Shrinking and Showing Functions
(Functional Pearl)

Koen Claessen
Chalmers University of Technology
koen@chalmers.se

Abstract

Although quantification over functions in QuickCheck properties has been supported from the beginning, displaying and shrinking them as counter examples has not. The reason is that in general, functions are infinite objects, which means that there is no sensible show function for them, and shrinking an infinite object within a finite number of steps seems impossible. This paper presents a general technique with which functions as counter examples can be shrunk to finite objects, which can then be displayed to the user. The approach turns out to be practically usable, which is shown by a number of examples. The two main limitations are that higher-order functions cannot be dealt with, and it is hard to deal with terms that contain functions as subterms.

Categories and Subject Descriptors D [*1*]: 1

General Terms Algorithms, Verification

Keywords testing, QuickCheck, counter example

1. Introduction

QuickCheck [2] is a library for stating properties about programs, and for random testing of these properties. An example of a QuickCheck property is the following. Here, a programmer has stated a property that relates the standard Haskell functions `reverse` and `(++)`:

```
prop_ReverseAppend xs (ys :: [A]) =
  reverse xs ++ reverse ys == reverse (xs ++ ys)
```

A property in QuickCheck is just a normal Haskell function, where the arguments of that function are interpreted as universally quantified variables. The property implicitly states that the function should always return `True`.

The programmer in question expects this property to hold polymorphically, i.e. for any type of argument. However, in order for the property to be executable, a concrete type to test with needs to be provided. QuickCheck defines helper types for this purpose, called `A`, `B`, and `C` which contain values `1`, `2`, ..., and only support equality.

Running QuickCheck on the above property yields the following:

```
GHCi> quickCheck prop_ReverseAppend
*** Falsifiable (after 4 tests and 4 shrinks):
[1]
[2]
```

A keen reader might already have noted that the stated property in fact does not hold, because the programmer forgot to swap the occurrences of `xs` and `ys`. And indeed, a counter example is displayed (one value per line for each argument to the property); when `xs = [1]` and `ys = [2]`, the property returns `False`. Counter examples are extremely valuable for understanding why a property failed.

The example also shows another important feature of QuickCheck, namely *shrinking*. The output of QuickCheck reports "...and 4 shrinks". This means that QuickCheck, after finding an initial counter example, has made it smaller in 4 steps. The reported counter example is a (local) minimal counter example. This means that all attempts to make the reported counter example smaller failed. In other words, replacing one of the lists in the counter example by an empty list does not lead to a counter example. And neither does replacing `2` by `1`. Counter examples that are minimal are even more valuable than regular counter examples for understanding why a property failed. We explain more about how shrinking works in the next section.

QuickCheck also supports quantification over functions in properties. As an example, here is a property that states that `map` and `filter` commute:

```
prop_MapFilter f p (xs :: [A]) =
  map f (filter p xs) == filter p (map f xs)
```

However, we encounter a problem if we try to QuickCheck this property:

```
GHCi> quickCheck prop_MapFilter
Error: No instances for (Show (A -> A), Show
(A -> Bool)) arising from a use of ‘quickCheck’
```

The problem is that QuickCheck uses the `show` function to display counter examples, but the function type in Haskell does not support the `show` function. In general, functions may be infinite and thus they are hard to convert into a String.

A common solution to this problem is to import the standard module `Text.Show.Functions`, which contains a trivial definition of `show` for functions. This definition simply produces the string `"<function>"` for every function argument. The result is:

```
GHCi> quickCheck prop_MapFilter
*** Falsifiable (after 3 tests):
<function>
<function>
[3]
```

Haskell'12, September 13, 2012, Copenhagen, Denmark.
Copyright © 2012 ACM 978-1-4503-1574-6/12/09...$10.00

So, we get to see that there is a counter example, but we can not see what the counter example is!

The standard way of dealing with this problem (up to the introduction of the technique presented in this paper) was to manually specify values to show when the property failed. This can be done using the QuickCheck combinator `whenFail`. For example, the property can be annotated as follows:

```
prop_MapFilter f p (xs :: [A]) =
  whenFail (do print fs; print ps) $
    map f (filter p xs) == filter p (map f xs)
 where
  fs = [ (x,f x) | x <- xs ]
  ps = [ (x,p x) | x <- xs ++ map f xs ]
```

The lists `fs` and `ps` are partial function tables of the functions `f` and `p`, respectively. Here, the programmer has decided which arguments to `f` and `p` were interesting to see (something which is not always easy to do; for example it is easy to forget the `map f xs` part in the definition of `ps`).

Testing this property gives:

```
*** Falsifiable (after 8 tests):
<function>
<function>
[6]
[(6,7)]
[(6,False),(7,True)]
```

And indeed, it is now easy to see what went wrong in the property. Note however that, although the technique using `whenFail` gives us a way of looking at values that play a role in a counter example, shrinking still does not work for functions. It seems a daunting task to shrink an object that is inherently infinite.

However, in any terminating computation, and therefore also in a failing property, any function is only applied to a finite number of arguments. In this paper, we show how we can automatically find out which these values are, by shrinking a representation of the function at hand. This process shrinks a function object used in a failing property down to a finite representation, which in turn can be shown as a counter example to the user. Thus, we solve both problems, not being able to show functions and not being able to shrink functions, at the same time.

Perhaps the most surprising trait of our approach is that all this is done in a pure setting. It is easy to use impure language features to find out to what values a function is applied to (and indeed this is what we did in an earlier version of the library). However, it turned out to be tricky to get right and interact in the intended way with shrinking. Therefore, we prefer the solution presented here over an impure one.

2. A quick QuickCheck recap

QuickCheck properties revolve around *generators*, which are used to create random values of a particular type. Generators have the following type:

```
type Gen a
```

We leave this type abstract for the rest of this paper. It suffices to know that `Gen` is a monad. There also exists a type class that associates a default generator with a type:

```
class Arbitrary a where
  arbitrary :: Gen a
  shrink    :: a -> [a]
```

Apart from a default generator, we can also specify a shrinking function, which we talk about later.

Generating functions Indeed, the function type is an instance of this class, too. Here, it is important to clear up a common misconception regarding the generation of functions as test data. In general, it is not a good idea to generate a random function of type, say `Int -> Int` by generating results of type `Int` for a few argument values chosen in advance, and leave the function constant for all other arguments. The reason is that we never know to what arguments the function may be applied to in the property.

As an example, consider the following property:

```
prop_PredicateStrings p =
  p "some long string" ==> p "some other string"
```

The property, somewhat artificially, states that for any predicate `p`, if it is true for some long string, it is also true for some other string. The falsity of this claim is of course readily demonstrated by a predicate that returns `True` for `"some long string"` and `False` for `"some other string"`. However, any method that pre-determines at generation time which arguments are going to be interesting will have a very hard time finding this counter example.

In QuickCheck, randomly generated functions therefore randomly determine the function result for each argument independently. (Which means that it will find the above counter example in 25% of tests.) Thus, functions have an infinite representation at generation time.

Shrinking Each type that has a default generator also has a default *shrinking function*. The shrinking function has type `a -> [a]`. Whenever a counter example of type `a` has been found, the shrinking function suggests smaller variants of that counter example. QuickCheck then performs a greedy search using this shrinking function, always picking the first element in the shrink list that makes the property fail again. The shrinking process stops when the found counter example cannot be shrunk any further.

As an example, consider the following recursive tree datatype:

```
data Tree a = Nil
            | Node a (Tree a) (Tree a)
  deriving ( Show )
```

The standard shrinking function belonging to this datatype is:

```
shrink :: Arbitrary a => Tree a -> [Tree a]
shrink Nil          = []
shrink (Node x p q) =
     [ p, q ]
  ++ [ Node x p' q | p' <- shrink p ]
  ++ [ Node x p q' | q' <- shrink q ]
  ++ [ Node x' p q | x' <- shrink x ]
```

For empty trees, `shrink` returns the empty list; empty trees cannot be shrunk further. For non-empty trees, we first try to shrink to the left and right subtrees; after that we recursively try to only shrink the left subtree, then only the right subtree, and then only the element in the node. We do not try to shrink several components at the same time, since this might lead to a search space explosion.

Most shrinking functions include the general shrinking pattern above; first, we try to shrink to an immediate subcomponent of the same type, and then we try to shrink (in turn) each of the subcomponents. It is also common that a shrinking function adds extra, domain-specific cases to make shrinking more effective.

3. Modifiers

When we want to quantify over a subset of the elements of a type that satisfies a certain invariant, it is common in QuickCheck to define a new type with one constructor called a *modifier*. The generator and shrinking function of this new type respect the intended invariant (whereas the generator and shrinking function of the original type may not).

As an example, consider testing the `insert` function, used in implementing insertion sort. One property of this function says that inserting an element in an ordered list yields an ordered list again. Without using a modifier, the property might look something like:

```
prop_Insert x =
  forAllShrink ordList shrinkOrdList $ \xs ->
    ordered (insert x xs)
```

We use the explicit quantifier `forAllShrink` that takes an explicit generator and a shrinking function as an argument, because the default generator and shrinking function for lists does not generate or shrink ordered lists. We also assume suitable definitions of a generator `ordList` and a shrinking function `shrinkOrdList`.

With modifiers, we say:

```
data OrderedList a = Ordered [a]
  deriving ( Show )

instance (Ord a, Arbitrary a) =>
                    Arbitrary (OrderedList a) where
  arbitrary = (Ordered . sort) `fmap` arbitrary
  shrink    = filter ordered . shrink
```

The property can then be written as:

```
prop_Insert x (Ordered xs) =
  ordered (insert x xs)
```

This looks much more compact and is immediately understandable.

Modifiers are a way of naming subsets of types specified by an invariant. More generally, they are a way of pairing up a generator and a shrinking function by means of a new type that is not necessarily the type we want to quantify over.

Later on, we are going to use a modifier called `Fun` to help us quantify over shrinkable and showable functions. The definition looks has the shape:

```
data Fun a b = Fun (..) (a -> b)
```

A difference with the previous modifier example is that the constructor function now takes two arguments; the first is an internal argument needed for shrinking and showing (usually ignored when quantifying), and the second is the function we are interested in.

The example property from the introduction can then be rewritten as:

```
prop_MapFilter (Fun _ f) (Fun _ p) (xs :: [A]) =
  map f (filter p xs) == filter p (map f xs)
```

Running QuickCheck on the property results in the following:

```
GHCi> quickCheck prop_MapFilter
*** Falsifiable (after 5 tests and 17 shrinks):
{_->1}
{2->True, _->False}
[2]
```

Which shows a concrete counter example that explains what can go wrong in the property. Functions are shown as a finite list of argument-result pairs between `{...}`, with a final catch-all case for all other arguments.

The next two sections describe the design and implementation of the types and functions that the modifier `Fun` is based on. The section thereafter describes `Fun` itself.

4. A datatype for partial functions

Our approach revolves around a datatype `a :-> c` for concrete representations of *partial functions* from `a` to `c`. The idea is to make concrete the finite part of an infinite function needed for falsifying a property, by means of a partial function. Fig. 1 shows the datatype declaration for our partial functions. However, before we discuss the design of the datatype, we take a look at Fig. 2 which shows how it is connected to all other important types in our implementation.

```
data a :-> c where
  Unit  :: c -> (() :-> c)
  Pair  :: (a :-> (b :-> c)) -> ((a,b) :-> c)
  Lft   :: (a :-> c) -> (Either a b :-> c)
  Rgt   :: (b :-> c) -> (Either a b :-> c)
  (:+:) :: (a :-> c) -> (a :-> c) -> (a :-> c)
  Nil   :: a :-> c
  Map   :: (a -> b) -> (b -> a)
                    -> (b :-> c) -> (a :-> c)
```

Figure 1: Datatype for partial functions

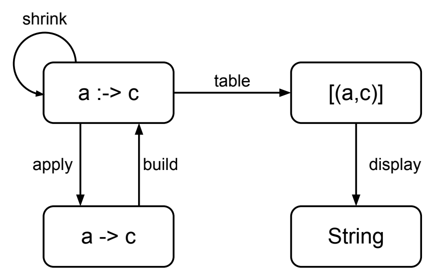

Figure 2: The main types and the conversion functions between them

Partial functions support three major classes of operations: (1) they can be turned into a table of function entries (using the function `table`) which enables showing them, (2) they can be transformed back and forth between normal functions (using the functions `build` and `apply`) which enables creating and using them, and (3) they can be shrunk (using the function `shrink`) which enables the finiteness of the tables. In this section, we present the functions `table`, `apply`, and `shrink`. The next section discusses `build`.

Datatype Elements of the type `a :-> c` are concrete representations of potentially partial functions from `a` to `c`. They inspect the structure of the argument of type `a` and then arrive at the corresponding answer of type `c`. As we can see in Fig. 1, they resemble *generalized tries* [5] in many ways. Just as for tries, we limit `a` to be represented by a polynomial datatype (using only units, products, and sums).

`Unit` constructs a constant function with domain `()`. `Pair` constructs a partial function with pairs as domain using a partial function returning another partial function, by using currying. `Lft` and `Rgt` construct partial functions from sums, which only yield a result if their arguments use a `Left` or a `Right` constructor, respectively. We can glue together two (assumed to be non-overlapping) partial functions using `:+:`. `Nil` is the partial function which never returns anything.

Finally, we provide `Map`, which allows the construction of functions over other types than `()`, `(a,b)` and `Either a b`. `Map` takes a pair of functions `g` and `h` such that `h . g == id`, which are used to convert back and forth between the new type and an already supported type.

```
table :: (a :-> c) -> [(a,c)]
table (Unit c)    = [((), c)]
table (Pair p)    = [((x,y),c) | (x,q) <- table p
                               , (y,c) <- table q]
table (Lft p)     = [(Left x, c) | (x,c) <- table p]
table (Rgt q)     = [(Right y,c) | (y,c) <- table q]
table (p :+: q)   = table p ++ table q
table Nil         = []
table (Map _ h p) = [(h x, c) | (x,c) <- table p]
```

Figure 3: Converting a partial function to a table

```
display :: (Show a,Show c)=> c -> [(a,c)] -> String
display d xys =
     "{"
  ++ intercalate "," (
       [ show x ++ "->" ++ show y
       | (x,y) <- xys
       ] ++
       [ "_->" ++ show d ]
     )
  ++ "}"
```

Figure 4: Displaying a function table

```
papply :: (a :-> c) -> (a -> Maybe c)
papply (Unit c)    _         = Just c
papply (Pair p)    (x,y)     = do q <- papply p x
                                  papply q y
papply (Lft p)     (Left x)  = papply p x
papply (Rgt q)     (Right y) = papply q y
papply (p :+: q)   x         = papply p x `mplus`
                                 papply q x
papply (Map g _ p) x         = papply p (g x)
papply _           _         = Nothing

apply :: c -> (a :-> c) -> (a -> c)
apply d p = fromMaybe d . papply p
```

Figure 5: Applying a partial function

Table The easiest way to understand what each constructor in the `a :-> c` type does is to look at the definition of `table` in Fig. 3, which converts a partial function to a table of entries. It is important to note here that if the input function is infinite, the produced table will be infinite as well; no guarantees are even given as to where in this list a given entry will appear in this case (if at all). If the input function is finite, the produced list will be finite too.

`Unit` results in a table with one entry. `Pair` goes through each entry of its argument function and expands the result function to a table too. `Lft` and `Rgt` label the arguments in the entries with `Left` and `Right`. `:+:` and `Nil` correspond to `++` and `[]` for tables. Lastly, `Map` uses one of its argument functions to convert the argument position in the entries.

Fig. 4 shows `display`, a handy function for turning a table (a list of function entries) into a String. Because all functions we display to the user are total, it also requires a default result `d`, which is added as the `_->d` at the end.

Apply The next functions we discuss are `apply` and `papply`, shown in Fig. 5. The function `papply` does most of the work, taking a partial function and an argument, and possibly yielding a result.

```
shrinkFun :: (c -> [c]) -> (a :-> c) -> [a :-> c]
shrinkFun shr (Unit c) =
  [ Nil ] ++
  [ Unit c' | c' <- shr c ]

shrinkFun shr (Pair p) =
  [ Pair p' | p' <- shrinkFun (shrinkFun shr) p ]

shrinkFun shr (Lft p) =
  [ Lft p' | p' <- shrinkFun shr p ]

shrinkFun shr (Rgt q) =
  [ Rgt q' | q' <- shrinkFun shr q ]

shrinkFun shr (p :+: q) =
  [ p, q ] ++
  [ p  :+: q' | q' <- shrinkFun shr q ] ++
  [ p' :+: q  | p' <- shrinkFun shr p ]

shrinkFun shr Nil =
  []

shrinkFun shr (Map g h p) =
  [ Map g h p' | p' <- shrinkFun shr p ]
```

Figure 6: Shrinking a partial function

The function `apply` guarantees the presence of a result by requiring an extra default result.

For `Unit`, we always return its corresponding result. For `Pair`, we first look up `x` in the argument function, and if it exists, then look up `y` in the result function. `Lft` and `Rgt` only succeed when their arguments match `Left` and `Right` respectively. For `:+:`, we first try the left hand function and then the right hand function. `Map` uses one of its argument functions to convert the argument and then looks up the result in the function. The last line catches three cases: `Nil`, `Lft` with a `Right` argument, and `Rgt` with a `Left` argument.

Shrink The last function we discuss in this section is shrinking, displayed in Fig. 6. Remember that shrinking is supposed to produce a list of smaller (in some way) variants of its argument, that are likely to also be a counter example if its argument is a counter example to a property.

The shrinking function we have here takes one extra argument `shr` of type `c -> [c]`, which is a shrinking function that should be used for the results of the function. An alternative design could have been to instead add `Arbitrary c` as the context. The reason why this does not work is that the result types of our functions change when we use the `Pair` constructor, and normal overloading mechanisms do not have the right information to automatically pick the right shrinking function. We can see this in the `Pair` constructor, which uses `shrinkFun shr` to shrink its results, which are also partial functions.

The constructors `Lft`, `Rgt`, `:+:`, `Nil`, and `Map` all follow the standard structure of a shrinking function. Only `Unit` is slightly different; when shrinking we first try to remove the result completely, and if that does not work we shrink the result.

Even though the case for `:+:` follows the usual pattern, it is still interesting to discuss. The arguments to the shrinking function here are often going to be *infinite* trees. Therefore, it is important that we first try `p` and `q` when shrinking (throwing away the other cases, effectively making a partial function). We normally do this anyway, because we want shrinking steps that throw away a lot (whole subtrees) to be tried first, for efficiency reasons. But here it is absolutely vital we do these first, for termination.

```
class Argument a where
  build :: (a -> c) -> (a :-> c)

instance Argument () where
  build f = Unit (f ())

instance (Argument a, Argument b) =>
                              Argument (a,b) where
  build f = Pair (build `fmap` build (curry f))

instance (Argument a, Argument b) =>
                       Argument (Either a b) where
  build f = Lft (build (f . Left))
        :+: Rgt (build (f . Right))

buildMap :: Argument b => (a->b) -> (b->a)
                       -> (a->c) -> (a:->c)
buildMap g h f = Map g h (build (f . h))
```

Figure 7: Basic build functions

The next section explains how partial functions can be constructed.

5. Building partial functions

We would like to have a function

```
build :: (a -> c) -> (a :-> c)
```

that takes a normal function and constructs the representation of the corresponding partial function. However, the way this is done depends on what the type `a` is, and it can certainly not be done for all types `a`. So, this is a natural point to introduce a type class. We have called it `Argument`, and its definition plus some instances are displayed in Fig. 7.

A small note: The only way of constructing a truly partial function (as opposed to a total one) using our API, is to take a total function and shrink it to a partial one. We will only create total functions in this section; for example, `Nil` is never used here.

The three basic instances are, not surprisingly, `()`, `(a,b)`, and `Either a b`. For `()`, we build a function using `Unit`. For `(a,b)`, we curry the function, build a partial function, and also build partial functions for all the results of that function. For `Either a b`, we build two partial functions, one for `Left` and one for `Right`, and we glue them together using `:+:`.

Finally, an auxiliary function `buildMap` is defined. We can use it to turn a function `a -> c` into a partial function `a :-> c`, but only if we know how to convert back and forth between `a` and another type `b` for which we already know how to build functions. Its usefulness is demonstrated in Fig. 8, where we show `Argument` instances for two standard Haskell types: booleans and lists. Booleans are isomorphic to `Either () ()` and the type `[a]` is isomorphic to `Either () (a,[a])`. Other standard algebraic datatypes, such as `Maybe` and tuple types of various lengths, can be dealt with in a similar way.

Fig. 5 shows an example of a discrete numeric type being handled, namely `Integer`. We use the two's complement representation to convert back and forth between `Integer` and `Either (Bool,Integer) Bool`. Having `Integer`, we can easily support other discrete numeric datatypes such as `Int`, `Char`, `Word32`, etc.

A function that sometimes comes in handy is the following:

```
buildShow :: (Show a, Read a) => (a->c) -> (a:->c)
buildShow f = buildMap show read f
```

```
instance Argument Bool where
  build = buildMap from to
    where
      from False = Left ()
      from True  = Right ()

      to (Left _)  = False
      to (Right _) = True

instance Argument a => Argument [a] where
  build = buildMap from to
   where
    from []     = Left ()
    from (x:xs) = Right (x,xs)

    to (Left _)       = []
    to (Right (x,xs)) = x:xs
```

Figure 8: Build function for booleans and lists

```
instance Argument Integer where
  build = buildMap from to
    where
      from 0    = Right False
      from (-1) = Right True
      from x    = Left (odd x, x `div` 2)

      to (Right False) = 0
      to (Right True)  = -1
      to (Left (b,x))  = bit b + 2*x
```

Figure 9: Build functions for integers

It provides a build function for any type that has a `show` and a `read` function, as long as `read . show` is the identity function. Together with `deriving (Show,Read)` on datatypes, this gives a really easy way of making instances for `Argument`.

6. The Fun modifier

All functions shown in the overview diagram in Fig. 2 have now been implemented. What is left is to put it all together, using the modifier technique presented earlier. The result is shown in Fig. 10. Here, we have introduced a new type `Fun`, with one constructor with 2 arguments. The first argument is our internal representation of functions, the second argument is the actual function we are representing, and which we want our users to use.

The representation for functions we use is a pair of a default result and a partial function. Together, these two pieces of information are enough to construct a corresponding total function. We provide two ways of creating a `Fun` object. The first, `fromFunc`, turns a Haskell function (and a default result) into a `Fun`, using `build`. The second, `fromPartial`, turns a partial function (and a default result) into a `Fun`, using `apply`.

We use `fromFunc` in the definition of `arbitrary`. To generate a `Fun`, we simply generate a Haskell function `f` and a default result `d` using standard QuickCheck generators. The class constraints `CoArbitrary a, Arbitrary c` are needed by QuickCheck to generate `f`, and `Argument a` is needed for `build`.

We use `fromPartial` in the definition of `shrink`. To shrink a `Fun`, we completely ignore the function argument `f`, and we try to shrink the partial function `p` and the default result `d`.

```
data Fun a c = Fun (c, a :-> c) (a -> c)

fromFunc :: Argument a => c -> (a -> c) -> Fun a c
fromFunc d f = Fun (d, build f) f

fromPartial :: c -> (a :-> c) -> Fun a c
fromPartial d p = Fun (d, p) (apply d p)

instance (Show a, Show c) => Show (Fun a c) where
  show (Fun (d,p) _) = display d (table p)

instance ( Argument a
         , CoArbitrary a
         , Arbitrary c
         ) => Arbitrary (Fun a c) where
  arbitrary =
    do f <- arbitrary
       d <- arbitrary
       return (fromFunc d f)

  shrink (Fun (d,p) _) =
       [ fromPartial d p' | p' <- shrinkPartial p ]
    ++ [ fromPartial d' p | d' <- shrink d ]
   where
    shrinkPartial = shrinkFun shrink
```

Figure 10: The Fun modifier

7. Examples

In this section we demonstrate the usefulness of our approach by applying it to a number of different examples. We make use of the following combinator, which prints out the left and right hand side of a failed equation when a property fails:

```
(=?=) :: (Show a, Eq a) => a -> a -> Property
x =?= y =
  whenFail (putStrLn (show x ++" =/= "++ show y)) $
    x == y
```

Folds A beginning student of Haskell might think that `foldr` and `foldl` do the same thing, and write down the property:

```
prop_FoldrFoldl f z (xs :: [A]) =
  foldr f z xs =?= foldl f z xs
```

However, the property fails to pass our tests, but there is no information as to why. After instrumenting the property as follows:

```
prop_FoldrFoldl (Fun _ f) z (xs :: [A]) =
  foldr (curry f) z xs =?= foldl (curry f) z xs
```

we can run QuickCheck:

```
GHCi> quickCheck prop_FoldrFoldl
*** Falsifiable (after 6 tests and 31 shrinks):
{(5,5)->1, _->5}
1
[1,5]
5 =/= 1
```

and we get a counter example that explains what is going on. Note that we used `curry f` instead of `f` in the property because we prefer to show the function tables as uncurried functions, which is more compact.

The student now learns about `foldr1`, and expects the following simple relationship to hold between `foldr` and `foldr1`:

```
prop_Foldr1 (Fun _ f) (x,xs) =
  foldr (curry f) x xs =?= foldr1 (curry f) (x:xs)
```

Alas, when we run QuickCheck:

```
GHCi> quickCheck prop_Foldr1
*** Falsifiable (after 3 tests and 25 shrinks):
{(2,3)->1, _->3}
(2,[3])
3 =/= 1
```

we find out that this does not hold! The more complicated property:

```
prop_Foldr1 (Fun _ f) (x,xs) =
  foldr (curry f) x xs =?=
    foldr1 (curry f) (xs++[x])
```

does go through however.

Some large string Remember the property we used to motivate generating good functions when testing? Here it is again, instrumented with a `Fun` modifier:

```
prop_PredicateStrings (Fun _ p) =
  p "some long string" ==> p "some other string"
```

Perhaps it is surprising to see what happens when this property is run through QuickCheck.

```
GHCi> quickCheck prop_PredicateStrings
*** Falsifiable (after 1 test and 163 shrinks):
{"some long string"->True, _->False}
```

After half a second or so, the shrinking functions technique has found the relevant string the predicate has been applied to. This is surprising, because all it can do is execute the program as a black box, and apply search!

In 50% of the runs the result looks different though:

```
GHCi> quickCheck prop_PredicateStrings
*** Falsifiable (after 3 tests and 177 shrinks):
{"some other string"->False, _->True}
```

Which string is eventually singled out as the interesting case solely depends on which boolean result is chosen as the default value.

Binary heaps Suppose we are implementing a `Heap` datatype as a binary tree.

```
data Heap a = Empty
            | Node a (Heap a) (Heap a)
  deriving ( Show )
```

To set things up for use with QuickCheck, an invariant is defined:

```
invariant :: Ord a => Heap a -> Bool
invariant Empty          = True
invariant p@(Node x _ _) = top x p
 where
  top x Empty        = True
  top x (Node y p q) = x <= y && top y p && top y q
```

We also define a generator for heaps which satisfy the invariant by construction.

Now, we might be tempted to provide the following function:

```
hmap :: (a -> b) -> Heap a -> Heap b
hmap f Empty        = Empty
hmap f (Node x p q) = Node (f x) (hmap f p)
                                 (hmap f q)
```

A good programmer will add properties that check that all functions preserve invariants:

```
prop_Hmap (Fun _ (f :: OrdA -> OrdB)) p =
  invariant p ==> invariant (hmap f p)
```

We use OrdA and OrdB, which are like A and B but also support the Ord class with a total ordering. Note that the condition invariant p is not strictly necessary.

QuickChecking the property gives:

```
GHCi> quickCheck prop_Hmap
*** Falsifiable (after 10 tests and 21 shrinks):
{2->2, _->1}
Node 2 Empty (Node 3 Empty Empty)
```

We can see that the function in the counter example is not monotonic, which destroys the heap invariant.

Monad laws In this example, we are dealing with a specific type M that looks and feels like a monad. For the sake of our example, M is a huge simplification from *behaviors* as introduced in functional reactive programming [3]. The datatype M for behaviors in this example looks as follows:

```
data M a
  = Step (M a)
  | Emit a (M a)
  | Stop
 deriving ( Eq, Show )
```

Step waits for a global clock to tick, Emit produces a value, and Stop stops. One operation that is supported on this datatype is synchronous composition:

```
(+++) :: M a -> M a -> M a
Stop     +++ q        = q
p        +++ Stop     = p
Emit x p +++ q        = Emit x (p +++ q)
p        +++ Emit x q = Emit x (p +++ q)
Step p   +++ Step q   = Step (p +++ q)
```

All steps are lined up, and emits between the same clock ticks are combined. Using synchronous composition, we can make M an instance of the Monad class:

```
instance Monad M where
  return x = Emit x Stop

  Stop     >>= k = Stop
  Step m   >>= k = Step (m >>= k)
  Emit x m >>= k = k x +++ (m >>= k)
```

Whenever a value is emitted, a new behavior is spawned off and synchronized with the current one. For someone familiar with the problem domain, these definitions feel natural. But, is M really a monad under this definition? Let us find out, using QuickCheck. We can state the three monad laws:

```
prop_ReturnBind x (Fun _ (k :: A -> M B)) =
  (return x >>= k) == k x

prop_BindReturn (m :: M A) =
  (m >>= return) == m

prop_BindBind (m :: M A) (Fun _ k1)
                         (Fun _ (k2 :: B -> M C)) =
  (m >>= (\x -> k1 x >>= k2)) == ((m >>= k1) >>= k2)
```

The first two properties go through without a problem. But the third one, prop_BindBind yields the following counter example:

```
GHCi> quickCheck prop_BindBind
*** Falsifiable (after 9 tests and 42 shrinks):
Emit 1 (Emit 1 Stop)
{_->Emit 1 (Step (Emit 1 Stop))}
{_->Emit 1 (Step (Emit 2 Stop))}
```

After studying the values of the left and right hand sides, it turned out that the order of emits changes when we change the associativity. Back to the drawing board.

8. Discussion and Conclusions

We have presented a solution to both problems identified in the introduction that are related to quantification over functions in QuickCheck; that they cannot be shown as counter examples, and that they cannot be shrunk. The solution was to shrink the function to a finite object which can then be shown. The technique presented here has been implemented and is part of the current standard QuickCheck distribution. Quantification over functions has not been used much traditionally by QuickCheck users, and we hope that the technique presented here can make quantification over functions more useful in practice. It has already been proven valuable for testing the typical higher-order library functions such as map and filter over datatypes, but also for testing scheduling functions, and in combination with polymorphic testing [1].

As stated in the introduction, it is possible to do much of what is presented here without the fancy use of GADTs and type classes, but using IORefs and unsafePerformIO instead. A function can easily store its argument in an IORef each time it is applied, and when the property fails this IORef can be read and analyzed. One of the early versions of the library was actually based on an unsafe technique like this. However, we decided to abandon it for 3 reasons: (1) the library sometimes behaved in unexpected ways due to the fact that during shrinking, functions are also applied in non-failing properties, (2) the library did not work at all when parallel shrinking was switched on, and (3) most importantly, a purely functional solution seemed more satisfactory than an impure one.

It is surprising however that the current approach, which is a general solution based on search using a black-box oracle, is still quite effective at finding out to which arguments a function is applied in a property.

Still worth considering is a slight variant of our approach that instead of using a concrete default value for the function, uses a special exception making the property succeed instead. In this way, the partial function is guaranteed to always cover all cases that are used in the failed property, and the default case (displayed as _->d in the function tables) is not needed. This is akin to the way exceptions are used in lazy SmallCheck [6]. An advantage of this is that it is even easier to understand the counter examples. This is future work.

We found that the modifier idiom, which up to now was just used for capturing invariants, worked nicely with this problem setting. So, one contribution of this paper can be seen as extending this modifier idiom. We have already found other situations in which hiding shrinking information in the modifier type leads to an elegant solution to a problem.

At the moment, QuickCheck generates functions using the CoArbitrary class, and the machinery for shrinking and showing functions is built on top of QuickCheck, introducing a new class Argument. It is future work to see if and how these two classes can be integrated. However, CoArbitrary can deal with general higher-order functions. We have ideas on how to extend the Argument approach to second-order functions, but it is hard to see how it could work for general higher-order functions. So, for now, these two classes are kept separated.

Another limitation of the approach shows itself when the functions we quantify over are embedded in another type, for example a recursive datatype with a function here and there. We require each function we quantify over to be made explicit, because we need to also have its representation as a partial function. One solution, which is quite cumbersome but doable, is to make a shadow copy of the datatype that uses Fun a b instead of a -> b, and convert the shadows to the real thing after quantifying. Another is to simply always use `Fun a b` instead of `a -> b` in your programs at those place, which is perhaps too invasive.

As a final remark, we believe that there are connections between shrinking going from infinite objects to finite objects as presented here, and the infinite sets admitting exhaustive search by Escardo [4]. Investigating this is part of future work.

Acknowledgments We thank David Sands for providing us with the `foldr1` example, and the anonymous referees for their helpful comments.

References

[1] J. Bernardy, P. Jansson, and K. Claessen. Testing polymorphic properties. In *Proc. of European Symposium on Programming (ESOP)*. Springer LNCS, 2010.

[2] K. Claessen and J. Hughes. QuickCheck: A lightweight tool for random testing of Haskell programs. In *Proc. of International Conference on Functional Programming (ICFP)*. ACM SIGPLAN, 2000.

[3] C. Elliott. Functional implementations of continuous modeled animation. In *Proc. of PLILP/ALP*. Springer LNCS, 1998.

[4] M. Escardo. Infinite sets that admit fast exhaustive search. In *Proc. of Logic in Computer Science (LICS)*. IEEE, 2007.

[5] R. Hinze. Generalizing generalized tries. *J. Funct. Program.*, 10(4): 327–351, July 2000. ISSN 0956-7968.

[6] C. Runciman, M. Naylor, and F. Lindblad. SmallCheck and Lazy SmallCheck – automatic exhaustive testing for small values. In *Proc. of Haskell Symposium*. ACM SIGPLAN, 2008.