Advanced Programming Concepts using Haskell

An introduction

Tony Morris

Revision History

Revision $LastChangedRevision: 201 $	$LastChangedDate: 2009-05-22 22:36:09 +1000 (Fri, 22 May 2009) $
	Initial authoring

Abstract

This article is intended as a presentation for an introduction to advanced programming concepts using Haskell. This presentation introduces automated specification-based testing.

This presentation is hands-on and requires each member of the audience have access to a computer running the Glasgow Haskell Compiler (GHC) which is available from http://haskell.org/ghc.

Table of Contents

Sometimes asking #haskell for help can be like taking a drink from a fire hose -- Brent Yorgey

Get Started

Are we good to go?

```
$ ghci
GHCi, version 6.10.2: http://www.haskell.org/ghc/  :? for help
Prelude> length [Test.QuickCheck.variant]
1
```

Uh oh

```
Not in scope: `Test.QuickCheck.variant'
```

Haskell

- Haskell is a *purely-functional, lazy-by-default, statically-typed, type-inferred* programming language
- In contrast, Scala is an *impure, strict-by-default, statically-typed, (somewhat) type-inferred, (somewhat) functional* programming language
- Java (C, C++, C#, Python, Ruby, Groovy, etc.) is an *impure, strict-by-force, (somewhat — ok not really) statically-typed, (emphatically) imperative* programming language

Pure? Lazy?

- In a purely-functional language, each function may produce results by using only its arguments
- To access files, network or print, each function must denote this in the type (IO)
- The function `length :: [a] -> Int` acts only on the given list to produce an integer value

- The function `readFile :: FilePath -> IO String` acts on the given file-path, I/O and produces a string
- Function arguments are evaluated as they are needed *call-by-need*

Automated testing

- Manual testing: JUnit, NUnit, TestNG — poor coverage, clumsy, laborious, error-prone
- Automated testing: **QuickCheck**, ScalaCheck, Functional Java, FsCheck
- Automated testing: Correctness verification for smart people

A note on purity

Purity

- Leads to composition and abstraction
- Has other far reaching implications that we will only allude to today

See a problem?

```
if(itRocks) println("boo") else println("bar")
```

Improved code

-
  ```
  println(if(itRocks) "boo" else "bar")
  ```
- We have a pure expression `if(itRocks) "boo" else "bar"` wrapped in a *side-effect* `println`
- This is preferable to intertwining pure expression and side-effect as we had earlier
- This approach is what allows us to achieve high reusability and composition and automate testing!

Let's Go!

Source file

Open a text editor to a file called `Testing.hs` and enter

```
import Test.QuickCheck
import Data.Char
import Data.List

ourConfig = defaultConfig {
  -- we'll override the default of 100, for kicks
  configMaxTest = 500
}
```

Load the file

At `ghci` load the file:

```
Prelude> :load ./Testing.hs
[1 of 1] Compiling Main             ( Testing.hs, interpreted )
Ok, modules loaded: Main.
```

Is `True` true?

```
Main> check ourConfig True
OK, passed 500 tests.
```

Our First Property

Multiplicative Right Identity

- Suppose we write a function called `*` which multiplies two numbers
- For all possible values of x, then x * 1 == x

Add to our source file

```
multiplicativeIdentity x = x * 1 == x
```

Then reload and run 500 tests

```
Main> :reload
Ok, modules loaded: Main.
*Main> check ourConfig multiplicativeIdentity
OK, passed 500 tests.
```

Let's do it again

- For all possible values of a and b, then a + b == b + a

Add to our source file

```
additionCommutes a b = a + b == b + a
```

Then reload and run 500 tests

```
Main> :reload
Ok, modules loaded: Main.
*Main> check ourConfig additionCommutes
OK, passed 500 tests.
```

And a bit more

Eschew Repetition!

- We are repeating `check ourConfig`

```
ourCheck :: (Testable a) => a -> IO ()
ourCheck = check ourConfig
```

Reload and run tests using `ourCheck`

DeMorgan's Laws

```
demorgan1 p q = not (p && q) == ((not p) || (not q))
demorgan2 p q = not (p || q) == ((not p) && (not q))
```

```
Main> ourCheck `mapM_` [demorgan1, demorgan2]
OK, passed 500 tests.
OK, passed 500 tests.
```

Let's Get It Wrong

Subtraction Commutes

```
subtractionCommutes a b = a - b == b - a
```

Does it?

```
Main> ourCheck subtractionCommutes
Falsifiable, after 1 tests:
1
-2
```

Subtraction does not commute and we are given a counter-example (a=1, b=-2)

Take n

take

- `take` is a function that accepts an integer value and a list and returns a list
- `take :: Int -> [a] -> [a]`
- Experiment with the `take` function

Take One

See a problem?

```
testTake :: String -> Bool
testTake s = length (take 5 s) == 5
```

```
Main> ourCheck testTake
Falsifiable, after 0 tests:
""
```

Take Two

```
testTake' :: String -> Bool
testTake' s = length (take 5 s) <= 5
```

Better!

```
Main> ourCheck testTake'
OK, passed 500 tests.
```

Implication

==>

- Sometimes there are values we do not wish to use in the test
- If there are too many of those values, we may be best to use a different type
- However, if there are only a few of those values, we can use the ==> function

See a problem?

```
divisionInverse a b = (a `div` b) * b + (a `mod` b) == a
```

```
Main> ourCheck divisionInverse
*** Exception: divide by zero
```

Fixed!

```
divisionInverse' a b = b /= 0 ==> (a `div` b) * b + (a `mod` b) == a
```

```
Main> ourCheck divisionInverse'
OK, passed 500 tests.
```

Test Driven Development for clever people

TDD

Suppose

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

Now make the following property pass

```
property1 = f "" == ""
```

Now this one

Make this one pass as well

```
property2 x = f [x] == [x]
```

And finally

Make this one pass

```
property3 x y = f (x ++ y) == f y ++ f x
```

Correctness guarantees

- If your tests always pass, then your function should be reversing the given list *guaranteed*
- All your functions are *extensionally equivalent*
- Unfortunately we cannot get exhaustive testing for the general program (equivalent to solving Turing Halting Problem)

Non-total

- Haskell is practical for general purpose programming — non-total like most languages you have probably encountered
- Observe that we have *tests as documentation* — there is nothing more we can write! These are the unambiguous and only essential elements of the `reverse` function
  - `f :: [a] -> [a]`
  - `f [] = []`
  - `forall x. f [x] = [x]`
  - `forall x y. f (x ++ y) = f y ++ f x`

Now try this one

```
g :: (a -> b) -> (b -> c) -> (a -> c)
g = undefined
```

Only one person lives here

- If your implementation guarantees termination and does not throw an exception, then you all have the same function
- It is **impossible** to write tests for this function since they would be redundant, due to the static guarantees given by the type system!
- The given type signature is called *once-inhabited* since there is only one inhabitant, its only implementation
- Aim for static guarantees first, concede to automated testing second — clumsy manual testing under (very) exceptional circumstances, weak typing under even more exceptional circumstances

Mocking

Mocking for clever programmers

- Automated generation of “interface” implementations
- Mocking is essentially quantifying over functions e.g. for any function f then p holds

Let's try it

- `map` is a function that applies a function to each element of a list to produce a new list
-

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

- For any two functions (f and g) then map with a function that applies g then f to each element is equivalent to map each element with g then map each element of the result with f

Really?

-
```
testMap :: (Bool -> String) -> (Int -> Bool) -> [Int] -> Bool
testMap f g x = map (f . g) x == map f (map g x)
```
- [1]

```
Main> ourCheck testMap
OK, passed 500 tests.
```

Custom Data Type

Quantifying over our own type

Suppose we have our own data type such as

```
data Person = P {
  age :: Int,
  firstName :: String,
  surname :: String,
  gender :: Char
} deriving Show
```

A function accepting Person

Let's sum the ages of the people

```
sumAges :: [Person] -> Int
sumAges = sum . map age
```

Write a property

For any list of `Person` (ps) subtracting their ages from (`sumAges ps`) equals zero

```
testSumAges ps = foldl' (-) (sumAges ps) (age `map` ps) == 0
```

Oh?

```
Main> ourCheck testSumAges
  No instance for (Arbitrary Person)
```

Arbitrary Person

The type checker is asking for an instance of the `Arbitrary` type-class for `Person`

Introducing some very interesting functions

```
(>>=) :: Gen a -> (a -> Gen b) -> Gen b
return :: t -> Gen t
```

Using an existing Arbitrary

We can use the `Arbitrary` implementation for `Bool` to create an `Arbitrary` for a gender of 'm' or 'f'

```
arbGender :: Gen Char
arbGender = arbitrary >>= \b -> return (if b then 'f' else 'm')
```

In fact

We can use the `Arbitrary` implementation for the components of `Person` to create the `Arbitrary` for `Person`

```
instance Arbitrary Person where
  arbitrary = arbitrary >>= \a ->
              arbitrary >>= \f ->
              arbitrary >>= \s ->
              arbGender >>= \g ->
              return (P a f s g)
  coarbitrary = undefined -- todo another day
```

In double fact

There is special language syntax built-in for this programming pattern[2]

```
instance Arbitrary Person where
  arbitrary = do a <- arbitrary
                 f <- arbitrary
                 s <- arbitrary
                 g <- arbGender
                 return (P a f s g)
  coarbitrary = undefined
```

What Pattern?

Name?

- Does this pattern have a name?
- Have I seen it before?
- I told a lie `(>>=) :: m a -> (a -> m b) -> m b`

Java?

- Java has two specific instances of this pattern[3] built right in
  - `;` keyword (semi-colon) `Effect a -> (a -> Effect b) -> Effect b`
  - `throws` keyword `(Either Throwable) a -> (a -> (Either Throwable) b) -> (Either Throwable) b`

[2]Note that `do` and `<-` are keywords

But that's for another day :)

A. Testing.hs

```
import Test.QuickCheck
import Data.Char
import Data.List

ourConfig = defaultConfig {
  -- we'll override the default of 100, for kicks
  configMaxTest = 500
}

-- forall x. x * 1 == x
multiplicativeIdentity x = x * 1 == x

-- forall a b. a + b == b + a
additionCommutes a b = a + b == b + a

ourCheck :: (Testable a) => a -> IO ()
ourCheck = check ourConfig

-- DeMorgan's Law (1)
demorgan1 p q = not (p && q) == ((not p) || (not q))

-- DeMorgan's Law (2)
demorgan2 p q = not (p || q) == ((not p) && (not q))

-- fails (subtraction is not commutative)
subtractionCommutes a b = a - b == b - a

-- fails
testTake :: String -> Bool
testTake s = length (take 5 s) == 5
-- fixed
testTake' :: String -> Bool
testTake' s = length (take 5 s) <= 5

-- division by zero
divisionInverse a b = (a `div` b) * b + (a `mod` b) == a
-- fixed
```

```
divisionInverse' a b = b /= 0 ==> (a `div` b) * b + (a `mod` b) == a

-- TDD
f :: [a] -> [a]
f = undefined

-- this is a tad perverse
property1 :: Bool
property1 = f "" == ""

property2 :: Int -> Bool
property2 x = f [x] == [x]

property3 :: [Int] -> [Int] -> Bool
property3 x y = f (x ++ y) == f y ++ f x

-- once-inhabited
g :: (a -> b) -> (b -> c) -> (a -> c)
g = undefined

instance Show (a -> b) where
  show _ = "<<function>>"

-- mocking (generating functions)
testMap :: (Bool -> String) -> (Int -> Bool) -> [Int] -> Bool
testMap f g x = map (f . g) x == map f (map g x)

-- create your own Arbitrary

data Person = P {
  age :: Int,
  firstName :: String,
  surname :: String,
  gender :: Char
} deriving Show

sumAges :: [Person] -> Int
sumAges = sum . map age

-- for any list of Person (ps) subtracting their ages from (sumAges ps) equals
testSumAges ps = foldl' (-) (sumAges ps) (age `map` ps) == 0

instance Arbitrary Char where
    arbitrary = choose ('\32', '\128')
    coarbitrary c = variant (ord c `rem` 4)

-- (>>=) :: Gen a -> (a -> Gen b) -> Gen b
-- return :: a -> Gen a

arbGender :: Gen Char
arbGender = arbitrary >>= \b -> return (if b then 'f' else 'm')

instance Arbitrary Person where
  arbitrary = arbitrary >>= \a ->
              arbitrary >>= \f ->
              arbitrary >>= \s ->
              arbGender >>= \g ->
              return (P a f s g)
```

```
{-
  arbitrary = do a <- arbitrary
                 f <- arbitrary
                 s <- arbitrary
                 g <- arbGender
                 return (P a f s g)
-}

  -- todo We won't have time to discuss this function.
  -- It is used for generating functions (where Person appears in the argument
  coarbitrary = undefined
```