskellの言語仕様が
プログラマに規律を課すから

7

Haskellの規律

- 副作用のある関数とない関数を明確に分ける



8

Haskellでは型で純粋か分かる

- 純粋な関数

```
(+) :: Num a => a -> a -> a
sort :: Ord a => [a] -> [a]
delete :: Eq a => a -> [a] -> [a]
```

- 副作用がある関数

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

9

純粋な関数

- Haskellの純粋関数はすべてが式

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

```
(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))
```

10

Haskellのコンパイルはテスト

- 式と式の型の関係は検査される
 - Haskellの純粋関数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)
```

11

副作用がある関数

- 式を文として使う

```
daemonize :: IO () -> IO ()
daemonize program = ensureDetachTerminalCanWork $ do
  detachTerminal
  ensureNeverAttachTerminal $ do
    changeWorkingDirectory "/"
    void $ setFileCreationMask 0
    mapM_ [stdInput, stdOutput, stdError]
      closeFd
    program
  where
    ensureDetachTerminalCanWork p = do
      void $ forkProcess p
      exitSuccess
    ensureNeverAttachTerminal p = do
      void $ forkProcess p
      exitSuccess
  detachTerminal = void createSession
```

- I/Oのコードは、なるべく小さくする

12

HaskeII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

13

つまり

HaskeIIでは、
コンパイルが通れば
型に関する間違いがない

14

型安全

HaskeIIは
コンパイルが通れば
だいたい思い通りに動く

15

型に関する誤りはない
～ 整数と文字列を足す ～

しかし

値に関する誤りは残る
～ 0で割り算する ～

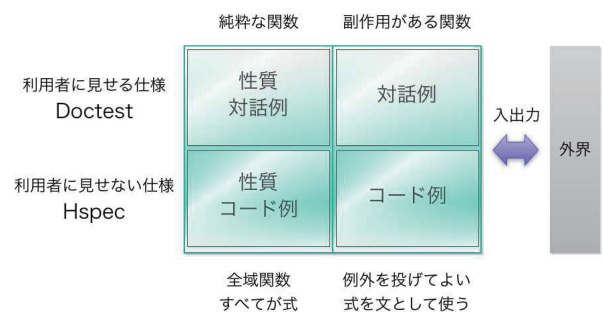
16

HaskeIIでもテストを書くべき

HaskeIIerにテストを書かせる
仕組みを考える

17

我々が提案するテスト体系



18

Doctest

- 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)
    26525285981219105863630848000000L
    """
    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

19

HaskeIIのドキュメント・ツールは

Haddock

20

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

```
unlines :: [String] -> String Source
unlines is an inverse operation to lines. It joins lines, after appending a
terminating newline to each.
```

21

対話例用のマークアップ

- ">>>" キーワード
- 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

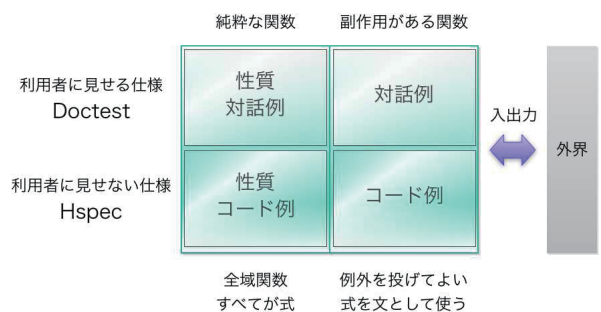

```
>>> 1 `div` 0
*** Exception: divide by zero
```

22

doctestは一粒で三度美味しい
設計、ドキュメント、自動テスト

23

我々が提案するテスト体系



24

Hspec

- Ruby の Rspec に触発された Haskell のテストツール
- Hspec でコード例を記述する

```
describe "sort" $ do
  it "sorts in the ascending order" $
    sort [3,7,1,5,2] 'shouldBe' [1,2,3,5,7]
  it "preserves a sorted list" $
    sort [1,2,3,5,7] 'shouldBe' [1,2,3,5,7]
```

25

楽しい Hspec

- xUnit系の HUnit の場合

```
do exist <- doesFileExist "foo.txt"
  exist @?= True

do ret <- try $ evaluate (1 'div' 0)
  ret @?= Left DivideByZero
```

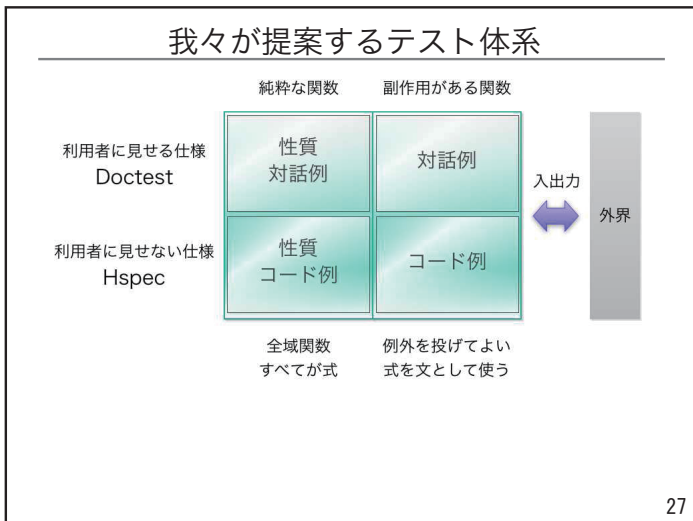
- Hspec の場合

```
do exist <- doesFileExist "foo.txt"
  exist 'shouldBe' True

doesFileExist "foo.txt" 'shouldReturn' True

evaluate (1 'div' 0)
  'shouldThrow' anyArithException
```

26



性質テスト

- Haskell では QuickCheck が有名
- 関数の性質を記述する

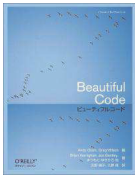
```
prop_doubleSort :: [Int] -> Bool
prop_doubleSort xs = sort xs == sort (sort xs)
```

- 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
 - 副作用のある関数は性質を見つけにくい

28



「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

29

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

30

Haskell にならなければ
二分探索が線形探索と同じ
とだけ

31

QuickCheck による美しきテスト

- 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
```

- 注意)加えてオーダーのテストも必要です

32

QuickCheck を組み込む

- Doctest
 - "prop>" キーワード

```
prop> Data.Map.null empty == True
prop> \xs -> sort xs == sort (sort (xs::[Int]))
```
- hspec
 - property コンビネータ

```
it "maintains a sorted list" $ property $ \xs ->
  sort xs == sort (sort (xs::[Int]))
```

33

我々が提案するテスト体系

	純粋な関数	副作用がある関数	
ユーザーに見せる仕様 Doctest	性質 対話例	対話例	入出力 ↔ 外界
ユーザーに見せない仕様 Hspec	性質 コード例	コード例	
	全域関数 すべてが式	例外を投げてよい 式を文として使う	

34

テストの再利用

- 型クラスがある法則を要求する場合がある
 - 型クラスとは、データ型のグループ
 - 同じ API を提供する
 - オブジェクト指向のクラスではない
- この法則を多相的に実装しておけば
テストを再利用できる

35

Monoid 則

- Monoid 則を Hspec で表現

```
shouldSatisfyMonoidLaws ::
  (Eq a, Show a, Monoid a, Arbitrary a) => a
  -> Spec
shouldSatisfyMonoidLaws t = do
  describe "mempty" $ do
    it "is a left identity" $ property $ \x ->
      mempty <> x == x 'asTypeOf' t
    it "is a right identity" $ property $ \x ->
      x <> mempty == x 'asTypeOf' t
  describe "<>" $ do
    it "is associative" $ property $ \x y z ->
      (x <> y) <> z == x <> (y <> z) 'asTypeOf' t
```

- Int のリストは Monoid 則を満たすべき

```
spec = do
  describe "Maybe as a Monoid" $ do
    shouldSatisfyMonoidLaws (undefined :: [Int])
```

36

まとめ

- 利用者に見せる仕様
 - Doctest + 性質 + 対話例
 - 設計、ドキュメント、自動テスト
- 利用者に見せない仕様
 - Hspec + 性質 + コード例
 - 楽しいテスト記述
- Haskell er はもっとテストを書きましょう