红尘里的Haskell（之三）—— 基于属性的单元测试

Felis sap...

函数式编程、编程语言、编程 话题的优秀回答者

已关注

本期起就是专题系列了。所有代码基于ghc 8.0.1和最新的Stackage Nightly snapshot。本期讲讲基于属性的单元测试。

前言

我们常希望写出的代码满足一定属性：这个属性应该可以用代码描述，而非用自然语言写到注释或文档里。我们可以人工构造一系列的输入数据及期望输出，写成单元测试的测例，甚至将测例的实现放在功能的实现之前，所谓TDD（Test-Driven Development）是也。当我们对代码的可靠性有极高要求，普通的单元测试不能胜任，则需要借助形式化方法编写可信的代码。

与代码的属性打交道，我们关心两个维度：其一是，这个属性的复杂程度，是否涉及副作用？是否可规约到某个形式系统的记法吗？其二是，想对这个属性做什么？100%可靠的形式验证？抑或单元测试即可？投入和回报需要仔细权衡。

接下来介绍的基于属性的单元测试，可以说是普通单元测试与形式化方法的折衷。

生成随机测例

首先，一点准备工作：

```
{-# LANGUAGE DataKinds #-}
{-# LANGUAGE FlexibleContexts #-}
{-# LANGUAGE FlexibleInstances #-}
{-# LANGUAGE GADTs #-}
{-# LANGUAGE GeneralizedNewtypeDeriving #-}
```

```
{-# LANGUAGE TypeFamilies #-}
{-# LANGUAGE TypeOperators #-}
{-# LANGUAGE UndecidableInstances #-}
{-# LANGUAGE UndecidableSuperClasses #-}

import Control.Monad
import Control.Monad.Trans.State.Strict
import Data.Kind
import Data.Maybe
import System.Random
```

我们的第一个任务：对指定类型生成随机测例。这里我们使用ghc boot libraries中的random库做随机数生成，transformers库做state transformation。定义一个随机值生成器：

```
newtype Gen a = Gen { unGen :: State StdGen a }
    deriving (Functor, Applicative, Monad)
```

现在，Gen a代表一个能生成a类型随机值的生成器。这个生成器基于random库的StdGen类型（immutable的随机数种子），可以输入一个StdGen，返回a类型的值以及更新的StdGen。对于一些“简单”（Random类型类成员）的类型如Bool、Int、Double等，我们已经可以用Gen来生成随机值了：

```
runGen :: Gen a -> StdGen -> (a, StdGen)
runGen (Gen g) = runState g

simpleGen :: Random a => Gen a
simpleGen = Gen (state random)
```

Gen类型最重要的特性是可组合（composable）。定义Gen时，我们用GeneralizedNewtypeDeriving扩展，让Gen类型“继承”了state monad的monad实例，现在可以用monad的return和bind操作，将多个Gen组合起来，顺序生成多个随机值，然后将其合并为目标值：

```
data BinTree a = Nil | Node a (BinTree a) (BinTree a)
    deriving (Eq, Show)

genBinTree :: Gen a -> Gen (BinTree a)
genBinTree g = do
    flag <- simpleGen
    if flag then return Nil else do
        val <- g
        left <- genBinTree g
```

以上是一个为自定义数据类型实现Gen的例子。对于product type，顺序生成每个成员的随机值然后组合之；对于sum type，生成一个随机的index，然后根据这个index判断要生成哪个constructor对应的值。

用函数表示属性

先处理最简单的一种属性：纯函数，不涉及副作用。

```
newtype Property a = Property (a -> Bool)
```

Property a类型表示对所有a类型的值都应成立的一个属性。比如之前定义的二叉树类型BinTree，假如我们写了一个flipTree函数用于翻转二叉树的话，那么一个应该成立的属性是：翻转两次之后的树内容不变。

```
flipTree :: BinTree a -> BinTree a
flipTree Nil = Nil
flipTree (Node val left right) = Node val (flipTree left) (flipTree right)

prop :: Eq a => Property (BinTree a)
prop = Property (\tree -> tree == flipTree (flipTree tree))
```

接下来，我们要测试Property a类型表达的属性。还记得前面实现的Gen吗？

```
test :: Property a -> Gen a -> State StdGen (Maybe a)
test (Property p) (Gen g) = do
    val <- g
    return (if p val then Nothing else Just val)

testN :: Property a -> Gen a -> Int -> StdGen -> [a]
testN p g n = evalState (fmap catMaybes (replicateM n (test p g)))
```

现在，可以用testN函数，给定一个Property a，一个用于生成随机值的Gen a，生成测例的数量，以及生成随机值所用的种子，返回找到反例的列表。我们已经实现了一个最简单的property-based testing框架！不妨试试将flipTree的实现改错，然后用testN来找找反例。

测试多参数的属性

前面的Property表示的属性仅有一个输入参数。需要测试带多个输入参数的属性时，最简单的做法是用tuple将多个输入值合为一个，然后验证属性时，不管传入参数还是报告错误值都用tuple。

使用tuple的一个很大问题是：Haskell的tuple是non-inductive的，不仅支持的最大长度有限，而且之后一系列需要inductive代码的场合非常麻烦。所以，我们首先实现一个可以将任意多个不同类型值放到一起的数据类型：HList（Heterogeneous List，异构列表）

```
data HList :: [*] -> * where
    HNil :: HList '[]
    HCons :: a -> HList as -> HList (a ': as)

type family LShow (as :: [*]) :: Constraint where
    LShow '[] = ()
    LShow (a ': as) = (Show a, LShow as)

deriving instance LShow as => Show (HList as)
```

HList类型携带的类型标签kind为[*]，即类型的列表，这个列表按顺序逐个标明了HList的元素类型。底下的LShow和deriving instance实现细节不用关心，其目的是为了让GHC自动生成HList的Show instance。

依照HList类型，我们的Property实现变更为：

```
newtype Property as = Property { unProperty :: HList as -> Bool }
```

Property现在能够表示带任意多个不同类型的输入参数的属性了。在测试属性之前，如果用到几个输入就传入几个Gen的话，API会很恶心，通常对同一个类型使用一个“默认”的Gen即可：

```
class HasGen a where
    defGen :: Gen a

instance HasGen (HList '[]) where
    defGen = return HNil

instance (HasGen a, HasGen (HList as)) => HasGen (HList (a ': as)) where
    defGen = liftM2 HCons defGen defGen
```

测试的实现跟上一节类似：

```
test :: HasGen (HList as) => Property as -> State StdGen (Maybe (HList as))
test (Property p) = do
    val <- unGen defGen
    return (if p val then Nothing else Just val)
```

```
testN p n = evalState (fmap catMaybes (replicateM n (test p)))
```

我们的property-based testing框架现在可以测试任意多个输入参数的属性了！不过且慢，HList as -> Bool类型的Property写起来很麻烦。。优雅的Property显然是咖喱味儿（Curry-style）的，a -> b -> c -> .. -> Bool比HList '[a, b, c, ..] -> Bool写起来好看多了。那么我们需要实现从前者到后者的uncurry操作。HList的inductive特性发挥作用了！

```
class ToProperty (as :: [*]) f where
    toProperty :: f -> Property as

instance ToProperty '[] Bool where
    toProperty = Property . const

instance ToProperty as f => ToProperty (a ': as) (a -> f) where
    toProperty f = Property (\(HCons v vs) -> unProperty (toProperty (f v)) vs)
```

ToProperty类型类的两个参数，分别代表HList-style Property的参数以及对应的Curry-style Property的类型。现在可以测试咖喱味儿的属性了：

```
prop' :: Int -> Double -> Bool
prop' x y = y == fromIntegral x

prop :: Property '[Int, Double]
prop = toProperty prop'
```

测试monadic的属性

刚才我们测试的属性全部使用纯函数表达，不涉及副作用。假设我们希望测试带副作用的属性，比如

```
-- Return value from satellite: nuked or not?
launchNuclearMissle :: City -> IO Bool
```

为了测试“对任意City，launchNuclearMissle一定成功”，刚才的框架需要修改，在IO monad中执行测例并获得输出；更一般地，我们需要支持Monad m => a -> b -> c -> .. -> m Bool这样的属性，在对应monad中获得输出。将上一节的代码中Property和test的部分稍加修改即可：

```
newtype Property m as = Property { unProperty :: HList as -> m Bool }

class ToProperty m (as :: [*]) f where
    toProperty :: f -> Property m as
```

```
    toProperty = Property . const . return

instance ToProperty m '[] (m Bool) where
    toProperty = Property . const

instance ToProperty m as f => ToProperty m (a ': as) (a -> f) where
    toProperty f = Property (\(HCons v vs) -> unProperty (toProperty (f v)) vs)

liftState :: Monad m => State s a -> StateT s m a
liftState st = StateT (return . runState st)

test :: (Monad m, HasGen (HList as)) => Property m as -> StateT StdGen m (Maybe (HList
test (Property p) = do
    val <- liftState (unGen defGen)
    flag <- lift (p val)
    return (if flag then Nothing else Just val)

testN :: (Monad m, HasGen (HList as)) => Property m as -> Int -> StdGen -> m [HList as
testN p n = evalStateT (fmap catMaybes (replicateM n (test p)))
```

现在，toProperty可以将形如a -> b -> c -> .. -> m Bool的monadic属性转化为Property，而test和testN可以在相对应的monad中执行单个测例并返回找到的反例。而对于原来的纯函数属性，toProperty仍然支持，最后在Identity monad中test即可。

从单元测试到形式验证

我们的测试框架使用普通的Haskell函数表示待测属性，因此可以表达的属性范围极广，其代价为对该种属性只能随机生成测例进行测试。如果对属性进行限制，可以用类似的API实现形式验证。（例子参见sbv和sbvPlugin）时间所限，这里不继续贴代码了，只讲一下思路吧：进行形式验证必定会调用ghc以外的工具：SMT solver如Z3，或者Model Checker如SPIN/NuSMV。

先考虑前者：SMT solver大多支持SMT-LIB格式的输入，这是一个S-expression语法的格式。我们的任务是将Haskell函数编译成该格式，其关键在于使用symbolic bool/integer/real等等来取代真正的Bool/Int/Double等，比如and :: SBool -> SBool -> SBool，实际上生成的是一个带and操作符的抽象语法树。最终，使用前面提到的技巧，在一个state monad中对函数中各个symbolic variable代入单独的变量名，即可获得一个完整的AST，将其pretty-print到SMT-LIB格式并启动SMT solver进行验证。感兴趣的可以看sbv的文档页，以及相应类型的实现代码。

对于后者而言，model checker基于时态逻辑（LTL/CTL等）验证与状态转换相关的属性，而用Haskell建模状态转换，自然想到的就是monad了。怎样设计一个受限制的、能够reify到外部工具格式的monad，有许多文章可以看，推荐从Simple and compositional reification of monadic

高质量的随机值生成

为简单起见，我们的Gen类型使用StdGen做种子，它的bit数很少（64位），随机值质量会有所影响。改进版可以选用mwc-random，不过随机值的生成就需要IO/ST monad的支持了，API会有不小改动。

对于自定义数据类型的随机值生成，前面所述方法非常naive，生成值没有好的统计特性——高质量的Gen实现应参考Boltzmann Samplers for the Random Generation of Combinatorial Structures。使用Generics或者Template Haskell，甚至可以自动生成Gen代码。

QuickCheck

property-based testing的最知名实现即为QuickCheck库。QuickCheck的实现与上文的代码主要差别有：

1. 不使用HList记录中间结果和报告错误，而是用Show约束，用字符串记录反例。
2. 通过CoArbitrary类型类，实现高阶函数的随机生成。时间所限，这篇没有实现高阶函数的部分。

另外还有一个smallcheck库，采用了不同的测例生成策略：穷举小范围内的所有值。

尾声

好久没写长文了。。技术错误、没有讲明白的地方或者遣词造句有问题，都欢迎大家评论区提出。还是那句老话，类型是人类的好朋友，不仅可以用类型避免错误，甚至可以用类型组织API和生成代码，希望本篇专题体现出了这一主旨。

关于后一个的专题：先预定讲基于free monad的mocking吧。另外我还在想要不要开个“魔界里的Haskell”坑，这边是Hackage带逛么，那边就是ICFP和JFP带逛？。。

关于题图：天都亮了，今天这一集蕾姆还没刷出来，太忧伤了，还是去歇了。。。待会求别剧透！

编辑于 2016-08-22

Haskell 函数式编程 软件测试

文章被以下专栏收录

魔鬼中的天使

主要会讨论关于函数式编程（haskell、scala）的内容。我会尽力讲的清晰明了，带你...

关注专栏

推荐阅读

工尘里的Haskell（之一）——Haskell工具链科普

elis... 发表于魔鬼中的天...

红尘里的Haskell（之二）——老司机带逛Hackage

Felis... 发表于魔鬼中的天...

Haskell开发环境配置

寸光寸阴 发表于Haske...

剖析

祖与

15 条评论

切换为时间排序

写下你的评论...

潘恒 2 年前

被封面勾引进来，却发现什么都看不懂

2

圆角骑士魔理沙 2 年前

滴，学生卡

1

雷姆好美

赞

李约瀚 2 年前

封面必须好评

赞

1ink4ever 2 年前

推迟更新了

赞

圆角骑士魔理沙 2 年前

缩短错误呢?

1

Felis sapiens (作者) 回复 圆角骑士魔理沙 2 年前

哦对。。待会起床再提一下。。

赞 查看对话

圆角骑士魔理沙 回复 Felis sapiens (作者) 2 年前

baka= =你好笨啊，蠢萌蠢萌的

赞 查看对话

祖与占 2 年前

画风变得太快

1

大魔头-诺铁 2 年前

其实我觉得property based test应该翻译为基于性质的测试，词典上可以查到property有性质的意思。

2

子非鱼 2 年前

标题党差评⊙ε⊙

1

bombless 2 年前

祖与占 2 年前

那个既然提了SMT Solver 那可以提下Liquid Haskell啊. 还有Brent Yorgey写过篇 Boltzmann sampling for generic Arbitrary instances. 期待JFP带逛!

2

neo lin 2 年前

滴，老人卡

赞

Nemo 1 年前

很早就关注了，一直没看。今天全部看完了，作者还会继续写吗?