SmartCheck: Automatic and Efficient Counterexample Reduction and Generalization

Lee Pike
Galois, Inc.
leepike@galois.com

Abstract

QuickCheck 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.

Categories and Subject Descriptors D.2.5 [Software Engineering]: Testing and Debugging

Keywords property-based testing; test-case generalization; delta-debugging

1. Introduction

The QuickCheck testing framework was a revolutionary step-forward in property-based 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.

That is, automated testing helps find a counterexample, but it may be a very large counterexample. We are left facing the shrinking problem: consider program \( P \) modeled as a function mapping an input vector \( i \) to an output value. Consider property \( \phi \) modeled as a function mapping the inputs of \( P \) to a Boolean value that has the form \( \text{pre}(i) \implies \text{post}(P(i)) \), where \( \text{pre} \) is a precondition on inputs and \( \text{post} \) is a postcondition on the program output. Let \( \prec \) be a partial order over the set of possible inputs. Given inputs \( i \) such that \( \phi(i) \) is false, the shrinking problem is to find inputs \( j \) such that \( \phi(j) \) is false and there exist no inputs \( k \) such that \( k \prec j \) and \( \phi(k) \) is false.

In general, the problem is too expensive to solve optimally, since it requires showing that for a set of failing inputs, there exist no strictly smaller inputs that also cause the property to fail. Such a demonstration requires exhaustive testing or proof. Solutions to the shrinking problem are therefore approximated to efficiently discover sufficiently small inputs causing the input to fail. Usually, the shrunk inputs are derived algorithmically (but perhaps non-deterministically) from known larger inputs resulting in failure.

Originally, QuickCheck defines a type class \texttt{Arbitrary} that presents a method \texttt{arbitrary} for generating random values of a given type. Gill added another method to the type class:

\begin{verbatim}
smaller :: a -> [a]
\end{verbatim}

The purpose of \texttt{smaller} is to generate strictly smaller values, according to some metric, from a given counterexample. These new values are then tested to attempt to find a smaller counterexample. Today, \texttt{smaller} is called \texttt{shrink}.

In industrial uses, shrinking is essential. In describing commercial applications of QuickCheck, Hughes has noted that “without it [shrinkaging], randomly generated failing cases would often be so large as to be almost useless.” [9]. Hughes \textit{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 characterizing the bug. This leap of abstraction is often implicitly made by the programmer in determining the reason why counterexamples fail the property. For example, Figure 1 contains a relatively small counterexample returned when using QuickCheck to test a property in (a bug-injected version of) \texttt{XMONAD}, a popular X11 window manager written in Haskell [19]. (This counterexample uses Haskell’s default \texttt{Show} instances, which uses record syn-
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. After presenting some preliminary definitions in Section 3, in Section 4, we describe SmartCheck’s generic counterexample reduction algorithm.

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 the property. For example, finding counterexamples (Left: 2) and (Right: True) for the type

```haskell
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 a counterexample structurally with respect to the property, so that the user can focus on the heart of the problem in a class of counterexamples. Given a program and large counterexample, SmartCheck returns such a formula.

SmartCheck takes a counterexample produced by some oracle and generically minimizes and generalizes the counterexample. After presenting some preliminary definitions in Section 3, in Section 4, we describe SmartCheck’s generic counterexample reduction algorithm.

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

```haskell
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, such that each bug in isolation is sufficient to cause failure. During testing, it is useful to discover counterexamples arising from all the bugs in one go. In practice, the problem is solved by discovering a counterexample cex, abstracting it, and then adding a new pre-condition 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

```haskell
data T = T I I I I I

toList :: T -> [Int16]
tolist (T i0 i1 i2 i3 i4) = a <- shrink i0, b <- shrink i1, c <- shrink i2, d <- shrink i3, e <- shrink i4
```

Figure 2: Example program and property.

In this section, we motivate in more detail the challenges in shrinking counterexamples by comparing manual approaches using QuickCheck to SmartCheck. We focus on shrinking rather than generalization here since counterexample generalization is unique to SmartCheck. We will show how a small data type and simple property can result in large counterexamples without any shrinking. Then we show the difficulty in designing an efficient shrink implementation. We will show a poor design before arriving at a “canonical” manual solution.

Consider the example in Figure 2.1 Data type `T` is a product type containing five lists of signed 16-bit integers.

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. 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, since \((-20000 + -20000) \mod ((2^{15}) = 25536\), and 25536 < 5 * 256, the value

```haskell
T [-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 average counterexample discovered contains just over 80 `Int16` values, and over five percent contain over 100 values. 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 `shrink` values over the arguments to the constructor `T`. This can be expressed using Haskell’s list-comprehension notation:

```haskell
shrink (T i0 i1 i2 i3 i4) = [a <- shrink i0, b <- shrink i1, c <- shrink i2, d <- shrink i3, e <- shrink i4]
```

While the above definition appears reasonable on first glance, 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

1 All examples and algorithms in this paper are presented in Haskell 2010 [15] plus the language extensions of existential type quantification and scoped type variables.
be shrunk. More troublesome is that with QuickCheck’s default arbitrary instances, the length of potential counterexamples returned by shrink can exceed $10^{10}$, which is an intractable number of tests for QuickCheck to analyze. The reason for the blowup is that shrinkage a [a] produces a list [[a]], a list of lists; each list is a new shrinking of the original list. Then, we take the cross-product of the generated lists. For the example above, with some counterexamples, the shrinking stage may appear to execute for hours, consuming ever more memory, without returning a result.

The first way we will try to contain the search space is by preventing the shrink method from operating on numeric types. The standard approach for doing so is by defining a newtype wrapper:

```haskell
newtype J = J { getInt :: Int16 }
```

Next, a programmer might try to control the complexity by truncating lists using

```
take n ls
```
returns the first $n$ elements of $ls$. The trade-off is quicker shrinking with a lower probability of finding a smaller counterexample. For example, we might redefine shrink as follows:

```haskell
shrink (T i0 i1 i2 i3 i4) = [ T a b c d e | a <- take 10 (shrink i0),
, b <- take 11 c, c <- take 12 d,
, d <- take 13 e, e <- take 14 ]
```

Call this version “truncated shrink”. Truncation controls the blowup of the input-space, but the downside is that potentially smaller counterexamples may be omitted.

Someone who understands the semantics of QuickCheck’s shrink implementation defines a shrink instance as follows:

```
shrink (T i0 i1 i2 i3 i4) = map go xs
where
xs = shrink (i0, i1, i2, i3, i4)

    go (10', 11', 12', 13', 14')
  = T 10' 11' 12' 13' 14'
```

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 from pairs to arbitrary tuples). Thus, each argument to T is independently shrunk.

Finally, with this combination of shrinking tuples and disabling shrinking for Int16 values, we have an efficient and effective shrinking approach.

In early 2014, QuickCheck added the ability to derive shrink instances for algebraic data types. (The first release of the SmartCheck software was mid-2012.) With this new feature, the user declares

```
shrink = genericShrink
```
to obtain efficient shrink instances.

The results of our discussion above are summarized in Table 1 and the corresponding graph in Figure 3. First, we show the results of no shrinking (labeled ’none’) as a baseline in terms of worst-case value size. Then we show four approaches to shrinking, as described above: truncated shrinking (’QC trunc’) with Int16 shrinking disabled, in which each list is truncated to 10 elements, tuple shrinking (’QC tuple’), also with Int16 shrinking disabled, and finally shrinking using QuickCheck’s genericShrink (’QC generic’). We graph the final counterexample size, measured in

![Figure 3: Results for 1000 tests of the program in Figure 2.](image)

### Table 1. Summarizing data for the graph in Figure 3. Entries contain execution time (in seconds) and counterexample sizes (counting Int16 values).

<table>
<thead>
<tr>
<th></th>
<th>Mean</th>
<th>Std. dev.</th>
<th>95%</th>
</tr>
</thead>
<tbody>
<tr>
<td>size</td>
<td>time</td>
<td>size</td>
<td>time</td>
</tr>
<tr>
<td>none</td>
<td>85</td>
<td>0.013</td>
<td>22</td>
</tr>
<tr>
<td>QC trunc</td>
<td>40</td>
<td>0.064</td>
<td>20</td>
</tr>
<tr>
<td>QC tuple</td>
<td>10</td>
<td>0.031</td>
<td>4</td>
</tr>
<tr>
<td>QC generic</td>
<td>13</td>
<td>0.204</td>
<td>7</td>
</tr>
<tr>
<td>SmartCheck</td>
<td>6</td>
<td>0.021</td>
<td>4</td>
</tr>
</tbody>
</table>

The total number of Int16 values, and execution time for counterexample discovery and shrinking. Figure 3 shows how many of the 1000 runs resulted in a final value of a particular size, fitting the data with a Bezier curve.

In Table 1, we summarize the averaged values and provide execution time (including both counterexample discovery and shrinking) in seconds as well. For both execution time and final value size, we summarize the mean, standard deviation, and the results at the 95th percentile. (While we provide the standard deviations, note that the plots are not necessarily Gaussian.)

SmartCheck produces consistently small counterexamples and does so quickly—even smaller and faster than the best hand-tuned solution—all without writing any problem-specific code. Note that the ’none’ example provides a lower-bound on execution time (only including counterexample discovery time) and an upper-bound on counterexample size. In this example, the hand-written “tuple shrink” algorithm is only slightly slower and produces slightly larger results than SmartCheck. (QuickCheck’s generic shrinking under performs the “tuple shrink” likely because the generic shrink implementation shrinks the Int16 values.)

In Section 7.3, we provide additional benchmarks, but now we turn to the design of SmartCheck.

### 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 class. (SmartCheck requires instances to be defined for data being ana-
lyzed; 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—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 constructor names 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, Char, or Bool, and may be user-defined. See Section 4.2.2 for a full discussion.
- subVals returns a tree of all non opaque-type sub-values. A tree has the type

```haskell
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 = B_0 (B_1 L_3)
      (B_2 L_5 L_7)

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

```haskell
size tree = 9
index tree 0 = tree
index tree 3 = (Just . SubVal) (B_4 L_6 L_8)
index tree 12 = Nothing

replace tree 2 (SubVal L) =
  B_0 (B_1 L_3)
  (B_4 L_6 L_8))
  L

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

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

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 3rd sub-value of tree. An invariant of index is that for any value a, and for the smallest i ≥ 0 such that

```haskell
index a i == Nothing
then for all 0 ≤ 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 [8] 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, the SubTypes class and its methods depend on GHC Generics [14], which we describe in Section 6. For simplicity, we omit here Generics-specific super-class constraints on the SubTypes class here. 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. Perhaps the most significant difference between SmartCheck’s reduction algorithm and QuickCheck shrink implementations is that the latter is deterministic. SmartCheck combines counterexample search with shrinking.

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. By a structurally smaller value, we mean one with fewer constructors. We continue until we reach a fixed-point.

Figure 4 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 (this is an optimization described in Section 4.2.1). When the index idx becomes out-of-bounds and returns Nothing, the algorithm terminates. Otherwise, a list of new random values are generated.

```haskell
sizedArbitrary :: SubTypes a => Size -> a -> a -> IO a
```
holds if \( cex' \) is a variant of \( cex \) and testing it. The result of \( \text{pass} \ prop \ cex' \) for

\[
\text{pass} :: (a \to \text{Property}) \to a \to \text{Bool}
\]

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 \( \text{pass} \) return \( \text{True} \). 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 \( O(n^2) \), 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 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 zeroth sub-value so that the variant of the counterexample remains the same. No invariant of the algorithm would be violated by beginning with the zeroth sub-value, and in particular, the algorithm would still terminate.

Incidentally, QuickCheck's generic shrink implementation is fundamentally built around the variant counterexample hypothesis. For a given counterexample \( cex \), smaller candidates produced by \( \text{shrink} \) contain only the constructors found in \( cex \) as opposed to just the outermost constructor, as in SmartCheck. Sometimes shrinking gets "stuck" at a local minimum due to a lack of entropy in generating smaller candidates.

The hypothesis may be unsuitable for some properties, in which case SmartCheck (and especially QuickCheck) may potentially fail to discover a smaller counterexample. However, in Sections 5.2 and 5.3, we describe 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 \( \text{shrink} \) method. Consequently, SmartCheck ignores "primitive" types without value constructors, such as \( \text{Char} \), \( \text{Int} \), and \( \text{Word16} \). 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 \text{opaque types}. If the reduction algorithm encounters an opaque type, it is ignored. Opaque types include the primitive types mentioned above, but the user can declare any substructure in a data type to be an 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 elements of the list are opaque themselves. Such a definition is possible. Sometimes shrinking primitive types is imperative; for example, to determine if there is a relationship between two integers in a counterexample.

Opaque types are defined by providing the method for \( \text{opaque} \) in the \text{SubTypes} type class.

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 \text{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.

```haskell
generateSubValue :: SubVal a -> a

newVals :: SubVal a -> [SubVal]

newVals (SubVal v) = 
  replicateM 10 v

getSize :: SubVal -> Size

getSize (SubVal v) = newVals (getSize v)

scMaxReduce = do
  vs <- newVals (getSize v) |
  Just v <- index cex idx |
  otherwise = return cex

  case test cex idx vs prop of
  Nothing -> reduce' (idx+1)
  Just a -> reduce args prop a |
  otherwise = return cex

reduce' idx
|
  Just v <- index cex idx |
  do vs <- newVals (getSize v)
  case test cex idx vs prop of
  Nothing -> reduce' (idx+1)
  Just a -> reduce args prop a |
  otherwise = return cex

  | otherwise = return cex

reduce args prop cex' = reduce' 1

where

reduce' idx
|
  Just v <- index cex idx |
  do vs <- newVals (getSize v)
  case test cex idx vs prop of
  Nothing -> reduce' (idx+1)
  Just a -> reduce args prop a |
  otherwise = return cex

| otherwise = return cex

figure 4: counterexample reduction algorithm.
```
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 =
          | case testHole v of
          |   Nothing -> test' v idx
          |   Just a -> reduceOpt args prop a
          | otherwise = return cex

    test' v idx = do
        vs <- newVals (getSize v) (scMaxReduce args) v
        case test cex vs of
            Nothing -> reduce' (idx+1)
            Just a -> reduceOpt args prop a

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

Figure 5: Reduction algorithm with the sub-value counterexample optimization.

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)
    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
divSubTerms (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 (Add (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 5, 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, then we run the execution algorithm on 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:

```
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 4. 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. Characterizing formula reduces the noise in favor of the signal by abstracting away portions of a 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 automatically strengthen a property’s precondition to obtain new counterexamples.

In SmartCheck, the universal and existential generalization phases run after a counterexample has been minimized. Precondition strengthening is used when iteratively generating multiple counterexamples, so it is interspersed with counterexample reduction.
extrapolate :: SubTypes a -> a -> [Idx] -> IO [Idx]
extrapolate cex idx prop = extrapolate' 1 []
    where
    extrapolate' idx idxs
        | subTrees cex idx dsx = extrapolate' (idx+1) idxs
        | Just v <- index cex idx dx = 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 idxs
    length res >= scMinForall args && and res
    where
    res = mapMaybe go vs
    go = fail prop . replace cex idx

Figure 6: Universal sub-value generation algorithm.

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

\[ \text{Div } (\text{Add } (C 7) (C 3)) (\text{Add } (C (-5)) (C 5)) \]

by the formula

\[ \forall x . \text{Div } x (\text{Add } (C (-5)) (C 5)) \]

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 6; let us sketch its specification. The algorithm is similar to the reduction algorithm in Figure 4 (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

\[ \text{subTree } cex \text{ idx0 } \text{ idx1} \]

where subTree has the type

\[ \text{subTree} :: \text{SubTypes a} \Rightarrow a \Rightarrow \text{Idx} \Rightarrow \text{Idx} \Rightarrow \text{Bool} \]

The value

\[ \text{subTree } cex \text{ idx0 } \text{ idx1} \]

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 4.

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

\[ \text{fail :: (a -> Property)} \Rightarrow a \Rightarrow \text{Maybe Bool} \]

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 \(O(n^2)\), where \(n\) is the number of constructors in the counterexample. Again, we assume that the cost for 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.

Second, 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:

\[ \text{Add } (\text{Div } (C 1) (\text{Add } (C (-2)) (C 2))) \]

\[ \text{Add } (C 0) (\text{Add } (C (-1)) (C 1)) \]

Since no matter what random value the first field of the outermost Add constructor is replaced with, the property fails by Add’s second field, and vice versa for replacing the second field. Consequently, the universal generalization algorithm might return the formula

\[ \forall \text{values } x0 \text{ x1 . Add } x0 \text{ x1} \]

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

\[ \forall \text{values } x0 \text{ x1 . Add } x0 \text{ x1} \]

one should read

\[ \forall \text{values } x0 . \text{Add } x0 \text{ (Div } (C 0) (C (-1)) (C 1)) \]

and

\[ \text{forall values } x1 \]

\[ \text{Add } (\text{Div } (C 1) (C (-2)) (C 2)) \text{ x1} \]

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 be 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 v .
such that Div x (c v)

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 7. The function sumTest takes a set of flags, a counterexample, a property, and a list of indexes that have already been extrapolated—perhaps by the extrapolation algorithm in Figure 6. 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 4. 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.
Then the following hold:

\[ e_3 = \text{Div}\left(\text{Div}\left(\text{C}\ 8\right)\left(\text{C}\ 2\right)\right)\left(\text{C}\ 7\right)\right) \]

\[ e_2 = \text{Div}\left(\text{Add}\left(\text{C}\ 1\right)\left(\text{C}\ 2\right)\right)\left(\text{C}\ 7\right)\right) \]

\[ e_1 = \text{Div}\left(\text{C}\ 1\right)\left(\text{C}\ 3\right) \]

\[ e_0 = \text{Div}\left(\text{C}\ 1\right)\left(\text{C}\ 2\right) \]

Exp

values of type

terexamples at the risk of omitting some.

choice that is more aggressive about covering the space of coun-
teriorly generalized indexes to match any value. Doing so is a design
dition strengthening. The function takes a candidate counterexam-
all their children) match any value, since a universally generalized
matching algorithm used for precondition strengthening. The function takes a candidate counterexam-
example generalization.

Figure 9 shows the shape-matching algorithm used for precon-
ders, their constructors at each node in the tree

evaluation. Therefore, properties provided to SmartCheck are expected to be of

instance Subtypes D

Predefined 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 itself. An initial counterexample can be passed in explicitly, so SmartCheck will use QuickCheck as a library to generate an initial counterexample to analyze.

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.

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.

\[ \text{matchesShape } e_0 \left( \text{e1, \{\}} \right) = \text{True} \]

\[ \text{matchesShape } e_1 \left( \text{e2, \{\}} \right) = \text{False} \]

\[ \text{matchesShape } e_1 \left( \text{e2, [1]} \right) = \text{True} \]

\[ \text{matchesShape } e_3 \left( \text{e2, [1]} \right) = \text{True} \]

The first equation holds because we ignore opaque types, so the arguments to C are considered equal. The second equation fails because at index 1, e1 contains the constructor C and e2 contains the constructor Add. If, however, index 1 has been generalized, then we ignore the sub-value at index 1, and they match (the third equation). The same reasoning holds for the fourth equation.

SmartCheck can run in a real-eval-print loop (REPL), discovering new counterexamples, shrinking them, generalizing them, and then repeating. In each iteration of the REPL, the property precondition is strengthened, requiring that matchesShape does not hold on the current counterexample candidate and any previous counterexample generalization already discovered.

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 generics library for Haskell [14]. The 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 [11].

The data type to be tested must derive the Typeable and Generic type classes. Deriving Typeable and Generic requires using the GHC library 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,

\[ \text{instance Subtypes D} \]

4https://github.com/leepike/SmartCheck.git
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

```haskell
> 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

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

as

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

We find that for very large 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. Then we present a small set of benchmarks comparing SmartCheck and QuickCheck.

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. An existing deletion function in `XMONAD` exists, 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:

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

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

```haskell
removeFromWorkspace ws =
  ws { stack = stack ws >>= filter (== u) }
```

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 Figure 1. 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 fourth element of `StackSet` is not generalized, but is simply the empty map, which looks uninteresting. The second and third fields of `StackSet` are generalized, but the first one is not. 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 data types. The poster writes:

```text
... 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.
```

The question relates to the Geni natural language processing (NLP) package implemented in Haskell [12]. 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 (in private communication):

```text
... While your improved shrinking may not have gone ‘all’ the way to the bottom, it got me a huge chunk of the way there!
```

Through the entire process, we never had to learn how Geni works, what the property meant, or how to write a custom shrink method!

7.3 Benchmarks

Unfortunately, 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:

> http://stackoverflow.com/questions/8788542/
how-do-i-get-good-small-shrinks-out-of-quickcheck
All benchmarks can be found online. We compare the size of the final counterexample returned (by counting constructors) and the time required for counterexample generation and shrinking in seconds. The results are presented in Table 2. Again, we summarize the mean, standard deviation, and the results at the 95th percentile. (While we provide the standard deviations, note that the plots are not necessarily Gaussian.)

### Table 2. Summarizing data for the graphs in Figure 3. Entries contain execution time (in seconds) and counterexample sizes (counting constructors).

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Method</th>
<th>Size</th>
<th>Time (sec)</th>
<th>Std. Dev.</th>
<th>95th Percentile</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reverse</td>
<td>QuickCheck</td>
<td>2.002</td>
<td>0.002</td>
<td>2.000</td>
<td>0.002</td>
</tr>
<tr>
<td></td>
<td>SmartCheck</td>
<td>2.001</td>
<td>0.001</td>
<td>2.001</td>
<td>0.001</td>
</tr>
<tr>
<td>Div0</td>
<td>QuickCheck</td>
<td>0.004</td>
<td>0.006</td>
<td>0.005</td>
<td>0.007</td>
</tr>
<tr>
<td></td>
<td>SmartCheck</td>
<td>0.001</td>
<td>0.001</td>
<td>0.001</td>
<td>0.001</td>
</tr>
<tr>
<td>Heap</td>
<td>QuickCheck</td>
<td>9e-9</td>
<td>0.001</td>
<td>9e-9</td>
<td>0.001</td>
</tr>
<tr>
<td></td>
<td>SmartCheck</td>
<td>7e-9</td>
<td>0.002</td>
<td>7e-9</td>
<td>0.002</td>
</tr>
<tr>
<td>Parser</td>
<td>QuickCheck</td>
<td>0.010</td>
<td>0.006</td>
<td>0.011</td>
<td>0.025</td>
</tr>
<tr>
<td></td>
<td>SmartCheck</td>
<td>0.124</td>
<td>0.124</td>
<td>0.124</td>
<td>0.124</td>
</tr>
</tbody>
</table>

8. Related Work

Zeller and Hildebrandt describe an application of greedy search to shrink counterexamples they call “delta-debugging” (DD) [20]. 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. The idea of randomization in test-case reduction was independently developed at approximately the same time as SmartCheck and first published in the domain of reducing C programs that demonstrate compiler bugs [17]. We believe our work is the first to explore 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 [13]. There, the motivation was not to design new approaches to counterexample reduction, but simply to derive instances for the shrink method.

SmallCheck is another testing framework for Haskell for which shrinking is irrelevant: SmallCheck is guaranteed to return a smallest counterexample, if one exists [18]. 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 [10]), counterexamples to even relatively simple properties may be practically infeasible to discover due to the size of the input space. For example, SmallCheck does not find a counterexample to the example presented in Section 2 after running it for several minutes.

Besides QuickCheck and SmallCheck, another testing framework related to SmartCheck is the recent Haskell library Feast [6]. Feast 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. Feast 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 is used to discover counterexamples, not shrink them. However, shrinking is less necessary with Feast, since discovered counterexamples are often small, if one is found. For example, on the overflow example in Section 2, with a limit of 100 test cases, Feast finds a counterexample just two percent of the time, whereas QuickCheck finds one nearly 100%. Even at a limit of 10000, a counterexample is found about 50% of the time (with a correspondingly longer search time). Sampling from a uniform distribution does not work so well here. Feast does a better job of discovering counterexamples in the parser benchmark, but the size of the average counterexample contains 500 constructors, with a standard deviation of 500 (compared with 16 and 75, respectively, for SmartCheck). Still, Feast is powerful at what it does well and can be seamlessly used with SmartCheck, since it just defines the arbitrary method.

Finally, SmartCheck bears some similarity to QuickSpec, a testing-based library that infers equational properties about pro-

grams [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.

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 [18]. 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

\[
\text{Add (Div (C 1) (Add (C (-2)) (C 2)))} \\
\text{(Div (C 1) (Add (C (-1)) (C 1)))}
\]

we might instead return

\[
\text{Add (Div (-2) 2) (Div (-1) 1)}
\]

where \( \text{div x y = Div (C 1) (Add (C x) (C y))} \)

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!

Acknowledgments

I thank Andy Gill, Joe Hurd, John Launchbury, Simon Winwood, Tritan Ravitch, the anonymous reviewers of ICFP’13 (from which an earlier draft of this paper was rejected), and especially John Hughes. Their deep insights and bug catches dramatically improved the presentation and content.

References


