QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs

Koen Claessen
Chalmers University of Technology
koen@cs.chalmers.se

John Hughes
Chalmers University of Technology
rjmh@cs.chalmers.se

ABSTRACT

QuickCheck is a tool which aids the Haskell programmer in formulating and testing properties of programs. Properties are described as Haskell functions, and can be automatically tested on random input, but it is also possible to define custom test data generators. We present a number of case studies, in which the tool was successfully used, and also point out some pitfalls to avoid. Random testing is especially suitable for functional programs because properties can be stated at a fine grain. When a function is built from separately tested components, then random testing suffices to obtain good coverage of the definition under test.

1. INTRODUCTION

Testing is by far the most commonly used approach to ensuring software quality. It is also very labour intensive, accounting for up to 50% of the cost of software development. Despite anecdotal evidence that functional programs require somewhat less testing ('Once it type-checks, it usually works'), in practice it is still a major part of functional program development.

The cost of testing motivates efforts to automate it, wholly or partly. Automatic testing tools enable the programmer to complete testing in a shorter time, or to test more thoroughly in the available time, and they make it easy to repeat tests after each modification to a program. In this paper we describe a tool, QuickCheck, which we have developed for testing Haskell programs.

Functional programs are well suited to automatic testing. It is generally accepted that pure functions are much easier to test than side-effecting ones, because one need not be concerned with a state before and after execution. In an imperative language, even if whole programs are often pure functions from input to output, the procedures from which they are built are usually not. Thus relatively large units must be tested at a time. In a functional language, pure functions abound (in Haskell, only computations in the IO monad are hard to test), and so testing can be done at a fine grain.

A testing tool must be able to determine whether a test is passed or failed; the human tester must supply an automatically checkable criterion of doing so. We have chosen to use formal specifications for this purpose. We have designed a simple domain-specific language of *testable specifications* which the tester uses to define expected properties of the functions under test. QuickCheck then checks that the properties hold in a large number of cases. The specification language is embedded in Haskell using the class system. Properties are normally written in the same module as the functions they test, where they serve also as checkable documentation of the behaviour of the code.

A testing tool must also be able to generate test cases automatically. We have chosen the simplest method, random testing [11], which competes surprisingly favourably with systematic methods in practice. However, it is meaningless to talk about random testing without discussing the distribution of test data. Random testing is most effective when the distribution of test data follows that of actual data, but when testing reuseable code units as opposed to whole systems this is not possible, since the distribution of actual data in all subsequent reuses is not known. A uniform distribution is often used instead, but for data drawn from infinite sets this is not even meaningful – how would one choose a random closed λ-term with a uniform distribution, for example? We have chosen to put distribution under the human tester's control, by defining a *test data generation language* (also embedded in Haskell), and a way to observe the distribution of test cases. By programming a suitable generator, the tester can not only control the distribution of test cases, but also ensure that they satisfy arbitrarily complex invariants.

An important design goal was that QuickCheck should be *lightweight*. Our implementation consists of a single pure Haskell'98 module of about 300 lines, which is in practice mainly used from the Hugs interpreter. We have also written a small script to invoke it, which needs to know very little about Haskell syntax, and consequently supports the full language and its extensions. It is not dependent on any particular Haskell system. A cost that comes with this decision is that we can only test properties that are expressible and observable *within* Haskell.

It is notoriously difficult to say how effective a testing method is in detecting faults. However, we have used QuickCheck in a variety of applications, ranging from small exper-

ICFP '00, Montreal, Canada.

iments to real systems, and we have found it to work well in practice. We report on some of this experience in this paper, and point out pitfalls to avoid.

The rest of this paper is structured as follows. Section 2 introduces the concept of writing properties and checking them using QuickCheck. Section 3 shows how to define test data generators for user-defined types. Section 4 briefly discusses the implementation. Section 5 presents a number of case studies that show the usefulness of the tool. Section 6 concludes.

2. DEFINING PROPERTIES

2.1 A Simple Example

As a first example, we take the standard function `reverse` which reverses a list. This satisfies a number of useful laws, such as

```
          reverse [x]  =  [x]
      reverse (xs++ys)  =  reverse ys++reverse xs
 reverse (reverse xs)  =  xs
```

In fact, the first two of these characterise `reverse` uniquely.

Note that these laws hold only for *finite, total* values. In this paper, unless specifically stated otherwise, we always quantify over completely defined finite values. This is to make it more likely that the properties are computable.

In order to check these laws using QuickCheck, we represent them as Haskell functions. Thus we define

```
prop_RevUnit x =
  reverse [x] == [x]

prop_RevApp xs ys =
  reverse (xs++ys) == reverse ys++reverse xs

prop_RevRev xs =
  reverse (reverse xs) == xs
```

Now, if these functions return `True` for every possible argument, then the properties hold. We load them into the Hugs interactive Haskell interpreter [14], and call for example

```
Main> quickCheck prop_RevApp
OK: passed 100 tests.
```

The function `quickCheck` takes a law as a parameter and applies it to a large number of randomly generated arguments — in fact 100[1] — reporting "OK" if the result is `True` in every case.

If the law fails, then `quickCheck` reports the counter-example. For example, if we mistakenly define

```
prop_RevApp xs ys =
  reverse (xs++ys) == reverse xs++reverse ys
```

then checking the law might produce

```
Main> quickCheck prop_RevApp
Falsifiable, after 1 tests:
[2]
[-2,1]
```

where the counter model can be extracted by taking `[2]` for `xs`, and `[-2,1]` for `ys`.

[1] 100 is a rather arbitrary number, so our library provides a way to specify this as a parameter.

In fact the programmer must provide a little more information: the function `quickCheck` is actually overloaded, in order to be able handle laws with a varying number of variables, and the overloading cannot be resolved if the law itself has a polymorphic type, as in these examples. Thus the programmer must specify a fixed type at which the law is to be tested. So we simply give a type signature for each law, for example

```
prop_RevApp :: [Int] -> [Int] -> Bool
```

Of course, the property `prop_RevApp` holds polymorphically, but we must specify which monomorphic instance to test it at, so that we can generate test cases. This turns out to be quite important in the case of overloaded functions. For example, $+$ is associative for the type `Int`, but not for `Double`! In some cases, we can use parametricity [17] to argue that a property holds polymorphically.

2.2 Functions

We are also able to formulate properties that quantify over functions. To check for example that function composition is associative, we first define extensional equality `(===)` by `(f === g) x = f x == g x`, and then write:

```
prop_CompAssoc :: (Int -> Int) -> (Int -> Int)
               -> (Int -> Int) -> Int -> Bool
prop_CompAssoc f g h =
  f . (g . h) === (f . g) . h
```

The only difficulty that function types cause is that, if a counter-example is found (for example if we try to check that function composition is commutative), then the function values are printed just as "`<<function>>`". In this case we discover that the 'law' we are checking is false, but not why.

2.3 Conditional Laws

Laws which are simple equations are conveniently represented by boolean function as we have seen, but in general many laws hold only under certain conditions. QuickCheck provides an implication combinator to represent such conditional laws. For example, the law

$$\texttt{x <= y} \Longrightarrow \texttt{max x y == y}$$

can be represented by the definition

```
prop_MaxLe :: Int -> Int -> Property
prop_MaxLe x y = x <= y ==> max x y == y
```

Likewise, the insertion function into ordered lists satisfies the law

```
prop_Insert :: Int -> [Int] -> Property
prop_Insert x xs =
  ordered xs ==> ordered (insert x xs)
```

Note that the result type of the property is changed from `Bool` to `Property`. This is because the testing semantics is different for conditional laws. Instead of checking the property for 100 random test cases, we try checking it for 100 test cases *satisfying the condition.* If a candidate test case does not satify the condition, it is discarded, and a new test case is tried.

Checking the laws `prop_MaxLe` and `prop_Insert` succeed as usual, but sometimes, checking a conditional law produces the output

```
Arguments exhausted after 64 tests.
```

If the precondition of a law is seldom satisfied, then we might generate many test cases without finding any where it holds. In such cases it is hopeless to search for 100 cases in which the precondition holds. Rather than allow test case generation to run forever, we generate only a limited number of candidate test cases (the default is 1000). If we do not find 100 valid test cases among those candidates, then we simply report the number of successful tests we were able to perform.

In the example, we know that the law passed the test in 64 cases. It is then up to the programmer to decide whether this is enough, or whether it should be tested more thoroughly.

2.4 Monitoring Test Data

Perhaps it seems that we have tested the law for insert thoroughly enough to establish its credibility. However, we must be careful. Let us modify `prop_Insert` as follows[2]

```
prop_Insert :: Int -> [Int] -> Property
prop_Insert x xs =
  ordered xs ==>
    classify (null xs) "trivial" $
      ordered (insert x xs)
```

Checking the law now produces the message

```
OK, passed 100 tests (43% trivial).
```

The `classify` combinator does not change the meaning of a law, but it classifies some of the test cases, in this case those where `xs` is the empty list were classified as "trivial". Thus we see that a large proportion of the test cases only tested insertion into an empty list.

We can get more information than just labelling some test cases. The combinator `collect` will gather all values that are passed to it, and print out a histogram of these values. For example, if we write:

```
prop_Insert :: Int -> [Int] -> Property
prop_Insert x xs =
  ordered xs ==>
    collect (length xs) $
      ordered (insert x xs)
```

we might get as a result:

```
OK, passed 100 tests.
49% 0.
32% 1.
12% 2.
4% 3.
2% 4.
1% 5.
```

So we see that only 19 cases tested insertion into a list with more than one element. While this is enough to provide fairly strong evidence that the law holds, it is worrying that very short lists dominate the test cases so strongly. After all, it is easy to define an erroneous version of `insert` which nevertheless works for lists with at most one element.

The reason for this behaviour, of course, is that the precondition `ordered xs` skews the distribution of test cases towards short lists. Every generated list of length 0 or 1 is ordered, but only 50% of the lists of length 2 are. Thus test cases with longer lists are more likely to be rejected by the precondition. There is a risk of this kind of problem every time we use conditional laws, so it is always important to investigate the proportion of trivial cases among those actually tested.

The best solution, though, is to replace the condition with a custom test data generator for ordered lists. We write

```
prop_Insert :: Int -> Property
prop_Insert x =
  forAll orderedList $ \xs ->
    ordered (insert x xs)
```

which specifies that values for `xs` should be generated by the test data generator `orderedList`. Checking the law now gives "`OK: passed 100 tests`", as we would expect.

QuickCheck provides support for the programmer to define his or her own test data generators, with control over the distribution of test data, which we will look at in the next section.

2.5 Infinite Structures

The Haskell function `cycle` takes a non-empty list, and returns a list that repeats the contents of that list infinitely. Now, take a look at the following law, formulated in QuickCheck as[3]:

```
prop_DoubleCycle :: [Int] -> Property
prop_DoubleCycle xs =
  not (null xs) ==>
    cycle xs == cycle (xs ++ xs)
```

Although intuitively the law is true, it cannot be checked since we are comparing two infinite lists using computable equality `==`, which does not terminate. Instead, we can reformulate the property as a logically equivalent one, by using the fact that two infinite lists are equal if all finite initial segments are equal.

```
prop_DoubleCycle :: [Int] -> Int -> Property
prop_DoubleCycle xs n =
  not (null xs) && n >= 0 ==>
    take n (cycle xs) == take n (cycle (xs ++ xs))
```

Another issue related to infinite structures is quantification over them. We will later see how to deal with properties that for example hold for all infinite lists, but in general it is not clear how to formulate and execute properties about structures containing bottom.

3. DEFINING GENERATORS

3.1 Arbitrary

The way we generate random test data of course depends on the type. Therefore, we have introduced a type class `Arbitrary`, of which a type is an instance if we can generate arbitrary elements in it.

```
class Arbitrary a where
  arbitrary :: Gen a
```

`Gen a` is an abstract type representing a generator for type `a`. The programmer can either use the generators built in to QuickCheck as instances of this class, or supply a custom generator using the `forAll` combinator, which we saw in the previous section. For now, we define the type `Gen` as

[2] `$` is Haskell's infix function application.

[3] Note that leaving the condition `not (null xs)` out results in an error, because `cycle` is not defined for empty lists.

```
newtype Gen a = Gen (Rand -> a)
```

Here Rand is a random number seed; a generator is just a function which can manufacture an a in a pseudo random way. But we will treat Gen as an *abstract* type, so we define a primitive generator

```
choose :: (Int, Int) -> Gen Int
```

to choose a random number in an interval, and we program other generators in terms of it.

We also need combinators to build complex generators from simpler ones; to do so, we declare Gen to be an instance of Haskell's class Monad. This involves implementing the methods of the Monad class

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

the first one of which contructs a constant generator, and the second one being the monadic sequencing operator which generates an a, and passes it to its second argument to generate a b. The definition of (>>=) needs to pass *independent* random number seeds to its two arguments, and is only passed one seed, but luckily the Haskell random number library provides an operation to split one seed into two.

Defining generators for many types is now straightforward. As examples, we give generators for integers and pairs:

```
instance Arbitrary Int where
  arbitrary = choose (-20, 20)

instance (Arbitrary a, Arbitrary b) =>
                        Arbitrary (a,b) where
  arbitrary = liftM2 (,) arbitrary arbitrary
```

In the second case we use a standard monadic function, liftM2, which is defined in terms of return and (>>=), to make a generator that applies the pairing operator (,) to the results of two other generators. QuickCheck contains such declarations for most of Haskell's predefined types.

3.2 Generators for User-Defined Types

Since we define test data generation via an instance of class Arbitrary for each type, then we must rely on the user to provide instances for user-defined types. In principle we could try to generate these automatically, in a pre-processor or via polytypic programming [2], but we have chosen instead to leave this task to the user. This is partly because we want QuickCheck to be a lightweight tool, easy to implement and easy to use in a standard programming environment; we don't want to oblige users to run their programs through a pre-processor between editing them and testing them. But another strong reason is that it seems to be very hard to construct a generator for a type, without knowing something about the desired distribution of test cases.

Instead of producing generators automatically, we provide combinators to enable a programmer to define his own generators easily. The simplest, called oneof, just makes a choice among a list of alternative generators with a uniform distribution. for example, if the type Colour is defined by

```
data Colour = Red | Blue | Green
```

then a suitable generator can be defined by

```
instance Arbitrary Colour where
  arbitrary = oneof
    [return Red, return Blue, return Green]
```

As another example, we could generate arbitrary lists using

```
instance Arbitrary a => Arbitrary [a] where
  arbitrary = oneof
    [return [], liftM2 (:) arbitrary arbitrary]
```

where we use liftM2 to apply the cons operator (:) to an arbitrary head and tail. However, this definition is not really satisfactory, since it produces lists with an average length of one element. We can adjust the average length of list produced by using frequency instead, which allows us to specify the frequency with which each alternative is chosen. We define

```
instance Arbitrary a => Arbitrary [a] where
  arbitrary = frequency
    [ (1, return [])
    , (4, liftM2 (:) arbitrary arbitrary) ]
```

to choose the cons case four times as often as the nil case, leading to an average list length of four elements.

However, for more general data types, it turns out that we need even finer control over the distribution of generated values. Suppose we define a type of binary trees, and a generator:

```
data Tree a = Leaf a | Branch (Tree a) (Tree a)

instance Arbitrary a => Arbitrary (Tree a) where
  arbitrary = frequency
    [ (1, liftM  Leaf   arbitrary)
    , (2, liftM2 Branch arbitrary arbitrary) ]
```

We want to avoid choosing a Leaf too often, hence the use of frequency.

However, *this definition only has a 50% chance of terminating!* The reason is that for the generation of a Branch to terminate, *two* recursive generations must terminate. If the first few recursions choose Branches, then generation terminates only if very many recursive generations all terminate, and the chance of this is small. Even when generation terminates, the generated test data is sometimes very large. We want to avoid this: since we perform a large number of tests, we want each test to be small and quick.

Our solution is to limit the *size* of generated test data. But the notion of a size is hard even to define in general for an arbitrary recursive datatype (which may include function types anywhere). We therefore give the responsibility for limiting sizes to the programmer defining the test data generator. We change the representation of generators to

```
newtype Gen a = Gen (Int -> Rand -> a)
```

where the new parameter is to be interpreted as some kind of size bound. We define a new combinator

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

which the programmer can use to access the size bound: sized generates an a by passing the current size bound to its parameter. It is then up to the programmer to interpret the size bound in some reasonable way during test data generation. For example, we might generate binary trees using

```
instance Arbitrary a => Arbitrary (Tree a) where
  arbitrary = sized arbTree

arbTree 0 = liftM Leaf arbitrary
```

```
arbTree n = frequency
  [ (1, liftM  Leaf   arbitrary)
  , (4, liftM2 Branch (arbTree (n `div` 2))
                      (arbtree (n `div` 2))) ]
```

With this definition, the size bound limits the number of nodes in the generated trees, which is quite reasonable.

Now that we have introduced the notion of a size bound, we can use it sensibly in the generators for other types such as integers and lists (so that the absolute value respective length is bounded by the size). So the definitions we presented earlier need to be modified accordingly.

We stress that the size bound is simply an extra, global parameter which every test data generator may access; every use of `sized` sees the same bound[4]. We do *not* attempt to 'divide the size bound among the generators', so that for example a longer generated list would have smaller elements, keeping the overall size of the structure the same. The reason is that we wish to avoid correlations between the sizes of different parts of the test data, which might distort the test results.

We do vary the size between different test cases: we begin testing each property on small test cases, and then gradually increase the size bound as testing progresses. This makes for a greater variety of test cases, which both makes testing more effective, and improves our chances of finding enough test cases satisfying the precondition of conditional properties. It also makes it more likely that we will find a small counter example to a property, if there is one.

3.3 Generating Functions

If we are to check properties involving function valued variables, then we must be able to generate arbitrary functions. Rather surprisingly, we are able to do so. To understand how, notice that a function generator of type `Gen (a->b)` is represented by a function of type `Int -> Rand -> a -> b`. By reordering parameters, this is equivalent to the type `a -> Int -> Rand -> b`, which represents `a -> Gen b`. We can thus define

```
promote :: (a -> Gen b) -> Gen (a->b)
```

and use it to produce a generator for a function type, provided we can construct a generator for the result type which somehow depends on the argument value. We take care of this dependence by defining a new class,

```
class Coarbitrary a where
  coarbitrary :: a -> Gen b -> Gen b
```

whose method `coarbitrary` modifies a generator in a way depending on its first parameter. We will think of `coarbitrary` as producing a *generator transformer* from its first argument. Given this class, we can define

```
instance (Coarbitrary a, Arbitrary b) =>
                         Arbitrary (a->b) where
  arbitrary =
    promote (\a -> coarbitrary a arbitrary)
```

which generates an arbitary function that uses its argument to modify the generation of its result.

In order to define instances of `Coarbitrary` we need a way to construct generator transformers. We therefore define the function

[4]Unless the programmer explicitly changes it using the `resize` combinator.

```
variant :: Int -> Gen a -> Gen a
```

where `variant n g` constructs a generator which transforms the random number seed it is passed in a way depending on `n`, before passing it to `g`. This function must be defined very carefully, so that all the generators we construct using it are independent. Given any list of integers `[n1,n2,...nk]`, we can construct a generator transformer

```
variant n1 . variant n2 . ... . variant nk
```

We define `variant` so that different lists of integers give rise to independent generator transformers (with a very high probability).

Now we can define instances of `coarbitrary` that choose between generator transformers depending on the argument value. For example, the boolean instance

```
instance Coarbitrary Bool where
  coarbitrary b =
    if b then variant 0 else variant 1
```

transforms a generator in independent ways for `True` and for `False`; the generators `coarbitrary True g` and `coarbitrary False g` will be independent. In a similar way, we can define suitable instances for many other types. For example, the integer instance just converts its integer argument into a sequence of bits, which are then used as generator transformers in turn.

Instances of `Coarbitrary` for recursive datatypes can be defined according to a standard pattern. For example, the list instance is just

```
instance Coarbitrary a => Coarbitrary [a] where
  coarbitrary []     = variant 0
  coarbitrary (x:xs) =
    variant 1 . coarbitrary x . coarbitrary xs
```

The goal is that different lists should be mapped to independent generator transformers; we achieve this by mapping each constructor to an independent transformer, and composing these with transformers derived from each component. Other recursive datatypes can be treated in the same way. Since the programmer is responsible for making these definitions for user-defined types, it is important that they be straightforward.

Finally, we can even interpret functions as generator transformers, with an instance of the form

```
instance (Arbitrary a, Coarbitrary b) =>
                       Coarbitrary (a->b) where
  coarbitrary f gen =
    arbitrary >>= \a -> coarbitrary (f a) gen
```

The idea is that we apply the given function to an arbitrary argument, and use the result to transform the given generator. In this way, two functions which are different will give rise to different generator transformers.

Note that, if we had tried to avoid needing to split random number seeds by defining the `Gen` monad as a state transformer on the random seed, rather than a state reader, then we would not have been able to define the `promote` function, and we would not have been able to generate random functions.

4. IMPLEMENTING QUICKCHECK

As we have seen, the function `quickCheck` can handle properties with a varying number of arguments and different result types. To implement this, we introduce the type `Property`, and we create the type class `Testable`.

```
class Testable a where
  property :: a -> Property
```

The Property type represents predicates that can be checked by testing. This means that it needs to be able to generate random input, and finally product a test result. So, a Property is a computation in the Gen monad, ending in an abstract type Result, which keeps track of the boolean result of the testing, the classifications of test data, and the arguments used in the test case.

```
newtype Property = Prop (Gen Result)
```

Let us take a look at some instances of Testable. An easy type to check is of course Bool. Further, functions for which we can generate arbitrary arguments can be tested. And lastly, even the property type itself is an instance, so that we can nest property combinators.

```
instance Testable Bool where
  property b = Prop (return (resultBool b))

instance (Arbitrary a, Show a, Testable b) =>
                          Testable (a->b) where
  property f = forAll arbitrary f

instance Testable Property where
  property p = p
```

Using the function property, it becomes easy to define the function quickCheck. Its type is:

```
quickCheck :: Testable a => a -> IO ()
```

More details of the implementation can be found in the appendix.

5. SOME CASE STUDIES

5.1 Unification

As a first (and rather pathological) case study, we discuss a unfication algorithm which we have developed along with a QuickCheck specification. This was quite revealing, both as regards the impact that QuickCheck has on programming, and the pitfalls that must be avoided. It is too large to present in detail, so we will just discuss the lessons we learned.

5.1.1 Impact on Type Definitions

First of all, the use of QuickCheck had an impact on the design of the types in the program. We defined the type of terms to be unified as

```
data Term    = Var Var | Constr Name [Term]
newtype Var  = Variable Nat
newtype Name = Name String
```

rather than the equivalent

```
data Term = Var Int | Constr String [Term]
```

which we would probably have chosen otherwise. The type we used distinguishes between a string used as a constructor name, and a string used in other contexts, and between a natural number used as a variable name, and an integer used in other contexts.

The reason we chose to make these distinctions in the type is that it enabled us to define a different test data generator for Names for example, than for strings. Had we generated terms using the default test data generator for strings, then it is very unlikely that we would ever generate unifiable terms, since it is unlikely we would generate the same constructor name twice. Instead, we chose to generate constructor names using

```
instance Arbitrary Name where
  arbitrary = sized $ \n -> oneof
    [ return (Name ("v" ++ show i))
    | i <- [1..log n+1] ]
```

which gives us a good chance that generated terms will be at least partially unifiable. Likewise, we limited unification variables in test data to a small set.

Of course, we could have used the second Term type above and specified a custom test data generator with an explicit forAll in each property. But it is more convenient to let test data be automatically generated using arbitrary, so one is encouraged to make distinctions explicit in types. There are other advantages to doing so also: it permits the type checker to detect more errors. So, using QuickCheck changes the balance of convenience in the question of introducing new types in programs.

5.1.2 Checking Functional Properties

A unifier needs to manage the current substitution, and also the possibility of failures in recursive calls. A convenient way to do so is to use a monad. We defined a unification monad M, represented by a function, with operations

```
setV :: Var -> Term -> M ()
getV :: Var -> M Term
```

to read and write variables, among others. We were able to define an 'extensional equality' operator eqM on monadic values, and check both the monad laws and properties such as

```
prop_SetGet v t = (do setV v t; getV v)
            `eqM` (do setV v t; return t)
```

5.1.3 Errors Found

It would be nice to be able to report that QuickCheck found a large number of errors in this example. In fact, no errors at all were found in the unifier itself. This is probably more a reflection on the number of times the authors have programmed unifiers previously, than on the effectiveness of QuickCheck — we know how to do it, quite simply.

On the other hand, we did find errors in the *specification*. For example, we defined a substitution function which repeatedly substitutes until no variables in the domain of the substitution remain, and stated the obvious property

```
prop_SubstIdempotent s t =
  subst s (subst s t) == subst s t
```

QuickCheck revealed this property to be false: it holds only for acyclic substitutions (otherwise an infinite term is generated, and the equality test loops). This error was found using the function verboseCheck, which prints out the arguments to every test case before it checks it.

We were obliged to correct the specification to

```
prop_SubstIdempotent s t =
  acyclic s ==> subst s (subst s t) == subst s t
```

Thus QuickCheck made us think harder about the properties of our code, and document them correctly.

On the downside, formulating the specification correctly turned out to be quite a lot of work, perhaps more than writing the implementation. This was partly because predicates such as `acyclic` are non-trivial to define; a good set theory library would have helped here.

5.1.4 A False Sense of Security

The most serious pitfall we uncovered with this experiment was the false sense of security that can be engendered when one's program passes a large number of tests in trivial cases. We have already referred to this problem when we discussed conditional properties; in this example, it bit us hard.

Many properties of unification apply to the case when unification succeeds. They can be stated conveniently in the form

```
prop_Unify t1 t2 = s /= Nothing ==> ...
   where s = unifier t1 t2
```

since our unifier returns `Nothing` when it fails. With a little reflection, we see that two randomly chosen terms are fairly likely to be unifiable, since variables occur quite often, and if either term is a variable then unification will almost certainly succeed. On the other hand, if neither term is a variable then the probability that they will unify is small. Thus the case where one term is a variable is heavily overrepresented among the test cases that satisfy the precondition — we found that over 95% of test cases had this property. Although QuickCheck succeeded in 'verifying' every property, we can hardly consider that they were thoroughly tested.

The solution was to use a custom test data generator

```
prop_Unify =
  forAll probablyUnifiablePair $ \(t1,t2) ->
    s /= Nothing ==> ...
 where s = unifier t1 t2
```

We generated 'probably unifiable pairs' by generating *one* random term, and replacing random subterms by variables in two different ways. This usually generates unifiable terms, although may fail to when variables are used inconsistently in the two terms. With this modification, the proportion of trivial cases fell to a reasonable 20–25%.

This experience underlines the importance of investigating the distribution of actual test cases, whenever conditional properties are used.

5.2 Circuit Properties

5.2.1 Lava in a Nutshell

Lava [3, 7] is a tool to describe, simulate and formally verify hardware. Lava is a so-called *embedded language*, which means that the circuit descriptions and properties are all expressed in an existing programming language, in this case Haskell.

The idea is to view a hardware circuit as a function from signals of inputs to signals of outputs. The Lava system provides primitive circuits, such as `and2`, `xor2`, and `delay`. More complicated circuits are defined by combining these. Circuits defined in Lava can be *simulated* as follows: one provides inputs and the outputs are calculated.

```
Main> simulate and2 (high, low)
high
```

Furthermore, the Lava system defines combinators for circuits, such as sequential composition (`>->`), parallel composition (`<|>`), and `column`, which takes one circuit and replicates it in a column of circuits, connecting the vertical wires.

5.2.2 Properties in Lava

Properties of circuits can be defined in a similar way. For example, to define the property that a certain circuit is commutative, we say:

```
prop_Commutative circ (a, b) =
  circ (a, b) <==> circ (b, a)
```

where `<==>` is logical equivalence lifted to arbitrary types containing signals, in this case a pair.

Properties can be formally verified. We do this by providing *symbolic* inputs to the circuit or property, and calculating a concrete expression in a Haskell datatype, representing the circuit.

We can then write this expression to a file and call an external theorem prover. All this is done by the overloaded Lava function `verify`. Here is how we can use it to verify that a so-called half adder component is commutative:

```
Main> verify (prop_Commutative halfAdd)
Proving: ... Valid.
```

The Lava system provides a number of functions and combinators to conveniently express properties and formally verify them.

5.2.3 QuickCheck in Lava

Though we are able to verify properties about circuits in Lava, we greatly benefit from extending it with a testing tool like QuickCheck. There are two main reasons for that.

The first one is that calling an external theorem prover is a very heavyweight process. When verifying, say, a 32-bit multiplier, the formulas that we generate for external theorem proving are quite big and we often have to wait for a long time to get an answer.

So, a typical development cycle is to write down the specification of the circuit first, then make an implementation, QuickCheck it for obvious bugs, and lastly, call the external theorem prover for verifying the correctness.

Here is an example of how to use QuickCheck in Lava:

```
Main> quickCheck (prop_Commutative halfAdd)
OK: passed 100 tests.
```

Adding this testing methodology to Lava turned out to be quite straightforward, because Lava already had a notion of properties. Testing can be done for all circuit types, even sequential circuits (containing latches). We simply check the circuit property for a sequence of inputs.

5.2.4 Higher Order Testing

The second reason for using testing in Lava is simply that we can test *more* properties than we can formally verify! The external theorem provers that are connected to Lava can only deal with at most first-order logics, and the Lava system is only able to generate formulas of that type.

Sometimes, we would like to verify properties about *combinators*. For example, proving that `column` distributes over (`>->`):

```
prop_ComposeSeqColumn circ1 circ2 inp =
      column (circ1 >-> circ2) inp
 <==> (column circ1 >-> column circ2) inp
```

is very hard to verify in Lava *for all* `circ1` *and* `circ2`. In fact, such properties are hard to verify automatically in general (we can do it for small fixed sizes however). But since we can randomly generate functions, we can at least *test* these kind of properties for arbitrary circuits.

A drawback is that we have to fix the types of these circuits, whereas the combinators themselves, and thus the properties about them, are polymorphic in the circuits' input and output types.

5.2.5 Errors Found

The authors used the QuickCheck library while developing a collection of arithmetic circuits. Previously, testing was already used in the development process, but only in a very limited and ad-hoc way. Now, much more thorough testing was possible.

The errors we found in these particular circuits were of two kinds. Firstly, we found errors that our formal verification method would have found as well: logical errors in the circuits. But secondly, we also found errors due to the fact that random input also means random input *size*. For example, for an n-bit $\times$ m-bit adder, we only use and formally verify the circuit for specific input sizes. Random testing checks many more combinations, and it often turned out that we had forgotten to define one of these cases!

5.3 Propositional Theorem Proving

For teaching purposes, we implemented two different well-known methods of checking if a set of propositional logic clauses is contradictory. One of these methods was the Davis-Putnam method [8], which uses backtracking to generate a list of all models. The other one was Stålmarck's method [16], which is an incomplete method and uses a variant on the dilemma proof system to gather information about the literals in the clause set.

```
type Clause = [Lit]  -- disjunction
type Model  = [Lit]  -- conjunction

davisPutnam :: [Clause] -> [Model]
stålmarck   :: Int -> [Clause] -> Maybe Model
```

The `stålmarck` function takes an extra argument, an `Int`, which is the so-called "saturation level", a parameter which limits the depth of the proofs, and usually lies between 0 and 3. If the result of `stålmarck` is `Nothing`, it means that there was a contradiction. If the result is `Just m`, it means that every model of the clause set should have `m` as a sub-model.

Since `davisPutnam` is much more straight-forward to implement than `stålmarck`, we wanted to check the latter against the first. Here is how we formulate the informal property stated above:

```
prop_Stålmarck_vs_DP :: Property
prop_Stålmarck_vs_DP =
  forAll clauses $ \cs ->
    forAll (choose (0,3)) $ \sat ->
      case stålmarck sat cs of
        Nothing ->
          collect "contradiction" $
            davisPutnam cs == []

        Just m ->
          not (null m) ==>
            collect (length m) $
              all (m `subModel`) (davisPutnam cs)
```

Note that we collect some statistics information: "contradiction" when the result was `Nothing`, and the size of `m` in the case of `Just m`. We also expressed that we disqualify a test case when `stålmarck` returns `Just []`.

With the help of this property, QuickCheck found 3 bugs! These bugs were due to implicit unjustified assumptions we had about the input. The implementations of both algorithms assumed that no clauses in the input could contain the same literal twice, and the `stålmarck` function assumed that none of the input clauses was empty.

The data generator `clauses` is defined using the same techniques as in section *5.1.1*. Testing the property took about 30 seconds, and from the output we could see that the distribution of `Nothing` vs. `Just m` was about 50/50.

5.4 Pretty Printing

Andy Gill reported an interesting story about using QuickCheck to us. He used it in developing a variant of Wadler's pretty printing combinator library [18] in Java. First, he implemented his variant functionally, using Haskell. Then, still using Haskell, he used a state monad with exceptions to develop an imperative implementation of the same library. The idea was that the second implementation models what goes on in a Java implementation.

Then, he expressed the relationship between the two different implementations using QuickCheck properties. He writes: *"This quickly points out where my reasoning is faulty, and provides great tests to catch the corners of the implementation. Three problems were found, the third of which showed that I had merged two concepts in my implementation that I should not have."*

Furthermore, he made an improvement in the way QuickCheck reports counter examples. Sometimes, the counter examples found are very large, and it is difficult to go back to the property and understand why it is a counter example. However, when the counter example is an element of a tree-shaped datatype, the problem can often be located in one of the sub-trees of the counter example found. Gill extended the `Arbitrary` class with a new method

```
smaller :: a -> [a]
```

which is intended to return a list of smaller, but similar values to its argument – for example, direct subtrees. He adapted the `quickCheck` function so that when a counter example is found, it tries to find a smaller one using this function. In some cases much smaller counterexamples were found, greatly reducing the time to understand the bug found.

The last step Gill made in developing his Java pretty printing library was porting the state and exception monad model in Haskell to Java. He then used QuickCheck to generate a large number of test inputs for the Java code, in order to check that the Java implementation was equivalent to the two Haskell models.

5.5 Edison

Chris Okasaki's *Edison* is a library of efficient data structures suitable for implementation and use in functional programming languages. He has used QuickCheck to state and test properties of the library. Every data structure in the library has been made an instance of `Arbitrary`, and he has included several extra modules especially for formulating properties about these data structures. He reports: *"My experience has mostly been that of a very satisfied user.* Quick-

Check *lets me test Edison with probably 25% (maybe less!) of the effort of my previous test suite, and does a much better job to boot."*

Okasaki also mentions a drawback, having to do with the Haskell module system. He often uses one specification of a data structure together with different implementations. A natural way to do this is to place the specification in one module, and each implementation in a seperate module. But since the specification refers to the implementation, then the specification module must import the implementation one currently under test. Okasaki was obliged to edit the specification module by hand before each test, so as to import the right implementation! Much preferable would be to parameterise the specification on an implementation module; ML-style functors would be really helpful here!

6. DISCUSSION

6.1 On Random Testing

At first sight, random selection of test cases may seem a very naive approach. Systematic methods are often preferred: in general, a *test adequacy criterion* is defined, and testing proceeds by generating test cases which meet the adequacy criterion. For example, a simple criterion is that every reachable statement should be executed in at least one test, a more complex one that every feasible control-flow path (with exceptions for loops) be followed in at least one test. A wide variety of adequacy critera have been proposed; a recent survey is [19].

We have chosen not to base QuickCheck on such an adequacy criterion. In part, this is because many criteria would need reinterpretation before they could be applied to Haskell programs – it is much less clear, for example, what a control-flow path is in a language with higher-order functions and lazy evaluation. In part, such a criterion would force us to use much more heavyweight methods – even measuring path coverage, for example, would require compiler modifications and thus tie QuickCheck to a particular implementation of Haskell (namely the one we modified to collect path information). Generating test data to exercise a particular path requires constraint solving: one must find input values which make the series of tests along the given path produce specified results. While such constraint solving may be feasible for arithmetic data, for the rich symbolic datatypes found in Haskell programs it is a difficult research problem in its own right.

However, apart from the difficulty of automating systematic testing methods for Haskell, there is no clear reason to believe that doing so would yield better results. In 1984, Duran and Ntafos compared the fault detection probability of random testing with partition testing, and discovered that the differences in effectiveness were small [9]. Hamlet and Taylor repeated their study more extensively, and corroborated the original results [12]. Although partition testing is slightly more effective at exposing faults, to quote Hamlet's excellent survey [11], *"By taking 20% more points in a random test, any advantage a partition test might have had is wiped out."*

For small programs in particular, it is likely that random test cases *will* indeed exercise all paths, for example, so that test coverage is likely to be good by any measure. Using QuickCheck, we apply random testing at a fine grain: we check properties of individual functions, but the functions they call are tested independently. So even when QuickCheck is used to test a large program, we always test a small part at a time. Therefore we may expect random testing to work particularly well.

Given this, together with the much greater difficulty of automating systematic testing for Haskell, our choice of random testing is clear.

6.2 Correctness Criteria

The problem of determining whether a test is passed or not is known as the *oracle problem*. One solution is to compare program output with that of another version of the program, perhaps an older one, or perhaps a simpler, slower, but 'obviously correct' version. Alternatively, an executable specification might play the same rôle. This kind of oracle can easily be expressed as a QuickCheck property, although our properties are much more general.

However, often one can check that a program's output is correct much more efficiently than one can compute the output. Blum and Kannan exploit this in their work on *result-checking* [4]: a program checker is defined to be another program which classifies the program's output as correct or buggy, with a high probability of classifying correctly, and does so with strictly lower complexity. They distinguish program *checking* from program *testing*: their proposal is that programs should always check their output, and indeed in further work Blum et al. showed how programs which usually produce correct answers can even *correct* wrong output [5] (in particular domains). Of course, result checkers can also be expressed as QuickCheck properties, although we use them for testing rather than as a part of the final program.

QuickCheck's property language is however more general than result checking. Via conditional properties or specific test data generators, we can express properties which hold only for a subset of all possible inputs. Thus we avoid testing functions in cases which lead to run-time errors, or cases in which we do not care about the result. For example, we do not test insertion into an unordered list — there is no point in doing so. Yet a result checker must verify that a program produces the 'correct' output in *all* cases, even those which are uninteresting. Moreover, QuickCheck properties are not limited to checking the result of an individual function call — the property that an operator is associative, for example, cannot really be said to check the result of any individual use of the operator, but still expresses a useful 'global' property that can be checked by testing.

The idea of testing the properties in a specification directly was used in the DAISTS system [10] for testing abstract data types, which compiled equational properties into testing code, although test cases had to be supplied by the user. Lacking automatic test case generation, DAISTS did not need equivalents of our conditional and quantified properties. Although the language used was imperative, abstract data type operations had to be forbidden to side-effect their arguments, thus the programs to be tested were essentially restricted to be functional. Later work aims to relax this restriction: Antoy and Hamlet describe a technique for testing C++ classes against an algebraic specification, which is animated in order to predict the correct result [1]. However, the specification language must be restricted in order to guarantee that specifications *can* be animated.

There seems to be no published work on automatic testing of *functional* programs against specifications. We simply observe that functional programs and property based specifications are a very good match: we can use the given properties directly for testing. Moreover, embedding the specification language in Haskell permits us to write very powerful and flexible properties, with a minimum of learning effort required.

6.3 Test Data Generation

Commercial random testing tools generate test data in limited domains, with the goal of matching the distribution of actual data for the system under test – the so-called *operational profile*. In this case, statistical inferences about the mean time between system failures can be drawn from the test results.

In order to generate more complex data, it is necessary to provide a description of the data's structure. A popular approach to doing so uses *grammars*. However, it was realised very early that context-free grammars cannot express all the desired properties of test data – for example, that a generated random program contains no undeclared identifiers. Therefore the grammars were enhanced with actions [6], or extended to attribute grammars. This approach has been most used for testing compilers, although Maurer argues for its use in many contexts [15].

Grammars have been used for systematic testing, where for example the generated test data is required to exercise each production at least once. Such an adequacy criterion maybe be particularly appropriate for compiler testing. Maurer also used grammars for random testing [15], and noted the termination problem for recursive grammars. His solution, though, was just to increase the probabilities of generating leaves so that eventual termination is guaranteed. Our experience is that this results in far too high a proportion of trivial test cases, and therefore inefficient testing – more tests must be run to exercise the program properly. We believe our method of controlling sizes is much superior.

It seems that the need to learn a complex language of extended grammars has hindered the adoption of these methods in practice. By embedding a test generator language in Haskell, we provide (at least) the same capabilities, but spare the programmer the need to learn more than a few new operators. At the same time we provide all the power and flexibility needed to generate test data satisfying complex invariants, in a language the programmer already knows. By linking generators to types via Haskell's class system, we relieve the programmer of the need to specify generators at all in many cases, and where they must be specified, the programmer's work is usually limited to specifying generators for his or her own new types.

6.4 On Randomness

We have encountered some interesting problems in reasoning about programs which use random number generation. In particular, the `Gen` monad which QuickCheck is based on is not a monad at all! Consider the first monad law:

```
return x >>= f = f x
```

Since our implementation of bind splits its random number seed to yield the seeds passed to each operand, then `f` is passed *different* seeds on the two sides of the equation, and may therefore produce different results. So the law simply does not hold. Morally, however, we consider the law to be true, because the two sides produce the same *distribution* of results, even if the results differ for any particular seed.

But what, precisely, do we mean by 'morally'? We cannot fix the problem just by reinterpreting equality for the `Gen` type, claiming the two sides are just different representations of the same abstract generator. This isn't good enough, because we can actually observe the difference at other types by supplying a random number seed - something we have to be able to do if the `Gen` type is to be useful. Instead we have to reinterpret what we mean by program equivalence in the presence of random number generation.

We note that this difficulty is by no means confined to Haskell: the imperative program

```
a := random(); b := random(); c := a - b;
```

is morally equivalent to the same program with the assignments reversed in the same sense, but of course produces a different result. There is some interesting semantic theory to be done here.

6.5 On Lazy Evaluation

We have argued in the past that lazy evaluation is an invaluable programming tool, that radically changes the way programs can be structured [13]. Yet QuickCheck is (of course) only able to test computable properties. Is there a conflict here?

In fact, the conflict is much less than one might imagine. As we have shown above, we can perfectly well use infinite structures in specifications, provided the properties we actually test are computable – for example, we can test that arbitrarily long prefixes of infinite lists are equal, rather than comparing the lists themselves. Our `Gen` monad has a lazy bind operation (because we split the random number seed, rather than threading it through first one operand, then the other), and so we can freely define generators that produce infinite results. What we cannot do is *observe* non-termination in a test result. So we cannot test, for example, the property

```
reverse (xs++undefined) == undefined
```

On the other hand, in a sense a human tester cannot observe non-termination either, and if we have been able to test lazy programs satisfactorily by hand so far, then we are not in a worse position if we use QuickCheck. Yet a human tester can observe that `reverse (xs++undefined)` produces an error message (from the evaluation of `undefined`) without producing any other output first, and can thus infer that the property above holds. The problem is that the Haskell standard provides no way for a *program* to make the same observation. Yet there are various extensions of Haskell which do indeed make this possible. Some work done by Andy Gill has shown that, given such extensions, we could formulate and check properties such as the one above using QuickCheck also.

6.6 Some Reflections

We are convinced that one of the major advantages of using QuickCheck is that it encourages us to formulate formal specifications, thus improving our understanding of our programs. While it is open to the programmer to do this anyway, few really do, perhaps because there is little *short term* payoff, and perhaps because a specification is of less value if there is no check at all that it corresponds to the implemented program. QuickCheck addresses both these issues: it gives us a short-term payoff via automated testing, and some reason to believe that properties stated in a module actually hold.

We have observed that the errors we find are divided roughly evenly between errors in test data generators, errors in the specification, and errors in the program. The first category is useless to discover (except insofar as it helps with further testing) – it tells us nothing about the actual program. The third category is obviously useful – in a sense these are the errors we test in order to find. But the second category is also important: even if they do not reveal a mistake in the code, they do reveal a misunderstanding about what it does. Correcting such misunderstandings improves our ability to make use of the tested code correctly later.

When formulating specifications one rapidly discovers the need for a library of functions that implement common mathematical abstractions. We are developing an implementation of finite set theory for use with QuickCheck; many of the abstractions in it are too inefficient to be of much use in programs, but in specifications, where the object is to state properties as clearly and simply as possible, they come into their own. Because of this difference in purpose, there is a need for libraries specifically targeted at specifications.

The major limitation of QuickCheck is that there is no measurement of test *coverage*: it is up to the user to investigate the distribution of test data and decide whether sufficiently many tests have been run. Although we provide ways to collect this information, we cannot compel the programmer to use them. A programmer who does not risks gaining a false sense of security from a large number of inadequate tests. Perhaps we could define adequacy measures just on the generated test data, and thus warn the user at least in this kind of situation.

7. CONCLUSIONS

We have taken two relatively old ideas, namely specifications as oracles and random testing, and found ways to make them easily available to Haskell programmers. Firstly, we provide an embedded language for writing properties, giving expressiveness without the learning cost. The language contains convenient features, such as quantifiers, conditionals and test data monitors. Secondly, we provide type-based default random test data generators, including random functions, greatly reducing the effort of specifying them. Thirdly, we provide an embedded language for specifying custom test data generators, which can be based on the default generators, giving a finer control over test data distribution. We also introduce a novel way of controlling size when generating random elements of recursive data types.

Further, we demonstrate that the combination of these old techniques works extremely well for Haskell. The functional nature allows for local and fine-grained properties, since all dependencies of a function are explicit. And precisely random testing is known to work very well for small, fine-grained programs, and is effective in finding faults.

Lastly, the tool is lightweight and easy to use, and provides a short-term payoff for explicitly stating properties of functions in a program, which greatly increases the understanding of the program, for the programmer as well as for documentation purposes.

Acknowledgements: We would like to thank Andy Gill, Chris Okasaki, and the anonymous referees for their useful comments on this paper.

8. REFERENCES

[1] S. Antoy and R. Hamlet. Automatically checking an implementation against its formal specification. In *Irvine Software Symposium*, pages 29–48, March 1992.

[2] Roland Backhouse, Patrik Jansson, Johan Jeuring, and Lambert Meertens. Generic Programming - An Introduction. In *Lecture notes in Computer Science*, volume 1608, 1999.

[3] P. Bjesse, K. Claessen, M. Sheeran, and S. Singh. Lava: Hardware Design in Haskell. In *International Conference on Functional Programming*, Baltimore, 1998. ACM.

[4] M. Blum and S. Kannan. Designing programs that check their work. In *Proc. 21st Symposium on the Theory of Computing*, pages 86–97. ACM, May 1989.

[5] M. Blum, M. Luby, and R. Rubinfeld. Self-testing/correcting with applications to numerical problems. In *Proc. 22nd Symposium on the Theory of Computing*, pages 73–83. ACM, May 1990.

[6] A. Celentano, S. C. Reghizzi, P. Della Vigna, and C. Ghezzi. Compiler testing using a sentence generator. *Software – Practice & Experience*, 10:897–918, 1980.

[7] K. Claessen and D. Sands. Observable Sharing for Functional Circuit Description. In *Asian Computer Science Conference*, Phuket, Thailand, 1999. ACM Sigplan.

[8] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the Association for Computing Machinery*, 7(3):201–215, 1960.

[9] J. Duran and S. Ntafos. An evaluation of random testing. *Transactions on Software Engineering*, 10(4):438–444, July 1984.

[10] J. Gannon, R. Hamlet, and P. McMullin. Data abstraction implementation, specification, and testing. *Trans. Prog. Lang. and Systems*, (3):211–223, 1981.

[11] D. Hamlet. Random testing. In J. Marciniak, editor, *Encyclopedia of Software Engineering*, pages 970–978. Wiley, 1994.

[12] R. Hamlet and R. Taylor. Partition testing does not inspire confidence. *Transactions on Software Engineering*, 16(12):1402–1411, December 1990.

[13] J. Hughes. Why Functional Programming Matters. In D. Turner, editor, *Research Topics in Functional Programming*. Addison Wesley, 1990.

[14] M. P. Jones. The Hugs distribution. Currently available from `http://haskell.org/hugs`, 1999.

[15] P. M. Maurer. Generating test data with enhanced context-free grammars. *IEEE Software*, 7(4):50–56, 1990.

[16] Gunnar Stålmarck. A System for Determining Propositional Logic Theorems by Applying Values and Rules to Triplets that are Generated from a Formula, 1989. Swedish Patent No. 467 076 (approved 1992), U.S. Patent No. 5 276 897 (1994), European Patent No. 0403 454 (1995).

[17] Philip Wadler. Theorems for free! In *International Conference on Functional Programming and Computer Architecture*, London, September 1989.

[18] Philip Wadler. A prettier printer, March 1998. Draft paper.

[19] H. Zhu, P. Hall, and J. May. Software unit test coverage and adequacy. *Computing Surveys*, 29(4):366–427, December 1997.

Appendix: Implementation

Here, we show the implementation of the QuickCheck library, except for the function `quickCheck`. The source code of QuickCheck is available from `www.cs.chalmers.se/~rjmh/-QuickCheck/`.

```
module QuickCheck where

import Monad ; import Random

-- Gen

newtype Gen a = Gen (Int -> Rand -> a)

choose :: Random a => (a, a) -> Gen a
choose bounds = Gen (\n r -> fst (randomR bounds r))

variant :: Int -> Gen a -> Gen a
variant v (Gen m) = Gen (\n r ->
  m n (rands r !! (v+1)))
 where
  rands r0 = r1 : rands r2 where (r1, r2) = split r0

promote :: (a -> Gen b) -> Gen (a -> b)
promote f = Gen (\n r -> \a ->
  let Gen m = f a in m n r)

sized :: (Int -> Gen a) -> Gen a
sized fgen = Gen (\n r ->
  let Gen m = fgen n in m n r)

instance Monad Gen where
  return a     = Gen (\n r -> a)
  Gen m1 >>= k =
    Gen (\n r0 -> let (r1,r2) = split r0
                      Gen m2  = k (m1 n r1)
                   in m2 n r2)

elements :: [a] -> Gen a
elements xs = (xs !!) `liftM` choose (0, length xs - 1)

vector :: Arbitrary a => Int -> Gen [a]
vector n = sequence [ arbitrary | i <- [1..n] ]

oneof :: [Gen a] -> Gen a
oneof gens = elements gens >>= id

frequency :: [(Int, Gen a)] -> Gen a
frequency xs = choose (1, sum (map fst xs)) >>= (`pick` xs)
 where
  pick n ((k,x):xs) | n <= k    = x
                    | otherwise = pick (n-k) xs

-- Arbitrary ; Coarbitrary

class Arbitrary a where
  arbitrary :: Gen a

instance Arbitrary Bool where
  arbitrary = elements [True, False]

instance Arbitrary Int where
  arbitrary = sized (\n -> choose (-n,n))

instance (Arbitrary a, Arbitrary b) => Arbitrary (a, b) where
  arbitrary = liftM2 (,) arbitrary arbitrary

instance Arbitrary a => Arbitrary [a] where
  arbitrary = sized (\n -> choose (0,n) >>= vector)

instance (Arbitrary a, Arbitrary b) => Arbitrary (a -> b) where
  arbitrary = promote (`coarbitrary` arbitrary)

class Coarbitrary a where
  coarbitrary :: a -> Gen b -> Gen b

instance Coarbitrary Bool where
  coarbitrary b = variant (if b then 0 else 1)

instance Coarbitrary Int where
  coarbitrary n
    | n == 0    = variant 0
    | n < 0     = variant 2 . coarbitrary (-n)
    | otherwise = variant 1 . coarbitrary (n `div` 2)

instance (Coarbitrary a, Coarbitrary b)
      => Coarbitrary (a, b) where
  coarbitrary (a, b) = coarbitrary a . coarbitrary b

instance Coarbitrary a => Coarbitrary [a] where
  coarbitrary []     = variant 0
  coarbitrary (a:as) =
    variant 1 . coarbitrary a . coarbitrary as

instance (Arbitrary a, Coarbitrary b)
      => Coarbitrary (a -> b) where
  coarbitrary f gen =
    arbitrary >>= ((`coarbitrary` gen) . f)

-- Property

newtype Property = Prop (Gen Result)

data Result = Result
  {ok :: Maybe Bool, stamp :: [String], arguments :: [String]}

nothing :: Result
nothing = Result
  {ok = Nothing, stamp = [], arguments = []}

result :: Result -> Property
result res = Prop (return res)

class Testable a where
  property :: a -> Property

instance Testable Bool where
  property b = result (nothing{ ok = Just b })

instance Testable Property where
  property prop = prop

instance (Arbitrary a, Show a, Testable b)
                           => Testable (a -> b) where
  property f = forAll arbitrary f

evaluate :: Testable a => a -> Gen Result
evaluate a = gen where Prop gen = property a

forAll :: (Show a, Testable b) => Gen a -> (a->b) -> Property
forAll gen body = Prop $
  do a   <- gen
     res <- evaluate (body a)
     return (arg a res)
 where
  arg a res = res{ arguments = show a : arguments res }

(==>) :: Testable a => Bool -> a -> Property
True  ==> a = property a
False ==> a = result nothing

label :: Testable a => String -> a -> Property
label s a = Prop (add `fmap` evaluate a)
 where add res = res{ stamp = s : stamp res }

classify :: Testable a => Bool -> String -> a -> Property
classify True  name = label name
classify False _    = property

collect :: (Show a, Testable b) => a -> b -> Property
collect v = label (show v)
```