SmartCheck: Automatic and Efficient Counterexample Reduction and Generalization

Lee Pike
Galois, Inc.
leepike@galois.com

Abstract

QuickCheck, developed by Claessen and Hughes, is a powerful library for automatic test-case generation. Because QuickCheck performs random testing, some of the counterexamples discovered are very large. QuickCheck provides an interface for the user to write shrink functions to attempt to reduce the size of counterexamples. Hand-written implementations of shrink can be complex, inefficient, and consist of significant boilerplate code. Furthermore, shrinking is only one aspect in debugging: counterexample generalization is the process of extrapolating from individual counterexamples to a class of counterexamples, often requiring a flash of insight from the programmer. To improve counterexample reduction and generalization, we introduce *SmartCheck*. SmartCheck is a debugging tool that reduces algebraic data using generic search heuristics to efficiently find smaller counterexamples. In addition to shrinking, SmartCheck also automatically generalizes counterexamples to formulas representing classes of counterexamples. SmartCheck has been implemented for Haskell and is freely available.

1. Introduction

The QuickCheck testing framework was a revolutionary stepforward in type-directed random testing [3, 4]. Originally designed for Haskell, QuickCheck has been ported to other languages and is a now a widely-used testing tool. Because QuickCheck generates random values for testing, counterexamples it finds may be substantially larger than a minimal counterexample. In their original QuickCheck paper [3], the authors report the following user experience by Andy Gill:

> Sometimes the counterexamples found are very large and it is difficult to go back to the property and understand why it is a counterexample.

QuickCheck defines a type class `Arbitrary` that presents a method `arbitrary` for generating random values of a given type. Gill added another method to the type class:

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

The purpose of `smaller` is to generate strictly smaller values, according to some measure, from a given counterexample. These new values are then tested to attempt to find a smaller counterexample. Today, `smaller` is called `shrink`.

In industrial uses, shrinking is essential. In describing commercial applications of QuickCheck, Hughes has noted that "without it [shrinking], randomly generated failing cases would often be so large as to be almost useless." [10]. Hughes *et al.* also give an extended example in which shrinking is essential in debugging telecom software [1].

Defining an efficient and effective shrink method requires a good understanding of how shrinking in QuickCheck works and the semantics of the property and program being evaluated. Bad definitions can be so slow or so ineffective at shrinking that they are unusable.

In addition, shrinking is one side of the coin when it comes to making counterexamples more understandable: the other side is extrapolation from individual counterexamples to a class of counterexamples. This leap of abstraction is often implicitly made by the programmer in determining the reason why counterexamples fail the property. For example, the following is a relatively small counterexample returned when QuickChecking a property in (a bug-injected version of) Xmonad, a popular X11 window manager written in Haskell [18]:

```
StackSet {current = Screen {workspace = Workspace
{tag = NonNegative {getNonNegative = 0}, layout =
-1, stack = Just (Stack {focus = 'S', up = "",
down = ""})}, screen = 1, screenDetail = 1},
visible = [Screen {workspace = Workspace {tag
= NonNegative {getNonNegative = 2}, layout =
-1, stack = Nothing}, screen = 2, screenDetail
= -1},Screen {workspace = Workspace {tag =
NonNegative {getNonNegative = 3}, layout = -1,
stack = Nothing}, screen = 0, screenDetail =
-1}], hidden = [Workspace {tag = NonNegative
{getNonNegative = 1}, layout = -1, stack =
Just (Stack {focus = '\NUL', up = "", down =
""})},Workspace {tag = NonNegative {getNonNegative
= 4}, layout = -1, stack = Just (Stack {focus =
'I', up = "", down = ""})}], floating = fromList
[]}
```

(This counterexample uses Haskell's default `Show` instances, which uses record syntax.) Programmers may be familiar with having to debug a "wall of text" as shown above. What if instead a formula like the following were returned, stating that for any well-typed values x_0, x_1, x_2, and x_3, tested, a counterexample is found?

```
forall values x0 x1 x2 x3:
  StackSet
    (Screen (Workspace x0 (-1) (Just x1)) 1 1)
    x2 x3 (fromList [])
```

The formula quantifies away all the irrelevant portions of the data structure with respect to the property, so that the user can focus on the heart of the problem in a class of counterexamples.

SmartCheck Motivated by the problems of reducing and generalizing large counterexamples, we developed *SmartCheck*. SmartCheck takes a counterexample produced by some oracle and generically minimizes and generalizes the counterexample.

As shown in the motivating example in Section 2, SmartCheck can shrink more than an order-of-magnitude faster than even the most efficient hand-written shrink implementations for some properties and programs. As well, the time required to shrink scales linearly with the size of the counterexample—other common shrink implementations often scale in polynomial time or worse. (Shrinking time is important in real-world uses; see Section 7 for examples.)

After presenting some preliminary definitions in Section 3, in Section 4, we describe the generic shrinking algorithm implemented. The algorithm is particularly novel in that it generates new random sub-values during the reduction phase (see related work in Section 8).

SmartCheck implements three novel approaches to automatically generalize counterexamples, which are described in Section 5. The first algorithm universally quantifies sub-values that always fail in tests. The second algorithm existentially quantifies sub-values for types in which every possible variant fails the property. For example, finding counterexamples (`Left 2`) and (`Right True`) for the type

```
Either Int Bool
```

means there exists a counterexample regardless of the variant chosen. Existential generalization is useful for large sum types, as found in abstract syntax tree (AST) definitions, for example.

The third algorithm automatically strengthens properties by omitting "similar" counterexamples to the ones previously observed. The algorithm is motivated by noting that there are often multiple ways in which a property may fail; for example, a property stating that pretty-printing an AST and then parsing it results in the original AST may fail due to multiple bugs. During testing, it is useful to discover counterexamples arising from each bug in one go. In practice, the problem is solved by discovering a counterexample `cex`, abstracting it, and then adding a new precondition to the property that informally says "omit counterexamples of form `cex`." Adding preconditions manually is laborious and may cause the programmer to make premature fixes to the program, if she believes she has isolated the error before she actually does.

We describe our implementation based on generic programming in Section 6; the implementation is open-source. In Section 7, we discuss some of our experiences with using SmartCheck, including checking properties from XMonad and a natural language processing library.

2. A Motivating Example

<<need to test with arbitrary for ints turned off. (Use newtype)>>

<<redo analysis, check text>> In this section, we motivate in more detail the challenges in shrinking counterexamples (we focus on shrinking and only touch on generalization here). Consider the example in Figure 1.[1] Data type `T` is a product type containing four lists of wrapped `Int16`s and a single wrapped `Word8`. For our purposes, it is not important what `T` models, but for concreteness, suppose it models the values in four linked lists in an operating system, together with an 8-bit status register.

```
type I   = [Int16]
data T   = T (Word8) I I I I

toList :: T -> [[Int16]]
toList (T w i0 i1 i2 i3) =
  [fromIntegral w] : (map . map) fromIntegral rst
  where
  rst = [i0, i1, i2, i3]

pre :: T -> Bool
pre t = all ((>) 256 . sum) (toList t)

post :: T -> Bool
post t = (sum . concat) (toList t) < 5 * 256

prop :: T -> Property
prop t = pre t ==> post t
```

Figure 1: Example program and property.

Now suppose we are modeling some program that serializes values of type `T`. The input to the program satisfies the invariant `pre`, that the sum of values in each list of `Int16`s is less than or equal to 256 (and by definition, the `Word8` value is less than or equal to 256). Assuming this, we want to show `post` holds, that the sum of all the values from `T` is less than $5 * 256$, where five is the number of fields in `T`. At first glance, the property seems reasonable. But we have forgotten about underflow; for example, the value

```
T 10 [-20000] [-20000] [] []
```

satisfies `pre` but fails `post` (the `==>` operator in the figure is implication from the QuickCheck library).

Despite the simplicity of the example, a typical counterexample returned by QuickCheck can be large. With standard settings and no shrinking, the median counterexample discovered contains around 70 `Int16` values, and counterexamples can contain over 100 values. <<recheck>> Thus, it pays to define `shrink`!

We might first naively try to shrink counterexamples for a data type like `T` by taking the cross-product of shrunk values over the arguments to the constructor `T`. This can be expressed using Haskell's list-comprehension notation:

```
shrink (T w i0 i1 i2 i3) =
  [ T a b c d e | a <- shrink w,  b <- shrink i0
                , c <- shrink i1, d <- shrink i2
                , e <- shrink i3 ]
```

While the above definition appears reasonable, there are two problems with it. First, the result of (`shrink t`) is null if any list contained in `t` is null, in which case `t` will not be shrunk. More troublesome is that with QuickCheck's default `Arbitrary` instances, the length of potential counterexamples returned by `shrink` can easily exceed 10^{10}, which is an intractable number of tests for QuickCheck to analyze. The reason for the blowup is that shrinking a list `[a]` produces a list `[[a]]`, the length of which is significantly longer than the original list. Then, we take the cross-product of the generated lists. For the example above, with some counterexam-

[1] All examples and algorithms in this paper are presented in Haskell 2010 [8] plus the language extensions of existential type quantification and scoped type variables. We use Haskell `Prelude` functions, `mapMaybe` from `Data.Maybe`, and `liftM` and `replicateM` from `Control.Monad`.

ples, the shrinking stage may appear to execute for hours, consuming ever more memory, without returning a result.

A programmer might try to control the complexity by truncating lists using the (`take n`) function that returns the first n elements of a list. The trade-off is quicker shrinking with a lower probability of finding a smaller counterexample. For example, we might redefine shrink as follows:

```
shrink (T w i0 i1 i2 i3) =
  [ T a b c d e | a <- tk w
                , b <- tk i0, c <- tk i1
                , d <- tk i2, e <- tk i3 ]
  where tk x = take 10 (shrink x)
```

Call this version "truncated shrink". Truncation controls the blowup of the input-space; the maximum number of possible new values is 10^5 in this case. While truncation controls the blowup, the downside is that potentially smaller counterexamples may be omitted.

A more clever programmer that understands the semantics of QuickCheck's `shrink` implementation[2] defines a `shrink` instance as follows:

```
shrink (T w i0 i1 i2 i3) = map go xs
  where xs = shrink (w, i0, i1, i2, i3)
        go (w', i0', i1', i2', i3')
           = T w' i0' i1' i2' i3'
```

This "tuple shrink" definition does not suffer the same shortcomings: it shrinks a value even if it contains an empty list, and the combinatorial blowup of shrink candidates is avoided, since a pair `(a,b)` is shrunk by attempting to shrink `a` while holding `b` constant, then attempting to shrink `b` while holding `a` constant (and is generalized to larger tuples). Thus, each argument to `T` is independently shrunk.

Still, using this definition, from some initial counterexamples, shrinking may take over one minute and result in a final counterexample containing just under 100 `Int16` values.[3]

SmallCheck is another testing framework for Haskell for which shrinking is irrelevant: SmallCheck is guaranteed to return a smallest counterexample, if one exists [17]. SmallCheck does this by enumerating all possible inputs, ordered from smallest to largest, up to some user-defined bound. While SmallCheck is effective for testing many programs and properties (in accordance with the *small scope hypothesis* [11]), counterexamples to even relatively simple properties may be infeasible to discover due to input-space explosion.

For a product type, SmallCheck must check values produced by taking the cross-product of the type's fields at a given depth. Unfortunately, using default instances, SmallCheck fails long before it reaches a depth required to discover a counterexample. (A related library named Feat combines some aspects of SmallCheck and QuickCheck [6]; we discuss it in Section 8).

<<QC without shrinking Ints>>

	QC none	QC trunc	QC tuple	SmartCheck
Mean	0.08s, 69	0.53s, 35	2.24s, 12	0.10s, 7
Median	0.07s, 69	0.17s, 33	1.81s, 13	0.10s, 6
95%	0.14s, 100	1.76s, 65	4.06s, 19	0.18s, 12

Table 1. Summarizing data for the graphs in Figure 2. Entries contain execution time (in seconds) and counterexample sizes (counting `N` constructors).

[2] The following approach was suggested to the author by John Hughes.

[3] All results reported in the paper are with a GHC-7.6.3-compiled program, using `-O2`, on a 4-core 2.7GHz Intel i7 processor.

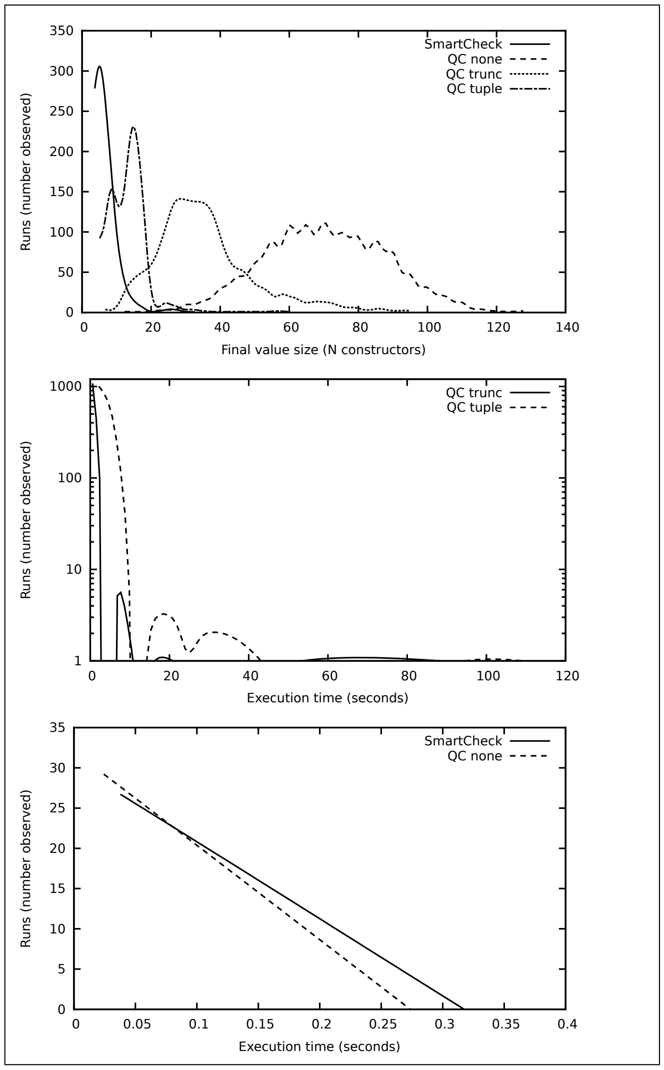

Figure 2: Results for 2000 tests of the program in Figure 1.

To motivate the benefits of SmartCheck reduction algorithm, consider Table 2 and the corresponding graphs in Figure 2, giving the results for using QuickCheck and SmartCheck on the program shown in Figure 1 (we omit SmallCheck, since as noted, it is not even feasible for the example with default instances). We graph the final counterexample size and execution time. Out of 2000 runs, we show the number of runs resulting in a shrunk counterexample of a given size (by counting `N` constructors), and we show the number of runs executing within a given time limit (in seconds). Note that we provide two execution-time graphs, since time scales differ dramatically (we use a logrithmic scale for the y axis in one graph and a linear scale in the other). In the table, we show the mean, median, median, and the results at the 95th percentile values. (We give standard deviations, but not that the plots are not necessarily Gaussian.)

The experiments show three approaches to shrinking with QuickCheck: no shrinking (QC none), to provide a baseline, truncated shrinking (QC trunc), as described above, in which each list is truncated to 10 elements, and tuple shrinking (QC tuple), also described above. SmartCheck produces smaller counterexamples than the other approaches, with a very small tail.

More dramatic, however, are the differences in execution time. The execution times for SmartCheck include the initial counterexample generation time required by QuickCheck (QC none); thus the time required by SmartCheck to shrink a counterexample is on average just two hundredths of a second (0.10 - 0.08). As can be seen in the graphs, SmartCheck's execution time scales linearly (like QuickCheck without shrinking). Truncation and tuple shrinking produce very long tails; the longest execution time recorded for truncation shrinking and tuple shrinking are 88 seconds and 110 seconds, respectively, although these outliers can vary wildly. SmartCheck is largely insensitive to the size of the original counterexample provided.

Finally, these performance results for SmartCheck are fairly generalizable to other types and properties, since the algorithm is not dependent on the specific program or property being checked. However, the results are particularly pronounced for counterexamples that are particularly "deep" or "broad" in terms of their tree structure. We now describe in detail SmartCheck's algorithm for generic shrinking.

3. Preliminaries

SmartCheck focuses on algebraic data types represented as "sums of products" including recursive and mutually recursive types. We sometimes refer to the elements of a sum type as a *variant* that is *tagged* by a constructor. For example, the type

```
Maybe a = Just a | Nothing
```

contains two variants, tagged by the constructors `Just` and `Nothing`, respectively.

To present the algorithms in the following sections, we provide some definitions in the form of methods of a `SubTypes` type class. (SmartCheck requires instances to be defined for data being analyzed; in Section 6, we describe how instances for the type class are derived automatically.)

```
type Idx    = Int
type Size   = Int
data SubVal = forall a. SubTypes a => SubVal a

class Arbitrary a => SubTypes a where
  size    :: a -> Size
  index   :: a -> Idx -> Maybe SubVal
  replace :: a -> Idx -> SubVal -> a
  constr  :: a -> String
  constrs :: a -> [String]
  opaque  :: a -> Bool
  subVals :: a -> Tree SubVal
```

The `SubTypes` type class requires QuickCheck's `Arbitrary` as a super-class. `SubTypes` has the following methods:

- `size` returns the size of a value—intuitively, the number of constructors contained within it,
- `index` returns a *sub-value* at a breadth-first index in a value,
- `replace` replaces a sub-value at a particular focus (returning the original value if the index is out-of-bounds),
- `constr` returns a string representation of the constructor tagging the value,
- `constrs` returns the list of all possible constructors from the value's type,
- `opaque` is false when the type of the value is an "interesting type"; informally, this is a type other than a primitive type like `Int` or `Char`. See Section 4.2.2 for a full discussion.
- `subVals` returns a tree of all non opaque-type sub-values. A tree is a value with the type

```
data Tree a = Node { rootLabel :: a
                   , subForest :: [Tree a] }
```

To illustrate typical evaluations of the methods, consider a binary tree type:

```
data T = L | B T T
```

and the value `tree`, labeled with indexes in a breadth-first order:

```
tree = B₀ (B₁ L₃
             (B₄ L₆ L₈))
          (B₂ L₅ L₇)
```

Here are example applications of `SubTypes` methods; in the following, we show the indexes with respect to the value `tree`:

```
size tree = 9

index tree 4  = (Just . SubVal) (B₄ L₆ L₈)
index tree 12 = Nothing

replace tree 2 (SubVal L) =
  B₀ (B₁ L₃
       (B₄ L₆ L₈))
     L

constr  tree = ["B"]
constrs tree = ["B", "L"]
constrs L    = ["B", "L"]

opaque (3 :: Int) = True
opaque tree       = False
opaque L          = False

subVals (B (B L₀ L₁) L₂) =
  Node (B L₀ L₁) [Node L₀ [], Node L₁ []]
       (Node L₁ [])
```

The `SubVal` type is an existential data type, used as a generic container for sub-values from a counterexample. We will sometimes refer to the unwrapped value returned by `index a i` as the *ith sub-value of a*, so for example, (B_4 L_6 L_8) is the 4th sub-value of `tree`. An invariant of `index` is that for any value `a`, and for the smallest $i \geq 0$ such that

```
index a i == Nothing
```

then for all $0 \leq j < i$,

```
index a j /= Nothing
```

We use this invariant as a termination case in recursive algorithms over the sub-values of a value. (Rather than indexes into a data-structure, an alternative representation is to use a zipper data structure [9] to traverse data. We have chosen explicit indexes to write simple tail-recursive algorithms that can easily be transcribed to imperative languages.)

In our implementation, `constr` and `constrs` depends on GHC Generics [15], which we describe in Section 6. For simplicity, we omit here Generics-specific super-class constraints on the `SubTypes` class. Moreover, our presentation simplifies the implementation (Section 6) somewhat to improve the presentation.

4. Shrinking Data

In this section, we describe how to efficiently and generically shrink algebraic data values. Recall the basic idea behind the `shrink` method of the `Arbitrary` class: generate a list of values, each of which is smaller than the current counterexample. Each of the new values generated may not bear any relationship to the original counterexample other than being smaller. SmartCheck pursues an approach that searches for smaller but structurally similar counterexamples, as we make precise below.

We describe the algorithm in Section 4.1 and then describe algorithmic details in Section 4.2. Some optimizations to the reduction algorithm are described in Section 4.3.

4.1 Reduction Algorithm Overview

The algorithm we present for efficiently searching for new counterexamples is an instance of greedy breadth-first search over a tree structure that represents a value. At each node, during the traversal, we generate arbitrary structurally smaller sub-values and build a new value from that, leaving the remainder of the tree unchanged. Intuitively, by a *structurally smaller value*, we mean one with fewer constructors. We continue until we reach a fixed-point.

```
getSize :: SubVal -> Size
getSize (SubVal a) = size a

newVals :: Size -> Int -> SubVal -> IO [SubVal]
newVals sz tries (SubVal a) =
  replicateM tries s where
  s = liftM SubVal (sizedArbitrary sz a)

reduce :: SubTypes a
  => ScArgs -> (a -> Property) -> a -> IO a
reduce args prop cex = reduce' 1
  where
  reduce' idx
    | Just v <- index cex idx
    = do vs <- newVals (getSize v)
                 (scMaxReduce args) v
         maybe (reduce' (idx+1)) (reduce args prop)
               (test cex idx vs prop)
    | otherwise = return cex

test :: SubTypes a => a -> Idx -> [SubVal]
     -> (a -> Property) -> Maybe a
test cex idx vs prop = go vs
  where
  go []      = Nothing
  go (v:vs') =
    let cex' = replace cex idx v in
    if pass prop cex' then go vs'
      else Just cex'
```

Figure 3: Counterexample reduction algorithm.

Figure 3 shows the reduction algorithm. In this algorithm and subsequent algorithms in the paper, functions in **bold font** are left undefined but their implementation is described in the text. The function `reduce` takes flags to customize the algorithm's behavior, a counterexample `cex`, and the property `prop`. The reduction begins at the first proper sub-value of `cex`; call it `v`. When the index `idx` becomes out-of-bounds and returns `Nothing`, the algorithm terminates. Otherwise, a list of new random values are generated.

```
sizedArbitrary :: SubTypes a => Size -> a -> IO a
```

generates a new value `v'` having the same type as `v` and that is strictly smaller (with respect to the `size` method) than `v`. Just like QuickCheck's `arbitrary` method, `sizedArbitrary` generates successively larger counterexamples when generating new values with which to replace a sub-value.

The flag `scMaxReduce` is the maximum number of tries to discover a new counterexample by replacing `v` in `cex` and testing it. The result of `pass prop cex'` for

```
pass :: (a -> Property) -> a -> Bool
```

holds if `cex'` satisfies the property `prop`. The property may be a conditional, in which case the value must pass the precondition as well as the consequent for `pass` to return `True`. If no failure is found, we move to the next sub-value of `cex` and continue. However, if a new smaller counterexample `cex'` is found, we start a new breadth-first traversal of `cex'`, attempting to shrink it further.

The algorithm is guaranteed to terminate: informally, the measure for the function is that either the index increases or the size of the counterexample being evaluated decreases. The algorithm's complexity is $\mathcal{O}(n^2)$, where n is the number of constructors in the counterexample, assuming that generating new sub-values and testing them is done in constant time.

4.2 Reduction Algorithm Details

Having described the reduction algorithm, there are two important details about its design we describe below.

4.2.1 Variant Counterexample Hypothesis

A motivation for the design of the reduction algorithm is something we call the *variant counterexample hypothesis*: in the search space of possible values from a given type `T`, if a known counterexample `cex` is a variant `v` of `T`, then it is most probable that other counterexamples are also from variant `v`. As an example supporting the hypothesis, consider a property about unintended variable capture over a language's parse tree represented by a sum type with constructors for module imports, function definitions, and global-variable assignments, respectively. A function definition counterexample can only be reduced to smaller function definition counterexamples, the only construct in which variable capture is possible.

Recall that the algorithm begins at the *first* sub-value of the counterexample rather than the 0th sub-value. The 0th sub-value of `cex` is `cex` itself. No invariant of the algorithm would be violated by beginning with the 0th sub-value, and in particular, the algorithm would still terminate. However, we begin with a strict sub-value because of the hypothesis.

The hypothesis may be incorrect for some properties, in which case SmartCheck may potentially fail to discover a smaller counterexample. However, in Sections 5.2 and 5.3, we describe an approaches to generalize counterexamples based on discovering new counterexample variants. These generalization techniques are executed in an (optional) generalization phase, run after the reduction phase, in which this hypothesis is implemented.

4.2.2 Opaque Types

SmartCheck focuses on efficiently shrinking and generalizing large data structures. It is not intended as a general replacement for QuickCheck's `shrink` method. Consequently, SmartCheck ignores "primitive" types without value constructors, such as `Char`, `Int`, and `Word16`. Our experience is that for the kinds of properties with counterexamples that contain massive data structures, shrinking primitive types does not significantly help in understanding them. Furthermore, by ignoring these types by fiat, shrinking time is dependent only on the size of a data structure as measured by the number of constructors.

We generalize the idea of ignoring primitive types by introducing the concept of *opaque types*. If the reduction algorithm encounters a opaque type, it is ignored. Opaque types include the primitive types mentioned above, but in the user can declare any substructure in a data type to be a opaque type by providing custom instances. Doing so effectively treats values from that type as "black boxes", making SmartCheck more efficient if the user knows that some portion of the structure cannot be shrunk or is irrelevant to the property.

Opaque types can be conditional. For example, the user may want lists to be shrunk in general, unless the element of the list are opaque themselves. Such a definition is possible.

Opaque types are defined by providing the method for `opaque` in the `SubTypes` type class.

<<reference into with QC, no int shrinking>>

4.3 Reduction Algorithm Optimizations

The reduction algorithm description above omits some details and optimizations we describe here.

4.3.1 Sub-value Counterexample Hypothesis

Sometimes, a counterexample fails a property due to a sub-value nested deep inside the counterexample. The rest of the value is irrelevant. We call this the *sub-value counterexample hypothesis*. Thus, one way to efficiently search the space of potential counterexamples is to test a counterexample's (well-typed) sub-values.

For example, consider a simple calculator language containing constants, addition, and division, together with an evaluator that checks if the divisor is 0 and returning `Nothing` in that case:

```
data Exp = C Int
         | Add Exp Exp
         | Div Exp Exp

eval :: Exp -> Maybe Int
eval (C i) = Just i
eval (Add e0 e1) =
  liftM2 (+) (eval e0) (eval e1)
eval (Div e0 e1) =
  let e = eval e1 in
  if e == Just 0 then Nothing
    else liftM2 div (eval e0) e
```

Now consider the property `prop_div`, claiming that if `divSubTerms` holds on an expression, then the evaluator returns `Just` a value:

```
divSubTerms :: Exp -> Bool
divSubTerms (C _)          = True
odivSubTerms (Div _ (C 0)) = False
divSubTerms (Add e0 e1)    =  divSubTerms e0
                           && divSubTerms e1
divSubTerms (Div e0 e1)    =  divSubTerms e0
                           && divSubTerms e1

prop_div e = divSubTerms e ==> eval e /= Nothing
```

Testing `prop_div`, we might have a counterexample like the following:

```
Add (Div (C 5) (C (-12))) (Add (Add (C 2) (C 4)) (Add (C
7) (Div (Add (C 7) (C 3)) (Add (C (-5)) (C 5)))))
```

The cause is that `divSubTerms` fails to check whether the divisor evaluates to zero. In the counterexample, the culprit is a buried sub-value:

```
Div (Add (C 7) (C 3)) (Add (C (-5)) (C 5))
```

Thus, when attempting to shrink an `Exp` value, it pays to test whether a sub-value itself fails the property.

Generalizing the scenario, during the reduction algorithm's breadth-first search through a counterexample `cex`'s sub-values, we may happen upon a sub-value `cex'` that has the same type as `cex` and fails the property (while passing any preconditions). In this case, we can return `cex'` directly, and rerun the reduction algorithm on `cex'`. In Figure 4, we show an updated reduction algorithm, `reduceOpt`, that implements this optimization. The function `testHole` tests the current sub-value and if it fails the property, the function tests the sub-value directly.

4.3.2 Bounding Counterexample Exploration

SmartCheck's implementation contains flags to allow the user to customize its behavior. Three flags that are relevant to the reduction algorithm are the following:

```
reduceOpt :: forall a . SubTypes a
  => ScArgs -> (a -> Property) -> a -> IO a
reduceOpt args prop cex = reduce' 1
  where
  reduce' idx
    | Just v <- index cex idx
    = maybe (test' v idx) (reduceOpt args prop)
            (testHole v)
    | otherwise = return cex

  test' v idx = do
    vs <- newVals (getSize v) (scMaxReduce args) v
    maybe (reduce' (idx+1)) (reduceOpt args prop)
          (test cex idx vs prop)

  testHole (SubVal a) = do
    a' <- cast a :: Maybe a
    if pass prop a' then Nothing else Just a'
```

Figure 4: Reduction algorithm with the sub-value counterexample optimization.

```
scMaxReduce :: Int
scMaxSize   :: Int
scMaxDepth  :: Maybe Int
```

The `scMaxReduce` flag controls the number of values generated by the reduction algorithm for each sub-value analyzed. `scMaxSize` controls the maximum size of values generated to replace sub-values by the reduction algorithm. Thus, new sub-values must be strictly smaller than the minimum of `scMaxSize` and the size of the sub-value being replaced. Finally, `scMaxDepth` determines the maximum depth in the counterexample the reduction algorithm should analyze. A value of `Nothing` means that the counterexample should be exhaustively reduced, as the algorithm is presented in Figure 3. For example, the depth of a list is determined by its length, and the depth of the binary tree defined in Section 4.1 is determined by the function `depth`:

```
depth L         = 0
depth (B t0 t1) = 1 + max (depth t0) (depth t1)
```

Of the flags, `scMaxDepth` is the most important for controlling efficiency, particularly for large product types with significant "fan out". The number of sub-values of a product type value can grow exponentially with respect to the depth. Furthermore, note that as the reduction algorithm descends further, there is less chance to reduce the size of the value overall, since smaller and smaller sub-values are replaced.

5. Counterexample Generalization

Small counterexamples make debugging easier, but they are just half the battle. To go from a specific counterexample to the required fix in a program, the programmer must have a flash of insight in which she generalizes the counterexample to a set of counterexamples for which the program and property fails. The generalization step is an important yet under-appreciated step in the debugging process. A characterizing formula reduces the noise in favor of the signal by abstracting away portions of large counterexample that are irrelevant to why it violates the property.

The characterization of counterexamples that most helps the programmer should strike a middle ground. A single counterexample is too specific. On the other hand, the property itself is a formula that over-approximates the failing inputs. In this section, we describe two kinds of formula that fall between these two extremes that we call universal and existential sub-value generalization, respectively. We then describe a third approach to generalization to

```
subTrees :: SubTypes a => a -> Idx -> [Idx] -> Bool
subTrees cex idx = any (subTree cex idx)

extrapolate :: SubTypes a
  => ScArgs -> a -> (a -> Property) -> IO [Idx]
extrapolate args cex prop = extrapolate' 1 []
  where
  extrapolate' idx idxs
    | subTrees cex idx idxs
    = extrapolate' (idx+1) idxs
    | Just v <- index cex idx = mkNewVals v
    | otherwise = return idxs
    where
    mkNewVals v = do
      vs <- newVals (scMaxSize args)
                    (scMaxForall args) v
      extrapolate' (idx+1)
        (if allFail args cex idx vs prop
            then idx:idxs else idxs)

allFail :: SubTypes a => ScArgs -> a -> Idx
  -> [SubVal] -> (a -> Property) -> Bool
allFail args cex idx vs prop =
  length res >= scMinForall args && and res
  where
  res  = mapMaybe go vs
  go   = fail prop . replace cex idx
```

Figure 5: Universal sub-value generation algorithm.

automatically strengthen a property's precondition to obtain new counterexamples.

5.1 Universal Sub-Value Generalization

Consider again the calculator language from Section 4.3.1. The property `prop_div` is violated for any numerator, so we might generalize a counterexample like

```
Div (Add (C 7) (C 3)) (Add (C (-5)) (C 5))
```

by the formula

```
forall x . Div x (Add (C (-5)) (C 5))
```

since any dividend results in divide-by-zero for the given divisor. Not only do the generalizations assist the programmer's insight, but they reduce the sheer size of the counterexample. We call the kind of formula just shown *universal sub-value generalization* and it is implemented in SmartCheck.

An extrapolation algorithm performs universal sub-value generalization. The basic idea is as follows: for a counterexample `cex` and a property `prop`, a breadth-first search over the sub-values of the `cex` is performed. For each sub-value, the algorithm generates new sub-values and replaces them in `cex` to create a list of new potential counterexamples. If no new value satisfies the property, then we extrapolate, claiming that for *any* new value replacing that sub-value in `cex`, the property will fail.

The extrapolation algorithm is shown in Figure 5; let sketch its specification. The algorithm is similar to the reduction algorithm in Figure 3 (and in the implementation, the algorithms are generalized and combined). The function `extrapolate` returns a list of indexes to be generalized in the original counterexample. In the recursive function `extrapolate'`, there is a function guard with a call

```
subTree cex idx0 idx1
```

where `subTree` has the type

```
subTree :: SubTypes a => a -> Idx -> Idx -> Bool
```

The value

```
subTree cex idx0 idx1
```

is true if in `cex`, the value at index `idx0` is a child of index `idx1` in a tree representation of `cex` (i.e., `subVals cex`). The `subTrees` guard prevents the algorithm from trying to generalize sub-values that are abstracted away already since their parents have been generalized. New sub-values are generated by `newVals`, shown in Figure 3.

The function `allFail` takes a counterexample `cex`, an index into `cex`, a list of new sub-values, and a property. It returns true if no new values satisfy the property. The function

```
fail :: (a -> Property) -> a -> Maybe Bool
```

is roughly the dual of `pass` in the reduction algorithm: (`fail prop cex`) returns (`Just True`) if `cex` passes `prop`'s precondition but fails the property; (`Just False`) if `cex` non-trivially satisfies `prop`, and `Nothing` if `cex` fails `prop`'s precondition.

Like in the reduction algorithm user-specified flags bound the behavior of the algorithm. We bound the size of values to generate by the flag `scMaxSize`, which is independent of the size of the particular sub-value. The flag `scMaxForall` is the analogue of the `scMaxReduce` flag, determining the number of values generated in trying to generalize a value. The flag `scMinForall` is the minimum number of `Just False` results required from `fail` to extrapolate from failed tests to a universal claim. So, for example, if `scMaxForall` is set to 30 and `scMinForall` is set to 20, we generate 30 new values, 20 of which must pass the precondition but fail the property to claim the counterexample can be generalized.

The algorithm's complexity is $\mathcal{O}(n)$, where n is the number of constructors in the counterexample. Again, we assume that the cost for each generating random values and testing them at each index is constant.

Soundness The extrapolation algorithm is unsound in two ways. First, it extrapolates from a set of counterexamples to a universal claim, similar to QuickSpec or Daikon [5, 7]. By tuning the parameters, the risk of an unsound generalization is reduced by requiring more or larger values to fail the property.

In some cases, a formula may be returned that is overly general. For example, consider the counterexample in which both arguments of the outermost `Add` constructor contain values causing the failure:

```
Add (Div (C 1) (Add (C (-2)) (C 2)))
    (Div (C 0) (Add (C (-1)) (C 1)))
```

Since no matter what random value the first field of `Add` is replaced with, the property fails by the second field and vice versa for the second. The universal generalization algorithm might return the formula

```
forall values x0 x1 . Add x0 x1
```

An over-generalized formula is only a problem when generalizing without minimizing the counterexample first, as pointed out to the author by John Hughes, so in practice with SmartCheck, this form of unsoundness does not arise. However, if one were to generalize without shrinking first, the reader should read a universally quantified formula as shorthand for quantifying each variable independently and taking the conjunction of formulas. For example, instead of

```
forall values x0 x1 . Add x0 x1
```

one should read

```
      forall values x0 .
        Add x0 (Div (C 0) (C (-1)) (C 1))
  and forall values x1 .
        Add (Div (C 1) (C (-2)) (C 2)) x1
```

5.2 Existential Sub-Value Generalization

Sum types denote choice in a data type. Sometimes, a property over a sum type fails because there is a bug for some of the variants but not others. For example, recall again the calculator language from Section 4.3.1. The no-division-by-zero property fails only for values that contain a variant tagged with the Div constructor. Recall again the generalized counterexample from Section 5:

```
forall x . Div x (Add (C (-5)) (C 5))
```

Because the divisor does not generalize, we know there is something special about it that causes failure. But we might wonder if there is something special about variants tagged by the Add constructor, or might we finding failing sub-values with the other variants.

We therefore introduce another kind of generalization we call *existential sub-value generalization*. In this generalization, if there is a counterexample containing every possible variant as a sub-value, then we abstract it. For example, suppose that divSubTerms had no equation

```
divSubTerms (Div _ (C 0)) = False
```

Then the following would all counterexamples:

```
Div (Add (C 3) (C 2)) (C 0)
Div (C 7) (Add (C (-5)) (C 5))
Div (C (-2)) (Div (C 0) (C 1))
```

Because there are only three variants in the type, we have found counterexamples built from each of them in the divisor. We therefore can claim the following formula holds:

forall values x .
  forall constructors c .
    there exist arguments $\vec{y}$.
      such that Div x (c $\vec{y}$)

We therefore present an existential sub-value generalization algorithm that performs constructor generalization. Like with the other algorithms, this algorithm also performs a breadth-first search over a counterexample.

We show the algorithm in Figure 6. The function sumTest takes a set of flags, a counterexample, a property, and a list of indexes that have already been generalized—perhaps by the extrapolation algorithm in Figure 5. The list of course may be empty if no sub-values have been previously extrapolated. In a call to subTrees, discussed in Section 5.1, the guard prevents constructor generalization if the current index is a sub-value of a previously generalized value. Otherwise, a list of well-typed new values are generated by a call to newVals, as shown in Figure 3. In the arguments to newVals, we bound the size of values generated with scMaxSize as before, and bound the number of values generated with the flag scMaxExists. Because values are randomly generated, for "wide" sum-types (i.e., with a large number of constructors), scMaxExists should be large enough to ensure with high probability that each variant is generated.

The function constrFail returns true if we replace the sub-value at index idx in counterexample cex with every possible variant given the type and construct a counterexample to the property. There are four guards to the recursive function constrFail': the first guard holds if the list of constructors tagging variants in which a counterexample is found is equal in size to the list of all possible constructors for the type. The second guard tests whether the set of test values is null; if so (and if the first guard fails), then we have exhausted test values before finding all possible failing variants. Third, for a specific sub-value v, we test whether it fails the property. If so, we add its constructor to the list of constructors. Otherwise, we simply recurse. Note in the definition of prop', we add an additional precondition that the current constructor is not an element of constructors already seen. Thus, (go v) returns

```
Just True
```

if

```
replace cex idx v
```

passes this precondition (and any other preconditions of prop), but fails the property.

Unlike universal sub-value generalization, existential sub-value generalization is sound. The existential claim is only that for each variant, there exists at least one counterexample, so inductive reasoning is not required.

This algorithm's complexity is also $\mathcal{O}(n)$, where n is the number of constructors in the counterexample.

```
subConstr :: SubVal -> String
subConstr (SubVal a) = constr a

subConstrs :: SubVal -> [String]
subConstrs (SubVal a) = constrs a

sumTest :: SubTypes a => ScArgs -> a
  -> (a -> Property) -> [Idx] -> IO [Idx]
sumTest args cex prop exIdxs = sumTest' 1 []
  where
  sumTest' idx idxs
    | subTrees cex idx (exIdxs ++ idxs)
    = sumTest' (idx+1) idxs
    | Just v <- index cex idx = fromSumTest v
    | otherwise = return idxs
    where
    fromSumTest v = do
      vs <- newVals (scMaxSize args)
              (scMaxExists args) v
      sumTest' (idx+1)
        (if constrFail cex idx vs prop
           (subConstr v) (subConstrs v)
           then idx:idxs else idxs)

constrFail :: SubTypes a => a -> Idx -> [SubVal]
  -> (a -> Property) -> String -> [String] -> Bool
constrFail cex idx vs prop con allCons =
  constrFail' [con] vs
  where
  constrFail' cons vs'
    | length cons == length allCons = True
    | null vs'                      = False
    | go v == Just True
    = constrFail' (c:cons) (tail vs')
    | otherwise
    = constrFail' cons (tail vs')
    where
    v  = head vs'
    c  = subConstr v
    go = fail prop' . replace cex idx
    prop' a = c `notElem` cons ==> prop a
```

Figure 6: Universal sub-value generation algorithm.

5.3 Automated Precondition Strengthening

The universal and existential generalization algorithms generalize a counterexample, but in the "neighborhood" of the original counterexample. In particular, all generalizations are from the same variant as the original counterexample. To help the programmer in the generalization step, we would also like a way to test the property again, ensuring we get counterexamples (if they exist) outside of the neighborhood of the original one.

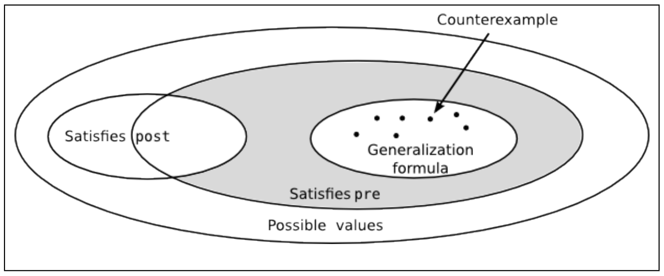

Figure 7: Counterexample generalization.

```
matchesShapes :: SubTypes a
  => a -> [(a,[Idx])] -> Bool
matchesShapes d = any (matchesShape d)

matchesShape :: SubTypes a
  => a -> (a, [Idx]) -> Bool
matchesShape a (b, idxs)
  | constr a /= constr b = False
  | Just a' <- aRepl
  = let x = subForest (subVals a') in
    let y = subForest (subVals b)  in
    all foldEqConstrs (zip x y)
  | otherwise = False
  where
  updateA idx d =
    fmap (replace d idx) (index b idx)
  aRepl = foldl go (Just a) idxs where
    go ma idx | Just x <- ma = updateA idx x
              | otherwise    = Nothing
  foldEqConstrs ( Node (SubVal l0) sts0
                , Node (SubVal l1) sts1 )
    | constr l0 == constr l1 =
      all foldEqConstrs (zip sts0 sts1)
    | otherwise              = False
```

Figure 8: Shape matching algorithm.

Figure 7 illustrates a property of the form (`pre ==> post`). Points are specific counterexamples that satisfy the precondition but fail the post-condition, and the enclosing oval represents the generalization of counterexamples resulting from either universal or existential generalization. Our goal is to find additional counterexamples in the shaded region. As new counterexamples are discovered in the shaded region (and generalized), the counterexample space becomes covered until no more classes of counterexamples exist or it becomes too difficult for the testing framework to discover them.

Figure 8 shows the shape-matching algorithm used for precondition strengthening. The function takes a candidate counterexample and a list of previous counterexamples, together with the indexes at which they have been generalized. The basic idea of the algorithm is to determine whether two values have the same "shape". Intuitively, we consider two values to have the same shape if in their respective tree representations, their constructors at each node in the tree match, ignoring all opaque types (Section 4.2.2). That is, for values `a` and `b`,

```
subVals a == subVals b
```

Furthermore, indexes that have been universally generalized (and all their children) match any value, since a universally generalized index indicates that counterexamples have been found for any possible constructor. In our implementation, we also consider existentially generalized indexes to match any value. Doing so is a design choice that is more aggressive about covering the space of counterexamples at the risk of omitting some.

To understand the shape matching algorithm, consider a few values of type `Exp`, defined in Section 4.3.1:

```
e0 = Div (C 1) (C 2)
e1 = Div (C 1) (C 3)
e2 = Div (Add (C 1) (C 2)) (C 7)
e3 = Div (Div (C 8) (C 2)) (C 7)
```

Then the following hold:

```
matchesShape e0 (e1, [])  == True
matchesShape e1 (e2, [])  == False
matchesShape e1 (e2, [0]) == True
matchesShape e3 (e2, [0]) == True
```

The first equation holds because we ignore opaque types, so the arguments to `C` are considered equal. The second equation fails because at index 0, `e1` contains the constructor `C` and `e2` contains the constructor `Add`. However, index 0 has been generalized, then we ignore the sub-value at index 0, and they match (the third equation). The same reasoning holds for the fourth equation.

6. Implementation and Usage

The implementation of SmartCheck is written in Haskell and is designed to test Haskell programs. The source code is licensed BSD3 and is freely available.[4]

SmartCheck generically operates over arbitrary algebraic data and so uses a generics library to encode generic traversals. Specifically, SmartCheck uses "GHC Generics", a recently-developed generics library for Haskell [15]. The generics library allows algebraic data types to automatically derive the type class `SubTypes` presented in Section 3. One limitation with GHC Generics is that it does not (currently) support generalized algebraic data types [12].

The data type to be tested must derive the `Typeable`, and `Generic` type classes. Deriving `Typeable` and `Generic` require using the GHC language extensions `DeriveDataTypeable` and `DeriveGeneric`, respectively. `Typeable` is used for dynamic typing in defining the `replace` method of `SubTypes` since it is unknown at compile-time whether the value to be replaced and replacing value have the same types. However, through SmartCheck's interface, run-time failures due to type-mismatches will not occur. Additionally, like with QuickCheck, deriving `Show` is required to print counterexamples discovered.

Then, the user simply declares, for a data type `D`,

```
instance Subtypes D
```

Default instances are provided for common Prelude types and some additional ones, including all types for which QuickCheck provides instances.

SmartCheck does not implement a counterexample discovery algorithm or an `arbitrary` method. Any initial counterexample can be passed in or generated, and the algorithm depends on QuickCheck's `arbitrary` to generate new well-typed values.

The kinds of programs SmartCheck is specialized for are ones that operate over a large data structure together with smaller inputs. Therefore, properties provided to SmartCheck are expected to be of the form

```
Testable prop => a -> prop
```

where `a` is the type of the value for SmartCheck to analyze, and `prop` is a testable property, as defined by QuickCheck; morally, these are functions (or degenerately, values) that evaluate to a Boolean value.

[4] `https://github.com/leepike/SmartCheck.git`

If QuickCheck is used to discover a counterexample, all arguments except the first are shrunk, if their types have `shrink` methods defined for them. The first argument is returned to SmartCheck to be shrunk or generalized according to the algorithms described earlier.

A read-eval-print loop is presented to the user, allowing her to iterate shrink and generalize counterexamples, and then generate new counterexamples after strengthening the property's precondition as described in Figure 5.3.

SmartCheck is executed using

```
> smartCheck args prop
```

where `args` (the arguments) are passed in, and `prop` is the property being tested.

The interface types and functions for SmartCheck with analogous behavior to QuickCheck's are prefixed with an `sc` to avoid name space collisions with QuickCheck. Others are specialized for SmartCheck; e.g., enabling or disabling universal or existential extrapolated, number of extrapolation rounds, and limits on the depth and size of the values to generate.

Counterexamples can be optionally shown in a tree format by setting the `format` field of the arguments to be equal to `PrintTree`. for example, the tree format shows a counterexample like

```
Div (C 1) (Add (C 0) (C 2))
```

as

```
Div
|
+- C 1
|
`- Add
   |
   +- C 0
   |
   `- C 2
```

We find that for very large and nested data structures, a tree representation aids in visually parsing the value.

7. Experiments

We describe two experiments using SmartCheck, including an XMonad property and a property about a natural language processing library.

7.1 XMonad

Recall from the introduction the XMonad example. The XMonad window manager is a large software project with many contributors, so naturally, a QuickCheck test harness is included to help ensure new commits do not introduce bugs. At the heart of XMonad is a `StackSet` data type that encodes the relationship between windows, work spaces, and which window has the focus. XMonad contains properties to ensure the correct manipulation of `StackSets`. Due to having one large data-structure that is essential to the entire program, XMonad is a perfect candidate for SmartCheck.

XMonad passes all of its QuickCheck tests, but let us see what might happen to a new contributor if things go awry. Suppose a developer defines a deletion function to delete a window, if it exists. There is a deletion function that exists in XMonad, which is quite complex, given the amount of state that is managed by `StackSet`. However, one function used in deletion is to filter the stack of windows associated with each workspace defined:

```
removeFromWorkspace ws =
  ws { stack = stack ws >>= filter (/= w) }
```

Now, suppose the programmer makes a simple typo and instead writes

```
removeFromWorkspace ws =
  ws { stack = stack ws >>= filter (== w) }
```

When testing the property `prop_delete`, which says that deleting the focused window of the current stack removes it from the `StackSet` `x`.

```
prop_delete x =
    case peek x of
        Nothing -> True
        Just i  -> not (member i (delete i x))
```

QuickCheck returns the large value shown in the introduction. That value is a relatively small counterexample, but even the smallest `StackSet` values are somewhat visually overwhelming due to the number of fields within it. Recall the value returned by SmartCheck after generalization:

```
forall values x0 x1 x2 x3:
  StackSet
    (Screen (Workspace x0 (-1) (Just x1)) 1 1)
    x2 x3 (fromList [])
```

Let us examine what was generalized. In our test run, we chose to treat data maps as opaque, so the forth element of `StackSet` is not generalized, but is simply the empty map, so looks uninteresting. The second and third fields of `StackSet` are generalized, but the first one is not. So there is something particular about it. So the culprit is one of the small constants (1 and -1) or having a `Just` value rather than a `Nothing`: it turns out that what matters is having a `Just` value, which is the stack field that deletion works on!

7.2 Natural Language Processing

In 2012, a question was posted on the programming message board Stack Overflow asking about how to shrink large nested data types.[5] The poster writes:

> ... I tend to get an incomprehensible page full of output. ... Implementing the shrink function for each of my types seems to help a little, but not as much as I'd like. ... If I try to tune my shrink implementations, I also find that QC starts taking a very long time.

<<how long unanswered?>>

The question relates to the Geni natural language processing (NLP) package implemented in Haskell [13]. Specifically, counterexamples to a property attempting to show that a macro expansion function is its own inverse are enormous, requiring 200-300 80-character lines to print.

Using SmartCheck, we are able to reduce counterexamples to around 25 80-character lines of output. Most of the savings in the counterexample size were due to universal generalization, like in the XMonad case: entire record fields are abstracted away. From that, we (syntactically) shrunk the counterexample by-hand further by naming common sub-expressions.

We were able to send a substantially reduced and generalized counterexample to the message poster, making the cause of the bug more obvious. The author responded:

> ...While your improved shrinking may not have gone 'all' the way to the bottom, it got me a *huge* chunk of the way there!

Recall that through the entire process, we never had to learn how GenI works, what the property meant, or how to write a custom shrink method!

[5] `http://stackoverflow.com/questions/8788542/how-do-i-get-good-small-shrinks-out-of-quickcheck`

"Retro-fitting" SmartCheck to GenI, which is a large software system, required some boilerplate work. Fortunately, because the program already uses QuickCheck, `arbitrary` instances were already defined, but we had to hunt down the data types used to write `instance Subtypes D`, for each data type `D` (indeed, the job could have largely been done with Sed and Awk).

7.3 Benchmarks

No set of testing benchmarks exists over which to compare different test-case generation and minimization approaches. Therefore, we have collected a small number of benchmarks, in addition to the more involved case-studies described earlier in this section.

The benchmarks presented, in addition to the motivating example presented in Section 2, are

- *Reverse*, with the false property

  ```
  prop_rev :: [a] -> Bool
  prop_rev ls = ls == reverse ls
  ```

  (the example appears in the original QuickCheck documentation);
- *Div0*, a division-by-zero property for a simple calculator language (introduced in Section 4.3.1);
- *Heap*, an example from the QuickCheck test suite, in which an incorrect "to sorted list" function is checked.

All benchmarks can be found at `https://github.com/leepike/SmartCheck/tree/master/regression`. Benchmarks are against QuickCheck 2.7. The benchmarks only compare the size of shrunk values and the time required to generate them using standard arguments to QuickCheck and SmartCheck. We do not benchmark the generalization algorithms, since there is no analog in QuickCheck or other testing frameworks.

<<include std dev.>> <<fill in benchmarks!>>

		QuickCheck	SmartCheck
Reverse	Mean	0.08s, 69	0.53s, 35
	Median	0.07s, 69	0.17s, 33
	Std. dev.		
	95%	0.14s, 100	1.76s, 65
Div0	Mean	0.08s, 69	0.53s, 35
	Median	0.07s, 69	0.17s, 33
	Std. dev.		
	95%	0.14s, 100	1.76s, 65
Heap	Mean	0.08s, 69	0.53s, 35
	Median	0.07s, 69	0.17s, 33
	Std. dev.		
	95%	0.14s, 100	1.76s, 65

Table 2. Summarizing data for the graphs in Figure 2. Entries contain execution time (in seconds) and counterexample sizes (counting constructors).

The *Reverse* benchmark essentially provides a lower-bound on the benefit of using SmartCheck to discover small counterexamples, since a list of length two falsifies the property (SmartCheck is still useful in generalizing the counterexample, however!). Perhaps surprisingly, SmartCheck still slightly outperforms QuickCheck. The other two benchmarks are slightly larger examples, where SmartCheck's benefits are more pronounced. But again, we emphasize the most pronounced benefit comes in much larger examples, like described in Section 7.2.

8. Related Work

Other researchers have also investigated counterexample shrinking. Zeller and Hildebrandt describe an application of greedy search to shrink counterexamples they call "delta-debugging" (DD) [19]. The authors apply their work to shrinking HTML inputs to crash Mozilla and shrinking C programs to trigger a bug in GCC. Subsequent generalizations are reported by Misherghi and Su in which they perform greedy search on tree-structured data; they call their approach hierarchical delta-debugging (HDD) [16].

HDD is most similar to SmartCheck's reduction algorithm, with an important difference: HDD (and DD) is deterministic, so the algorithm only succeeds in reducing the counterexample only if a new counterexample can be constructed from the original one. Our approach combines the speed of delta debugging with the power of QuickCheck to *randomly* discover structurally smaller counterexamples; in short, this is the first treatment we know of that integrates counterexample generation with counterexample reduction. Furthermore, neither paper explores the idea of counterexample generalization.

Within the functional programming community, one of the few treatments of generic shrinking is as a motivation for generic programming in Haskell's "Scrap your boilerplate" generic programming library [14]. There, the motivation was not to design new approaches to counterexample reduction, but simply to derive instances for the `shrink` method.

Besides QuickCheck and SmallCheck, another testing framework related to SmartCheck is the recent Haskell library Feat [6]. Feat provides automated enumerations of algebraic data types in Haskell, allowing for fast access to very large indexes. For example, from the enumeration of `([Bool])`

```
[[],[False],[True],[False,False],[False,True] ...
```

Accessing the 10^{1000}th element takes under 0.1 seconds in interpreted Haskell. Feat combines some advantages of SmallCheck and QuickCheck, since the user can choose to exhaustively test an enumeration up to some depth, like with SmallCheck, or she can create a uniform distribution of test cases up to some depth.

Feat finds small counterexamples effectively, but a limitation compared to QuickCheck is that value generation is applicative rather than monadic. Consequently, defining custom generators (i.e., instances for `arbitrary`) is constrained. On the other hand, generators can be derived automatically with Feat.

We have benchmarked Feat as well on the example in the introduction (Figure 1). It performs quite competitively to SmartCheck in terms the size of counterexamples discovered: the mean and median number of `N` constructors in the counterexamples returned by Feat are both seven. <<check>> However, Feat performs less well with respect to how long counterexample discovery takes: the mean time taken by Feat is 5.60 seconds, with a long tail: five percent of runs take over 14 seconds (the mean time taken by SmartCheck is 0.12 seconds, and 99% of runs take less than 0.30 seconds).

Finally, SmartCheck bears some similarity to QuickSpec, a testing-based library that infers equational properties about programs [5] insofar as they both attempt to generalize counterexamples based on specific inputs. QuickSpec attempts to infer equational properties of programs through random testing. Similarly, Daikon infers assertions for C, C++, Java, and Perl by observing relationships between variables in executions of a program [7]. SmartCheck does not attempt to infer properties like these tools do, so it is in one sense less general. However, SmartCheck's three kinds of counterexample generalization are novel.

9. Conclusions and Future Work

We have presented new approaches for generically shrinking and generalizing counterexamples over algebraic data. SmartCheck automates the laborious task of shrinking, and extrapolating from counterexamples, and in our experience, performs better and faster than hand-written shrink functions.

We envision a number of potential extensions and improvements to SmartCheck. First, we have considered only the simplest kind of data, algebraic data types. As noted in Section 6, SmartCheck does not work with GADTs currently, due to limitations with GHC Generics. It would be interesting to see if the approaches described here could be extended to function types as well—we are particularly motivated by Claessen's recent work in shrinking and showing functions [2].

Lazy SmallCheck can test partially-defined inputs by detecting the evaluation of undefined values [17]. This capability is useful in shrinking, too. For example, the universal sub-value generalization algorithm (Section 5.1) could be extended to shortcut testing and generalize a sub-value if it is not evaluated in testing the property. Not only does this shortcut the generalization phase, but it gives a *proof* that the sub-value can be generalized.

SmartCheck displays (generalized) counterexamples in a form similar to default `Show` instances or in a tree form, which can be helpful to parse the components of the value. Better approaches for showing large data types are needed. In particular, an interactive web-based viewer with hyperlinks to close or expand sub-values would be particularly useful.

Another aspect of displaying large counterexamples that we have not explored is to exploit sharing. Constructs might be repeated that can be abstracted out. For example, instead of a counterexample like

```
Add (Div (C 1) (Add (C (-2)) (C 2)))
    (Div (C 1) (Add (C (-1)) (C 1)))
```

we might instead return

```
Add (div (-2) 2) (div (-1) 1)
  where div x y = Div (C 1) (Add (C x) (C y))
```

Discovering and exploiting sharing automatically is future work.

Debugging is a difficult task. Functional programming has been at the forefront of testing research, with tools like QuickCheck and SmallCheck. We were motivated to build a tool like SmartCheck just because of how effective QuickCheck is at discovering counterexamples automatically—there would be no such problem of having very large counterexamples if inputs were written by hand. We hope SmartCheck and the ideas in this paper continue the tradition of highly-automated testing and debugging in the functional programming community, and beyond!

<<mention memoization (generalized tries)>>

<<tail-recursive algorithm is fast and easy to port to imperative languages>>

Acknowledgments

I thank Andy Gill, Joe Hurd, John Launchbury, Simon Winwood, and especially John Hughes for reading an earlier draft of this paper. Their deep insights and bug catches dramatically improved both the form and content. This paper was and rejected from ICFP'13; the anonymous reviewers' suggestions improved the presentation additional benchmarking.

References

[1] T. Arts, J. Hughes, J. Johansson, and U. T. Wiger. Testing telecoms software with quviq quickcheck. In *ACM SIGPLAN Workshop on Erlang Erlang Workshop*, pages 2–10. ACM, 2006.

[2] K. Claessen. Shrinking and showing functions: (functional pearl). In *Proceedings of the Haskell symposium*, pages 73–80. ACM, 2012.

[3] K. Claessen and J. Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In *ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 268–279. ACM, 2000.

[4] K. Claessen and J. Hughes. Testing monadic code with QuickCheck. In *ACM SIGPLAN workshop on Haskell*, pages 65–77, 2002.

[5] K. Claessen, N. Smallbone, and J. Hughes. QuickSpec: Guessing formal specifications using testing. In *Tests and Proofs Intl. Conference (TAP)*, LNCS, pages 6–21, 2010.

[6] J. Duregård, P. Jansson, and M. Wang. Feat: functional enumeration of algebraic types. In *Proceedings of the 5th ACM SIGPLAN Symposium on Haskell*, pages 61–72. ACM, 2012.

[7] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programing*, 69(1-3):35–45, Dec. 2007.

[8] haskell. Haskell 2010 language report. Available at `http://www.haskell.org/definition/haskell2010.pdf`, July 2010.

[9] G. P. Huet. The zipper. *Journal of Functional Programming*, 7(5): 549–554, 1997.

[10] J. Hughes. Software testing with quickcheck. In *Central European Functional Programming School (CEFP)*, volume 6299 of *LNCS*, pages 183–223. Springer, 2010.

[11] D. Jackson. *Software abstractions: logic, language and analysis*. MIT Press, 2006.

[12] P. Johann and N. Ghani. Foundations for structured programming with GADTs. In *Symposium on Principles of programming Languages (POPL)*, pages 297–308. ACM, 2008.

[13] E. Kow. GenI: natural language generation in Haskell. In *Proceedings of the 2006 ACM SIGPLAN workshop on Haskell*, pages 110–119. ACM, 2006.

[14] R. Lämmel and S. L. P. Jones. Scrap your boilerplate with class: extensible generic functions. In *ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 204–215. ACM, 2005.

[15] J. P. Magalhães, A. Dijkstra, J. Jeuring, and A. Löh. A generic deriving mechanism for Haskell. In *Proceedings of the 3rd ACM Haskell Symposium on Haskell*, pages 37–48. ACM, 2010.

[16] G. Misherghi and Z. Su. HDD: hierarchical delta debugging. In *Proceedings of the 28th international conference on Software engineering*, pages 142–151. ACM, 2006.

[17] C. Runciman, M. Naylor, and F. Lindblad. Smallcheck and lazy smallcheck: automatic exhaustive testing for small values. In *Proceedings of the ACM Haskell Symposium on Haskell*, pages 37–48. ACM, 2008.

[18] D. Stewart and S. Sjanssen. XMonad. In *ACM SIGPLAN Workshop on Haskell*, page 119. ACM, 2007.

[19] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2): 183–200, Feb. 2002.