QuickCheck

- Quick check is a Haskell library for doing random testing.
- You can read more about quickcheck at – http://www.math.chalmers.se/~rjmh/QuickCheck/

Overview

- Programmers define properties they would like to be true about their programs
- Programmers use default or user defined random test data generators
- The QuickCheck framework tests the properties against a large number of random inputs
- It reports the success and failure
- QuickCheck is in some sense orthogonal to the kind of testing done in HUnit.

Simple case

```
prop_bad xs = (reverse xs) == xs
    where types = xs::[Int]
```

Main> quickCheck prop_bad
Falsifiable, after 7 tests:
[0,4,-6,-2]
A useful case

```
prop_RevId xs = reverse (reverse xs) == xs
  where types = xs::[Int]
```

Main> quickCheck prop_RevId
OK, passed 100 tests.

Quantified tests

```
<condition> ==> <property>

prop_RevLong xs =
  length xs > 2 ==> reverse (reverse xs) == xs
  where types = xs::[Int]
```

Main> quickCheck prop_RevLong
OK, passed 100 tests.

But

```
prop_RevLong xs =
  length xs > 4 ==> reverse (reverse xs) == xs
  where types = xs::[Int]
```

Main> quickCheck prop_RevLong
Arguments exhausted after 0 tests.

Collecting data

```
collect <expression>$ <property>

prop_RevLong3 xs =
  length xs > 2 ==>
  collect (length xs) $ reverse (reverse xs) == xs
  where types = xs::[Int]
```

Main> quickCheck prop_RevLong3
OK, passed 100 tests.
Another quantified test

ordered [] = True
ordered [x] = True
ordered (x:y:zs) = (x <= y) && (ordered (y:zs))

insert x [] = [x]
insert x (y:ys) =
  if x<=y then x:y:ys else y:(insert x ys)

prop_Insert x xs =
  ordered xs ==> ordered (insert x xs)
  where types = x::Int

Main> quickCheck prop_Insert
OK, passed 100 tests.

A short side-trip Monads

• Monads & Actions
• A Monad is an Abstract Data Type (ADT)
• A Monad encapsulates effects
  – i.e. hides their implementation
• A Monad uses types to separates Actions from pure computations
• A Monad controls the order of effects

Actions

• It is sometimes useful to use actions or side effects in a functional program.
  – Update a piece of state (assignment)
  – do some IO
  – draw graphics on a display
  – return an exception rather than an answer
  – generate random test cases

• How do we do this?
  – We use a Monad
  – A Monad is a type constructor which has an implied action.
  – For example: IO Int is an action which performs some IO and then returns an Int

A monad is an ADT

• A monad is an abstract datatype
  – It abstracts computations with effects.
  – It hides the details of its implementation.
  – It exports an abstract interface.
  – It specifies the order in which effects occur.
  – It separates pure computations from effectfull ones using the type system.
  – It can have multiple implementations.
Monads Encapsulate Effects

• A monad captures the meaning and use of computations which have effects on the real world.
• A monad describes or specifies the effects in a “purely functional” way.
• A monad allows one to reason about effects, using the simple rule of substitution of equals for equals.
• A monad needn’t be implemented this way

A Monad uses types to separate Actions (with effects) from pure computations

• A Monad is a type constructor which has an implied action.
• For example:
  – a computation with type (IO Int)
  – an action which might perform some IO and which then returns an Int
• Computations with non-monadic types cannot have any effects. They are pure computations. The user (and the compiler) can rely on this.

A monad orders effects

• A Monad specifies which effects come before others.
• The Do operator provides this control over the order in which computations occur

do { var <- location x -- the first action
     ; write var (b+1) -- the next action
}

Typing the Do

If a variable is bound, it has type $a$, and its scope is to the end of the do

Each action must have a monadic type: $(M a)$

The very last action cannot have a binding
A monad’s interface hides details

• A monad has an interface that hides the details of its implementation

• Every monad has at least the two operations
  – Do          Lets users control the order of actions
  – Return : a -> Action a    makes an empty action

• Each monad has its own additional operations.

An Example: IO actions

• Each function performs an action of doing some IO
  – ? :t getChar     -- get 1 char from stdin
  – getChar :: IO Char

  – ? :t putChar     -- put Char to stdout
  – putChar :: Char -> IO()

  – ? :t getLine     -- get a whole line from stdin
  – getLine :: IO String

  – ? :t readFile    -- read a file into a String
  – readFile :: FilePath -> IO String

  – ? :t writeFile   -- write a String to a file
  – writeFile :: FilePath -> String -> IO ()

Io operations

getCharX :: Io Char     -- get 1 char from stdin
getCharX = Io(\ (c:cs) -> (c,cs,**))

putCharX :: Char -> Io()   -- put Char to stdout
putCharX c = Io(\ s -> (((),s,[c])))

getLineX :: Io String    -- get a whole line from stdin

getLineX = do { c <- getCharX
  ; if c == '"n'
    then return ""
    else do { cs <- getLineX
              ; return (c:cs)
    } }
Observations

- Actions are first class.
  - They can be abstracted (param’s of functions)
  - Stored in data structures. -- It is possible to have a list of actions, etc.
- Actions can be composed.
  - They can be built out of smaller actions by gluing them together with do and return
  - They are sequenced with do much like one uses semi-colon in languages like Pascal and C.
- Actions can be performed (run).
  - separation of construction from performance is key to their versatility.
- Actions of type: Action () are like statements in imperative languages.
  - They are used only for their side effects.

Back to Defining Generators

choose :: Random a => (a,a) -> Gen a
oneof :: [Gen a] -> Gen a

list:: Gen [Int]
list = do { elem <- choose (1,100)
            ; tail <- list
            ; oneof [ return [],
                    return(elem:tail) ]
            }

Note: Gen is a Monad

- Gen is a Monad

- A program with type (Gen a) produces an object of type ‘a’ while performing a possible ‘Gen’ action, generating random bits of data.

Generating Ordered lists

orderedList:: Int -> Gen [Int]
orderedList n =
do { elem <- choose (1,n)
    ; tail <- orderedList elem
    ; oneof [ return [],
             return(elem:tail) ]
    }
Controlling choice

frequency :: [(Int,Gen a)] -> Gen a

list2 :: Gen [Int]
list2 = do { elem <- choose (1,100)
            ; tail <- list
            ; frequency
              [ (1,return [])
              , (4,return(elem:tail)) ]
            }

using Monads better

lift :: Monad m => (b -> c) -> m b -> m c
lift f x = do { a <- x; return(f a)}

lift2 :: Monad m =>
    (b -> c -> d) -> m b -> m c -> m d
lift2 f x y =
    do { a <- x; b <- y; return(f a b)}

Another list generator

list3 :: Gen [Int]
list3 =
    frequency
    [ (1,return [])
    , (4,lift2 (:) (choose (1,100)) list3)
    ]

Generating Random RE

data RE
    = Empty
    | Union RE RE
    | Concat RE RE
    | Star RE
    | C Char
Using generators

```haskell
forall <generator>
  $ \langle\text{pattern}\rangle \rightarrow \langle\text{property}\rangle

prop_re =
  forall re $\n  \re \rightarrow \text{all (reAsPred re)}
  (reAsSet re)
```

Controlling the test generation

```haskell
re2 :: Int -> Int -> Gen RE
re2 star 0 = frequency [(1, return Empty),
  (3, lift C (choose ('a', 'z')))]
re2 star d =
  frequency [(3, return Empty),
  (8, lift C (choose ('a', 'z')))]
  (4, lift2 Union (re2 star (d-1)) (re2 star (d-1)))
  (4, lift2 Concat (re2 star (d-1)) (re2 star (d-1)))
  (star, lift Star (re2 0 (d-1)))
```

The sized function

- Test generation is controlled by the library.
- Test sizes start out small, and increase in size as the number of tests increase.
- Users can control this with the function sized.

```haskell
sized :: (Int -> Gen a) -> Gen a
```
Using sized

\[
\begin{align*}
\text{re3} :: \text{Int} \to \text{Int} \to \text{Gen RE} \\
\text{re3 star 0} &= \text{frequency} \begin{cases} 
1, \text{return Empty} \\
3, \text{lift C (choose ('a','z'))}
\end{cases} \\
\text{re3 star d} &= \\
&= \text{frequency} \\
&= \begin{cases} 
3, \text{return Empty} \\
8, \text{lift C (choose ('a','z'))} \\
4, \text{lift2 Union (re3 star (d \div 2)) (re3 star (d \div 2))} \\
4, \text{lift2 Concat (re3 star (d \div 2)) (re3 star (d \div 2))} \\
\text{star}, \text{lift Star (re3 0 (d \div 2))}
\end{cases}
\end{align*}
\]

prop_re3 =
\[
\text{forAll (sized (re3 1))}$
\]
\[
\text{\begin{align*}
\backslash r & \to \text{all (reAsPred r)} \\
& \text{ (reAsSet r)}
\end{align*}}
\]

Installing default generators

class Arbitrary a where
    arbitrary :: Gen a
    coarbitrary :: a -> Gen b -> Gen b

instance Arbitrary RE where
    arbitrary = sized (re3 1)