Random Testing of Purely Functional Abstract Datatypes

Guidelines for Dealing with Operation Invariance

Stefan Holdermans
Vector Fabrics
stefan@vectorfabrics.com

ABSTRACT

Algebraic specification, equational reasoning, and property-based random testing provide functional programmers with a powerful yet reasonably lightweight framework for reasoning about abstract datatypes and assuring the quality of their concrete implementations. However, as it turns out, naïvely implementing property-based random testing for purely functional abstract datatypes is prone to subtle errors and may well leave unaware practitioners with a false sense of security about their datatype implementations. In this paper, we pinpoint one particular pitfall, namely overlooking the need to take into account the invariance of datatype operations under an implied equivalence relation on concreate datatype values, and discuss how to systematically avoid it. Presented in the context of a concrete case study, the proposed technique generalises nicely into a common design principle for engineering random tests of purely functional datastructures.

Categories and Subject Descriptors

D.1.1 [**Programming Techniques**]: Applicative (Functional) Programming; D.2.1 [**Software Engineering**]: Requirements/Specifications; D.2.4 [**Software Engineering**]: Software/Program Verification—*Formal methods, Reliability*; D.2.5 [**Software Engineering**]: Testing and Debugging—*Testing tools (e.g., data generators, coverage testing)*; D.2.11 [**Software Engineering**]: Software Architectures—*Data abstraction*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*Abstract data types*

General Terms

Design, Reliability, Verification

Keywords

abstract datatypes, algebraic specification, equational reasoning, property-based testing, random testing

PPDP '13, September 16 - 18 2013, Madrid, Spain

ACM 978-1-4503-2154-9/13/09 ...$15.00.
http://dx.doi.org/10.1145/2505879.2505880.

1. INTRODUCTION

Modern programming languages provide developers with an impressive set of tools for assessing and assuring the correctness of their datastructure implementations. Equipped with powerful type and module systems, in particular functional languages like Haskell [21], Clean [22], and ML [19] offer good facilities for data abstraction and hence allow for datastructures to be implemented and organised as *abstract datatypes*.

An abstract datatype [18] is defined only by the operations that may be performed on it. In particular, it is defined independent from its concrete implementation, allowing for this implementation to be hidden from any codes that make use of the datatype and its operations. As a result, taking full advantage of encapsulation, implementations of abstract datatypes can change without affecting any client codes and client codes can easily switch between different implementations of a single abstract datatype.

A fitting framework [11, 9] for the definition of abstract datatypes is that of *algebraic specification* [3, 24], in which datatypes and the semantics of their operations are described by algebras. Algebraic specifications of abstract datatypes are a particularly good fit in the context of *purely* functional languages (such as Haskell and Clean) as the absence of unrestricted destructive assignment in these languages allows for the derivation of whole classes of properties from only a handful of algebraic axioms by mere and often straightforward *equational reasoning* [23]. For implementors of abstract datatypes it suffices to demonstrate that their concrete implementations fulfill the axioms of the specifications in order to have all derived properties hold. One popular and lightweight approach to doing so is by means of *property-based random testing* [12]: from the axioms one derives a small set of testable properties, which in turn give rise to an arbitrary large or perhaps even exhaustive set of randomly generated test cases.

What makes abstract datatypes implemented in purely functional languages especially suitable to this form of testing is that, in such languages, operations on datatypes are implemented simply as functions and their execution thus only depends on their inputs. Not having to concern oneself with how mutable state can affect the outcome of a computation and not having to assert that such state is updated correctly during the execution of an operation indeed greatly simplifies test engineering. Consequently, properties can also be represented by functions and test cases can be derived automatically by randomly choosing inputs to those [6, 7, 16, 1].

The resulting scheme for verifying the realisation of an abstract datatype is arguably elegant and easy to implement. Yet, in practice, being by definition generally nonexhaustive, random testing requires a lot of tuning in order to reliably assess the quality of nontrivial bodies of code. Sometimes it is all too easy to get it wrong and to end up in a situation in which random testing gives

nothing but a false sense of security.

In this paper, we highlight and—moreover—demonstrate how to systematically avoid one particular pitfall that precipitates incorrect implementations of abstract datatypes to pass test suites that were derived from seemingly complete specifications. This pitfall centres on the typically overlooked need to take into account the invariance of datatype operations under an implied equivalence relation on concrete datatype values. Avoiding this pitfall, in our proposal, amounts to adhering to a very general guideline for deriving properties to test against.

Specifically, our contributions are the following:

- We discuss how, in a purely functional setting, algebraic specification and equational reasoning allow for a whole range of theorems about abstract datatypes to be derived from small sets of axioms and to apply to all correct implementations of the datatypes involved, leaving to implementors no more than attestation to the axioms (Section 2).
- We revisit a problem with property-based testing of purely functional abstract datatypes that was accentuated in recent literature [17] and argue that specifications that involve a notion of equality between values of abstract datatypes, in order to justify equational reasoning, need to come with axioms that express that datatype operations are compatible with equality (Section 4).

Throughout this paper, we use Haskell as a language for implementing, testing, and reasoning about abstract datatypes. It should be noted, however, that all ideas and concepts transfer readily to other modern functional languages, such as Clean and the purely functional fragment of ML. In particular, these languages have ports or substitutes available for QuickCheck [6], the library that we use for property-based random testing of datatype implementations. A quick introduction to QuickCheck is given in Section 3.

2. TESTS AND PROOFS FOR PURELY FUNCTIONAL ABSTRACT DATATYPES

In this section, we take a closer look at algebraic specifications for Haskell datatypes (Section 2.1) and show how equational reasoning can be applied to derive theorems about datatypes from just their specifications (Section 2.2).

2.1 Algebraic Specification

An algebraic specification consists of a signature and a set of axioms. The signature, in turn, consists of a sort and a listing of operations and the sorts they apply to. In this paper, we are specifying the behaviour of Haskell datatypes, and so we have sorts corresponding to Haskell types and operations to Haskell functions.

As a simple example, Figure 1 presents a specification for the type of Booleans. Its signature introduces the sort Bool and lists the types for the constructors False and True as well as the connectives for negation, conjunction, and disjunction. The intended semantics of the connectives is captured by six axioms. Any metavariables mentioned in an axiom are implied to be universally quantified over at the axiom level. Hence, in Figure 1, the axioms B3, B4, B5, and B6 each quantify over the metavariable b, which can be told from the types of the operations to range over the sort Bool.

The given specification for Booleans is so straightforward that there is actually a one-to-one correspondence with how Booleans are implemented in the Haskell standard library. Therefore, let us also consider a datatype specification with a less direct connection to its typical implementations.

signature:

sort:

Bool

operations:

False	::	Bool
True	::	Bool
$\neg$	::	Bool $\to$ Bool
$(\wedge)$	::	Bool $\to$ Bool $\to$ Bool
$(\vee)$	::	Bool $\to$ Bool $\to$ Bool

axioms:

B1:	$\neg$ False	=	True
B2:	$\neg$ True	=	False
B3:	False $\wedge$ b	=	False
B4:	True $\wedge$ b	=	b
B5:	False $\vee$ b	=	b
B6:	True $\vee$ b	=	True

Figure 1: Algebraic specification of Booleans.

Figure 2 presents a specification for FIFO queues. This specification assumes that we have an implementation of Booleans in place that satisfies the specification from Figure 1. Furthermore, it refers to a sort Int of integers, that we will use to range over the elements that can go into a queue. The signature for queues then introduces a sort Queue; a nullary operation empty for obtaining the empty queue; a constructor operation enqueue for obtaining a new queue by adding an element to an existing queue; a query operation isEmpty for finding out whether a queue is empty; an operation front for selecting the oldest element from a queue, and an operation dequeue for obtaining a new queue by removing the oldest element from an existing queue.

The semantics of queues is captured by six axioms. The first axiom, Q1, states that the empty queue is indeed empty; the second, Q2, that adding an element to any queue yields a nonempty queue. Axiom Q3 says that adding an element to the empty queue and then querying for the front element of the resulting queue produces that element again. Axiom Q4 deals with adding an element to a nonempty queue and states that selecting the front element of the new queue reduces to selecting the front element of the original queue. Note that the nonemptiness of the original queue is enforced by a side condition on the axiom that states that querying for the emptiness of the queue should yield the Boolean False. The final two axioms deal with the dequeue operation. Q5 conveys that dequeueing from a queue that is obtained by enqueueing to the empty queue yields the empty queue again. Q6 expresses that first enqueueing an element to a nonempty queue and then dequeueing is equivalent to first dequeueing from the nonempty queue and then enqueueing the element to the resulting queue. As in Q4, the nonemptiness of the original queue in Q6 is enforced by a side condition.

As it turns out, the given signature and axioms constitute a complete specification for FIFO queues. Note that the axioms leave the semantics of selecting or dequeueing from the empty queue undefined.

2.2 Equational Reasoning

Now assume that we have a Haskell implementation of FIFO queues that satisfies the specification from Figure 2. In particular, such an implementation conforms to the six queue axioms.

One of the attractive aspects of purely functional programs is

signature:

sort:

Queue

operations:

empty	::	Queue
enqueue	::	Int → Queue → Queue
isEmpty	::	Queue → Bool
front	::	Queue → Int
dequeue	::	Queue → Queue

axioms:

Q1:	isEmpty empty	=	True	
Q2:	isEmpty (enqueue x q)	=	False	
Q3:	front (enqueue x empty)	=	x	
Q4:	front (enqueue x q)	=	front q	(if isEmpty q = False)
Q5:	dequeue (enqueue x empty)	=	empty	
Q6:	dequeue (enqueue x q)	=	enqueue x (dequeue q)	(if isEmpty q = False)

Figure 2: Algebraic specification of FIFO queues.

that they can be subjected to equational reasoning. This way, starting from just the queue axioms and simply manipulating Haskell expressions by "substituting equals for equals", we can derive a whole class of theorems for our queue implementation. Moreover, we can do so without making any specific assumptions about the implementation other than that it indeed satisfies its specification.

As a simple example, consider the following theorem:

```
isEmpty (dequeue (enqueue x empty))  =  True.
```

Its proof, by equational reasoning, proceeds by invoking the axioms Q5 and Q1:

```
    isEmpty (dequeue (enqueue x empty))
=       { Q5 }
    isEmpty empty
=       { Q1 }
    True.
```

That is, we first apply axiom Q5 to have the subexpression

```
dequeue (enqueue x empty)
```

replaced by empty. This yields the expression isEmpty empty, which then, by axiom Q1, can be substituted by the Boolean True.

As a second example, we have the theorem

```
front (dequeue (enqueue x (enqueue y (enqueue z empty))))
                                                    = y
```

and its proof

```
    front
      (dequeue (enqueue x (enqueue y (enqueue z empty))))
=       { Q6, Q2 }
    front
      (enqueue x (dequeue (enqueue y (enqueue z empty))))
=       { Q6, Q2 }
    front
      (enqueue x (enqueue y (dequeue (enqueue z empty))))
=       { Q5 }
    front (enqueue x (enqueue y empty))
=       { Q4, Q2 }
    front (enqueue y empty)
=       { Q3 }
    y.
```

Here, to apply axiom Q6 in order to have the subexpression

```
dequeue (enqueue x (enqueue y (enqueue z empty)))
```

replaced by

```
enqueue x (dequeue (enqueue y (enqueue z empty))),
```

we first have to show that the side condition for Q6 is satisfied, i.e., that

```
isEmpty (enqueue y (enqueue z empty))  =  False.
```

This is achieved by invoking axiom Q2. Then, in a similar fashion, we replace

```
dequeue (enqueue y (enqueue z empty))
```

by

```
enqueue y (dequeue (enqueue z empty))
```

to yield the expression

```
front
  (enqueue x (enqueue y (dequeue (enqueue z empty)))).
```

Next, we invoke Q5 to substitute dequeue (enqueue z empty) by empty and then Q4 and Q2 to yield

```
front (enqueue y empty).
```

Finally, by Q3, we obtain the required right-hand side y.

Note that the given theorems—and, for that matter, all theorems derived in a similar manner—apply to all concrete implementations of FIFO queues, provided that they abide by the specification from Figure 2. What is left to the devisers of such implementations is then to actually make certain that their queues behave in accordance with the axioms. One way to do that is by exposing those implementations to an extensive enough series of tests that randomly construct queues and then assert that application of the queue operations yields outcomes that conform to the axioms. The most widely used tool for realising such tests in Haskell is QuickCheck, the use of which we will discuss in the next section.

3. A QUICKCHECK PRIMER

In this section, we will explore property-based random testing with QuickCheck [6]. QuickCheck is a Haskell library for random testing, essentially consisting of two embedded domain-specific languages: one for defining properties (Section 3.1) and one for defining test-data generators (Section 3.2). To ease the presentation, we have simplified some aspects of the actual implementation—for example, by instantiating the types of some overloaded functions.

Readers already familiar with QuickCheck may skip this section and proceed to Section 4.

3.1 Properties

Consider the following—not so efficient—function for reversing a list of integers:

```
reverse          :: [Int] → [Int]
reverse []       = []
reverse (x : xs) = reverse xs ++ [x].
```

For finite lists, reverse satisfies the following properties:

$$\begin{aligned} \mathsf{reverse}\ [\,] &= [\,] \\ \mathsf{reverse}\ [x] &= [x] \\ \mathsf{reverse}\ (xs \mathbin{+\!\!+} ys) &= \mathsf{reverse}\ ys \mathbin{+\!\!+} \mathsf{reverse}\ xs. \end{aligned}$$

The first two of these state that reversing a list with less than two elements yields the original list again. The third property states that reversal of a list that can be constructed by concatenating two smaller lists xs and ys obtains the same result as appending the list obtained by reversing xs to the list obtained by reversing ys.

We can use QuickCheck to verify that these properties indeed hold for reverse. To do so, we need to encode them as values of a type Property provided by QuickCheck:

```
p1 :: Property
p1 = property (reverse [] ≡ [])

p2 :: Property
p2 = property (λx → reverse [x] ≡ [x])

p3 :: Property
p3 = property (λxs ys →
       reverse (xs ++ ys) ≡ reverse ys ++ reverse xs).
```

That is, QuickCheck properties arise from functions that produce values of type Bool. All variables implicitly universally quantified over in the original properties show up as parameters of the functions. The first property does not quantify over any variables and hence is represented by a nullary function, i.e., a Bool-typed constant.

Functions representing properties are turned into values of type Property by the overloaded function property from the QuickCheck library,

```
property :: Testable a ⇒ a → Property,
```

which is defined for all types in the class Testable. Members of this class are, amongst others, the type Property itself, the type Bool, and all types of functions that take printable values of types in the class Arbitrary to values of types that are themselves in the class Testable:

```
instance Testable Property where ...
instance Testable Bool where ...
instance Testable (Arbitrary a, Show a, Testable b) ⇒
  Testable (a → b) where ... .
```

The class Arbitrary is provided by QuickCheck and contains all types for which values can be generated at random:

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

with Gen a the type of generators that produce random values of type a.

In the properties p1, p2, and p3 defined above, we have instantiated property for the types Bool, Int → Bool, and [Int] → [Int] → Bool, respectively—and indeed QuickCheck comes with instances of Arbitrary for Int and, by means of a polymorphic instance for lists, [Int]:

```
instance Arbitrary Int where ...
instance Arbitrary a ⇒ Arbitrary [a] where ... .
```

Below, in Section 3.2, we will discuss how to supply instances of Arbitrary for user-defined datatypes, but let us now first see how properties are used to automatically derive test cases from.

For this purpose, the QuickCheck library exports a driver function quickCheck,

```
quickCheck :: Property → IO (),
```

which takes a property and invokes the Arbitrary-supplied generators to produce random arguments for the function from which the property was derived. The function is then executed 100 times (by default, that is; the actual number of runs is configurable); if the function returns True in all cases, an affirmation of the property is printed to the environment's standard output stream. For instance, quickCheck p1 produces the output

```
+++ OK, passed 100 tests.
```

Using a small helper function,

```
quickCheckMany :: [Property] → IO ()
quickCheckMany = mapM_ quickCheck,
```

we can test all three properties of reverse in a single batch by running quickCheckMany [p1, p2, p3]:

```
+++ OK, passed 100 tests.
+++ OK, passed 100 tests.
+++ OK, passed 100 tests.
```

Now let us replace our implementation of reverse by a more efficient—but unfortunately incorrect—version that makes use of an accumulating parameter [4]:

```
reverse :: [Int] → [Int]
reverse = go id
  where
    go f []       = []    -- incorrect!
    go f (x : xs) = go ((x :) ∘ f) xs.
```

The idea here is that, as we traverse the input list, we maintain a function f that prepends all elements encountered so far in reverse order to its argument. At the end of the list, we then apply this function to the empty list to obtain the original list in reverse. This way, we avoid the quadratic runtime due to repeated list concatenation exhibited by our original version of reverse. However, in our new version, in the first case of the helper function go, rather than applying the accumulated function f to the empty list as we are supposed to, we accidentally ignore f and always return the empty list.

Our mistake immediately shows if we attempt to verify that the properties p1, p2, and p3 still hold for our supposedly improved version of reverse:

```
+++ OK, passed 100 tests.
*** Failed! Falsifiable (after 1 test):
0
+++ OK, passed 100 tests.
```

QuickCheck reports that reverse still passes the tests for p1 and p3 (and, indeed, these properties still hold), but fails to satisfy p2 and that a counter example for this property can be constructed by instantiating the parameter of the function $\lambda x \to \mathsf{reverse}\ [x] \equiv [x]$, that was used to define the property, with the value 0. Counter examples reported by QuickCheck are not necessarily directly constructed from the randomly generated function arguments for which the property of interest failed; for types that support this, QuickCheck first tries to find similar but smaller values that also make the property fail and then reports those instead. This process, called *shrinking*, allows for minimal and therefore easy to understand counter examples.

QuickCheck is also capable of expressing properties with side conditions. For this, it provides a combinator $(\Rightarrow)$,

$$(\Rightarrow) :: \mathsf{Testable\ a} \Rightarrow \mathsf{Bool} \to \mathsf{a} \to \mathsf{Property},$$

that constructs a property from a testable value and a side condition. Now, when QuickCheck tests a property with a side condition, it checks the property not with just 100 random test cases, but with 100 test cases that actually satisfy the side condition. This means that it generates test cases until it has found 100 that satisfy the condition, discarding the ones that do not. As an example, here is a property that states that reversing a list puts its last element in head position:

$$\begin{aligned}&\mathsf{p4} :: \mathsf{Property}\\ &\mathsf{p4} = \mathsf{property}\ (\lambda \mathsf{xs} \to \neg\ (\mathsf{null\ xs}) \Rightarrow\\ &\qquad \mathsf{head}\ (\mathsf{reverse\ xs}) \equiv \mathsf{last\ xs}).\end{aligned}$$

Of course, this property is only meaningful for lists that have at least one element. Hence, the side condition ensures that the property will only be checked for nonempty lists.

By now, it should be clear how we can use QuickCheck to test against algebraic specifications. For example, from the axioms for Booleans from Figure 1 we derive the following QuickCheck properties:

$$\begin{aligned}&\mathsf{b1} :: \mathsf{Property}\\ &\mathsf{b1} = \mathsf{property}\ (\neg\ \mathsf{False} \equiv \mathsf{True})\\ &\mathsf{b2} :: \mathsf{Property}\\ &\mathsf{b2} = \mathsf{property}\ (\neg\ \mathsf{True} \equiv \mathsf{False})\\ &\mathsf{b3} :: \mathsf{Property}\\ &\mathsf{b3} = \mathsf{property}\ (\lambda \mathsf{b} \to (\mathsf{False} \wedge \mathsf{b}) \equiv \mathsf{False})\\ &\mathsf{b4} :: \mathsf{Property}\\ &\mathsf{b4} = \mathsf{property}\ (\lambda \mathsf{b} \to (\mathsf{True} \wedge \mathsf{b}) \equiv \mathsf{b})\\ &\mathsf{b5} :: \mathsf{Property}\\ &\mathsf{b5} = \mathsf{property}\ (\lambda \mathsf{b} \to (\mathsf{False} \vee \mathsf{b}) \equiv \mathsf{b})\\ &\mathsf{b6} :: \mathsf{Property}\\ &\mathsf{b6} = \mathsf{property}\ (\lambda \mathsf{b} \to (\mathsf{True} \vee \mathsf{b}) \equiv \mathsf{True}).\end{aligned}$$

Using these, we can then—successfully—seek confirmation that the type Bool and the operators $\neg$, $(\wedge)$, and $(\vee)$ from the Haskell standard library satisfy the specification from the previous section.

But what about axioms that give rise to functions taking arguments of user-defined types—as, for example, the axioms Q2, Q4, and Q6 from Figure 2? To encode these in QuickCheck, we need to provide instances of the class Arbitrary for the types involved. We discuss how to define such instances for user-defined datatypes in the next subsection.

3.2 Generators for User-defined Types

Consider the user-defined type Tree of node-labelled binary trees.

$$\mathbf{data}\ \mathsf{Tree} = \mathsf{Leaf} \mid \mathsf{Node\ Tree\ Int\ Tree}\ \mathbf{deriving}\ \mathsf{Show}$$

That is, a tree is either a leaf or otherwise a node with a left subtree, an integer label, and a right subtree.

If, for this type, we define functions size and depth,

$$\begin{aligned}&\mathsf{size} &&:: \mathsf{Tree} \to \mathsf{Int}\\ &\mathsf{size\ Leaf} &&= 0\\ &\mathsf{size\ (Node\ l\ x\ r)} &&= \mathsf{size\ l} + 1 + \mathsf{size\ r}\\ &\mathsf{depth} &&:: \mathsf{Tree} \to \mathsf{Int}\\ &\mathsf{depth\ Leaf} &&= 0\\ &\mathsf{depth\ (Node\ l\ x\ r)} &&= \mathsf{max\ (depth\ l)\ (depth\ r)} + 1,\end{aligned}$$

we can define properties such as

$$\begin{aligned}&\mathsf{t1} :: \mathsf{Property}\\ &\mathsf{t1} = \mathsf{property}\ (\lambda \mathsf{t} \to \mathsf{size\ t} \geqslant \mathsf{depth\ t}).\end{aligned}$$

However, for this definition to type check, Tree needs to be an instance of the class Arbitrary, i.e., there needs to be an implementation of the method arbitrary that provides a generator for values of type Tree.

Recall that QuickCheck generators for values of some type a are themselves values of a type Gen a. Generators are structured as monads,

$$\mathbf{instance}\ \mathsf{Monad\ Gen}\ \mathbf{where}\ldots,$$

and so functions that produce generators can be defined using Haskell's **do**-syntax. In addition, the QuickCheck library exports a combinator language for constructing more complex generators from simpler ones.

Let us start exploring generators by defining two very simple generators for trees. The first, arbitraryLeaf, always generates a leaf.

$$\begin{aligned}&\mathsf{arbitraryLeaf} :: \mathsf{Gen\ Tree}\\ &\mathsf{arbitraryLeaf} = \mathsf{return\ Leaf}\end{aligned}$$

The second, arbitraryNode, generates nodes.

$$\begin{aligned}&\mathsf{arbitraryNode} :: \mathsf{Gen\ Tree}\\ &\mathsf{arbitraryNode} = \mathbf{do}\ \mathsf{l} \leftarrow \mathsf{arbitrary}\\ &\qquad\qquad\qquad\quad \mathsf{x} \leftarrow \mathsf{arbitrary}\\ &\qquad\qquad\qquad\quad \mathsf{r} \leftarrow \mathsf{arbitrary}\\ &\qquad\qquad\qquad\quad \mathsf{return\ (Node\ l\ x\ r)}\end{aligned}$$

Here, we call the class method arbitrary to randomly generate, respectively, a left subtree l, a label x, and a right subtree r. For this to work, we once again need the type Tree to be in the class Arbitrary. To achieve this, we tie the recursive knot and define an Arbitrary instance for Tree by combining arbitraryLeaf and arbitraryNode into a single generator for trees.

$$\begin{aligned}&\mathbf{instance}\ \mathsf{Arbitrary\ Tree}\ \mathbf{where}\\ &\quad \mathsf{arbitrary} = \mathsf{oneof}\ [\mathsf{arbitraryLeaf}, \mathsf{arbitraryNode}]\end{aligned}$$

The combinator oneof takes a list of generators for a given type and produces a new generator that, when invoked, makes a random choice amongst the generators in the list.

$$\mathsf{oneof} :: [\mathsf{Gen\ a}] \to \mathsf{Gen\ a}$$

While this gives us a valid and straightforward generator for trees, in practice, we need some more fine-grained control over the generation of values.

For one thing, as any of the generators passed to oneof has an equal chance of being selected, half of the trees generated by our implementation of arbitrary will be leaves. Hence, if we use our generator to check the property t1 above, we end up executing the exact same test 50 out of 100 times (on average, that is). To remedy this situation, we can use the combinator frequency instead.

```
frequency :: [(Int, Gen a)] → Gen a
```

With this combinator, we can specify the frequency with which the generators for leaves and nodes are selected:

```
instance Arbitrary Tree where
  arbitrary = frequency
                [(1, arbitraryLeaf), (4, arbitraryNode)].
```

Now we have that, on average, nodes are generated four times as often as leaves.

But now another problem becomes apparent: with the chances of leaves being generated shrinking, the chances of our tree generator terminating decrease as well. The solution is to control and limit the size of generated trees. With the combinator sized,

```
sized :: (Int → Gen a) → Gen a,
```

which lifts size-aware generators to normal generators, we can express our instance of Arbitrary in terms of a function arbitraryTree:

```
instance Arbitrary Tree where
  arbitrary = sized arbitraryTree.
```

This function takes as its argument an upper bound for the size of the tree to generate.

```
arbitraryTree    :: Int → Gen Tree
arbitraryTree 0 = arbitraryLeaf
arbitraryTree n = frequency
                    [(1, arbitraryLeaf),
                      (4, arbitraryNode n)]
```

If the upper bound is zero, we generate a leaf; otherwise, we randomly choose between a leaf and a node. In the generator for nodes, the upper bound is enforced by dividing it by two before passing it on to the generators for subtrees:

```
arbitraryNode    :: Int → Gen Tree
arbitraryNode n = do l ← arbitraryTree (n `div` 2)
                     x ← arbitrary
                     r ← arbitraryTree (n `div` 2)
                     return (Node l x r).
```

QuickCheck varies the upper bound on the size of test data between different test cases; it begins testing each property with small values, increasing the likelihood of falsifying a property with a simple counter example.

We have now seen how axioms from algebraic specifications can be encoded as QuickCheck properties and how random test data for properties quantifying over values of custom datatypes can be generated. This gives us the building blocks for subjecting concrete implementations of abstract datatypes in Haskell to random testing. In the next section, we will discuss a case study that highlights a common pitfall related related to the choice of properties to check for.

4. DEALING WITH OPERATION INVARIANCE

In this section, we discuss an implementation of FIFO queues as they were specified in Figure 2 in Section 2 and how it can be tested with QuickCheck. The implementation is, in fact, incorrect (Section 4.1); however, testing it against a set of properties that was derived naïvely from the specification fails to uncover this (Section 4.2) as this set fails to account for the hidden assumption that all queue operations are invariant under queue equality (Section 4.3).

Extending the set with explicit invariance properties then indeed allows for tests that make the bug in our queue implementation manifest (Section 4.4).

4.1 Batched Queues

Our implementation of queues follows what is perhaps the most commonly used technique for implementing efficient FIFO queues in a functional setting. That is, queues are represented by two lists: one containing the front elements of the queue and one containing the rear elements in reverse order.

```
data Queue = BQ [Int] [Int] deriving Show
```

These so-called batched queues allow for all queue operations to take $O(1)$ amortised time provided that queues are used in a single-threaded fashion [14, 20]. To achieve these bounds, we maintain the invariant that the front list is empty only if the rear list is—i.e., if the queue as a whole is empty. To simplify enforcement of this invariant we make sure that, in our implementation of the queue operations, we never use the constructor BQ directly to create values of type Queue but instead delegate queue creation to a so-called *smart constructor* function bq.

```
bq      :: [Int] → [Int] → Queue
bq [] r = BQ (reverse r) []
bq f  r = BQ f r
```

If bq is called with an empty front list, it reverses the rear list and uses the obtained list as the front and the empty list as the rear list of the constructed queue. If it is called with a nonempty front list, it keeps the front and rear list as they were provided.

Then, the empty queue is created simply by calling bq with two empty lists.

```
empty :: Queue
empty = bq [] []
```

To enqueue an element x into a batched queue BQ f r consisting of a front list f and a rear list r, we put x at the head of r and thus call bq with arguments f and x : r.

```
enqueue              :: Int → Queue → Queue
enqueue x (BQ f r) = bq f (x : r)
```

Dequeueing is done by dropping the head element of the front list.

```
dequeue            :: Queue → Queue
dequeue (BQ f r) = bq (tail f) r
```

As the invariant guarantees that an empty front list implies an empty rear list, testing for the emptiness of a queue reduces to assessing the emptiness of the front list.

```
isEmpty            :: Queue → Bool
isEmpty (BQ f r) = null f
```

It remains to implement the operation front that selects the oldest element in the queue. Note that a correct implementation would project the head element of the front list. However, for the sake of the discussion, we instead introduce a version that erroneously selects the last element of the front list.

```
front             :: Queue → Int
front (BQ f r) = last f    -- incorrect!
```

One hopes of course that this error is detected once our queue implementation is tested for conformance to the queue axioms from Figure 2.

4.2 Testing Against the Queue Axioms

To test our implementation of FIFO queues, we first encode the axioms from Figure 2 as QuickCheck properties.

```
q1 :: Property
q1 = property (isEmpty empty)

q2 :: Property
q2 = property (λx q → ¬ (isEmpty (enqueue x q)))

q3 :: Property
q3 = property (λx → front (enqueue x empty) ≡ x)

q4 :: Property
q4 = property (λx q → ¬ (isEmpty q) ⇏
       front (enqueue x q) ≡ front q)

q5 :: Property
q5 = property (λx →
       dequeue (enqueue x empty) ≡ empty)

q6 :: Property
q6 = property (λx q → ¬ (isEmpty q) ⇏
       dequeue (enqueue x q) ≡ enqueue x (dequeue q))
```

Note that, rather than writing isEmpty empty ≡ True for property q1, we apply the Boolean axioms from Figure 1 and simply write isEmpty empty. Likewise, in the side conditions of the properties q4 and q6, we write ¬ (isEmpty q) rather than isEmpty q ≡ False.

The encodings of the queue axioms in QuickCheck make use of Haskell's equality-operator (≡),

```
(≡) :: Eq a ⇒ a → a → Bool,
```

and hence require our type Queue to be in the class Eq. Providing the corresponding instance declaration, we need to make explicit what it means for two queues to be equal. To this end, note that our implementation allows for multiple different physical representations of the same logical queue. For instance, the queue containing the elements 2, 3, and 5 can be represented by any of the batched queues BQ [2,3,5] [], BQ [2,3] [5], and BQ [2] [5,3]. To have this reflected in the Eq instance for Queue, we define a function toList that maps a batched queue to a list of its elements.

```
toList            :: Queue → [Int]
toList (BQ f r) = f ++ reverse r
```

Testing two queues for equality then reduces to comparing the corresponding lists of elements:

```
instance Eq Queue where
  q1 ≡ q2 = toList q1 ≡ toList q2.
```

That is, two queues are considered equal if they contain the same elements in the same order. Here, toList fulfills the rôle of an *abstraction function* [13] for batched queues.

We are almost ready for subjecting our queue implementation to property-based testing against the axioms from Figure 2. The only piece still missing is an instance of the class Arbitrary for Queue, so that QuickCheck knows how to randomly generate test cases for properties that quantify over queues.

```
instance Arbitrary Queue where
  arbitrary = do f ← arbitrary
                 r ← arbitrary
                 return (bq f r)
```

Here, we simply generate random front and rear lists and create a batched queue by supplying these lists as arguments to the smart constructor bq. With that, we are all set to start testing our queue implementation.

Recall that we hope—or expect—that testing against the QuickCheck properties q1 to q6, which we derived from what we consider a complete specification of FIFO queues, uncovers the programming error that was made in the definition of the function front for selecting the oldest element in a queue. However, if we run QuickCheck on our properties, it disappointingly reports

```
+++ OK, passed 100 tests.
+++ OK, passed 100 tests.
+++ OK, passed 100 tests.
+++ OK, passed 100 tests.
+++ OK, passed 100 tests.
+++ OK, passed 100 tests.
```

4.3 Invariance

What went wrong? Clearly our implementation has a bug, but testing against the properties that we derived from the queue axioms fails to bring it to light. And, as it turns out, no matter how often we rerun QuickCheck on these properties and whatever test data we use for testing, it will always be reported that no counter examples were found. In fact, one can even *prove* that the properties, as we have formulated them, hold for our implementation of queues. Does that mean that our specification was not complete after all? Does it mean that we need additional properties? And, if so, how do we know which properties we need to add and, moreover, how do we know when we have added enough properties?

Consider for example adding the following property.

```
q7 :: Property
q7 = property (λx y z →
       front (dequeue (enqueue x (enqueue y
         (enqueue z empty)))) ≡ y)
```

A moment's reflection should be enough to convince ourselves that this is indeed a property that we expect to hold for correctly implemented queues, but also one that, due to our incorrect implementation of front, is not satisfied by our batched queues. Indeed, running QuickCheck on q7 almost immediately yields

```
*** Failed! Falsifiable (after 2 tests):
0
1
0
```

as front (dequeue (enqueue 0 (enqueue 1 (enqueue 0 empty)))) evaluates to 0 rather than 1.

Interestingly, q7 is a representation of one of the theorems that we, by equational reasoning, proved to hold for all specification-abiding implementations of FIFO queues. And indeed, as it turns out, the problem with our implementation of queues is not so much that it violates the individual axioms of our algebraic specification, but rather that it fails to provide for an even more fundamental aspect of the framework that we laid out in Section 2. That is, our implementation is lacking a consistent notion of equality between queues. To appreciate the importance of such a notion, note that it is equality between expressions that we use to define the axioms of an algebraic specification. Moreover, it is equality that is—of

course—at the very basis of our use of equational reasoning when proving theorems about abstract datatypes. In particular, to have a stable basis for equational reasoning, one requires an abstract datatype's notion of equality to be compatible with its operations. That is, the operations should be *invariant* under equality. Formally, an n-ary operation h is invariant under equality if $x_i = y_i$ (for $1 \leqslant i \leqslant n$) implies that $h(x_1, \cdots, x_n) = h(y_1, \cdots, y_n)$. Indeed, it is invariance that justifies "substituting equals for equals" in equational reasoning.

Let us now revisit our faulty implementation of queues. In our QuickCheck renderings of the queue axioms, equality between queues manifests itself by means of Haskell's overloaded $(\equiv)$-operator and hence by the Eq-instance that we provided for our type Queue of batched queues. Yet, our function front fails to be invariant under $(\equiv)$. Consider for example the following batched queues.

```
qA :: Queue
qA = BQ [2,3] [5]

qB :: Queue
qB = BQ [2] [5,3]
```

We have that $\mathsf{qA} \equiv \mathsf{qB}$, but not that $\mathsf{front\ qA} \equiv \mathsf{front\ qB}$, as the expression front qA reduces—erroneously—to 3 while front qB reduces to 2. Hence, our queue implementation failing to satisfy the specification from Section 2 is evinced in it not meeting the hidden requirement from that specification, i.e., that all queue operations be invariant under the equality relation implied by the queue axioms.

4.4 Testing for Invariance

We have now established that, for our QuickCheck properties to match the complete specification of FIFO queues, they should cover not only the axioms but also the invariance requirements that emerge from those. But how do we achieve this? At first glance, it may seem that an additional set of simple side-conditioned properties will do. For instance, to test for the $(\equiv)$-invariance of front, we can define

```
qq :: Property
qq = property (λq q' → q ≡ q' ∧ ¬ (isEmpty q) ⤇
        front q ≡ front q').
```

However, if we try to derive and execute test cases from this property, QuickCheck responds with

```
*** Gave up! Passed only 1 test.
```

What has happened? Recall that when QuickCheck tests against a property with a side condition, it sets out to generate 100 test cases that satisfy the side condition. In this case that means that QuickCheck has to keep generating pairs of random queues q and q' until it has found 100 for which $\mathsf{q} \equiv \mathsf{q}'$. But the chance of two randomly generated queues turning out to be equal is of course very slim and so it would take a very long time for QuickCheck to generate enough suitable test cases. So, after a sufficiently large number of failed attempts, QuickCheck gives up, refraining from drawing any conclusions about the validity of the property at hand.

As a solution to this problem, we need to guide QuickCheck into generating appropriate test cases. To this end, we introduce a new type constructor Equiv.

```
data Equiv a = a :≡: a deriving Show
```

The idea is that values of type Equiv a contain two a-typed values x and y such that $\mathsf{x} \equiv \mathsf{y}$. For example, an acceptable value of type Equiv Queue would be BQ [2,3] [5] :≡: BQ [2] [5,3].

Next, we use the constructor $(:\equiv:)$ in a set of additional properties for queue-operation invariance:

```
qq1 :: Property
qq1 = property (λx (q :≡: q') →
        enqueue x q ≡ enqueue x q')

qq2 :: Property
qq2 = property (λ(q :≡: q') → isEmpty q ≡ isEmpty q')

qq3 :: Property
qq3 = property (λ(q :≡: q') → ¬ (isEmpty q) ⤇
        front q ≡ front q')

qq4 :: Property
qq4 = property (λ(q :≡: q') → ¬ (isEmpty q) ⤇
        dequeue q ≡ dequeue q').
```

That is, rather than abstracting over two Queue-typed values as in qq, the properties qq1 to qq4 abstract over single values of type Equiv Queue.

It remains to provide generators for Equiv-values. For this, we first define an auxiliary type class Concrete of concrete implementations of abstract datatypes.

```
class Arbitrary (Model a) ⇒ Concrete a where
  type Model a
  from :: Model a → Gen a
```

Here, Model is an *associated type synonym* [5]. That is, every type a in the class Concrete comes with an associated type Model a, for which a generator already exists, and a function from that takes models to generators. We require that all values produced by a generator for a single model belong to the same $(\equiv)$-induced equivalence class.

For batched queues, we use ordered element lists as models.

```
instance Concrete Queue where
  type Model Queue = [Int]
  from xs = do i ← choose (0, length xs − 1)
               let (xs1, xs2) = splitAt i xs
               return (bq xs1 (reverse xs2))
```

The function choose from the QuickCheck library,

```
choose :: (Int, Int) → Gen Int,
```

produces a generator for integer values from a specified range. Hence, the generator that we derive from an element list randomly splits the list in a front and a rear part and produces a queue accordingly.

Finally, we add all types Equiv a with the type argument a a member of Concrete to the class Arbitrary of types that come with generators.

```
instance Concrete a ⇒ Arbitrary (Equiv a) where
  arbitrary = do z ← arbitrary
                 x ← from z
                 y ← from z
                 return (x :≡: y)
```

These Arbitrary-provided generators proceed by first generating a model z and then invoking the generator produced by from z to obtain two values x and y with $\mathsf{x} \equiv \mathsf{y}$, which are passed as arguments to the constructor :≡:.

With that, we are set to run QuickCheck on our four invariance properties. Doing so by executing

```
quickCheckMany [qq1, qq2, qq3, qq4]
```

now helpfully yields:

```
+++ OK, passed 100 tests.
+++ OK, passed 100 tests.
*** Failed! Falsifiable (after 4 tests):
BQ [-1,-2] [2] :==: BQ [-1] [2,-2]
+++ OK, passed 100 tests.
```

That is, QuickCheck confirms that, for our queue implementation, the invariance property for front does not hold and, in the process, provides us with a counter example that may serve as a starting point for debugging.

So—where are we now? We have shown that, just naïvely transcribing the axioms from an algebraic specification for an abstract datatype into QuickCheck properties does not give us a set of tests that completely cover what is required for any theorems derived by equational reasoning to apply to implementations that pass the tests. Fortunately, rather than having to add seemingly ad-hoc properties (such as q7 above), for obtaining a complete set of properties, it suffices to systematically complement the axiom-derived properties with properties that state the invariance of the datatype's operations under equality.

5. RELATED WORK

The problem studied in the this paper was inspired by a recent publication of Koopman et al. [17] in which the authors state that "in practice it appears to be rather hard to develop a sufficiently strong set of properties to spot all errors" in the realisation of an abstract datatype and conclude that such a set of properties cannot be derived without studying the internals of the implementation. Their motivating example is exactly the bug in our implementation of FIFO queues; they do not, however, give an account of why testing against a naïvely derived set of queue properties fails to detect this bug. We have shown that it is quite well possible to systematically derive a set of properties that is strong enough to uncover bugs like the one from the example. Moreover, this set can be derived from just the specification of an abstract datatype and, hence, doing so requires no knowledge about the internals of concrete implementations.

Arts et al. [2] discuss an alternative approach to property-based random testing of datatypes. Rather than formulating the properties of the datatype directly, they create a model of the datatype that is simpler than the datatype itself (cf. our associated type Model) and express the required functionality of the datatype in terms of the model. A disadvantage of this approach is that all subsequent equational reasoning based on these properties is also in terms of the model rather than directly in terms of the datatype.

Jeuring et al. [15] present a framework for testing type-class instances against type-class laws. A considerable part for their framework deals with generating data for equality tests. However, this part exclusively deals with performing equality tests rather than, as is required in our setup, generating possibly different concrete values that represent the same logical datatype value.

One way of gaining insight into whether a set of tests reliably assesses the quality of an implementation is by pairing a testing tool such as QuickCheck with a tool that measures *code coverage* such as HPC [10]. However, even insisting on full coverage does not protect test engineers from testing against what is essentially an incomplete set of properties.

In the present paper, we have focused on testing purely functional implementations of abstract datatypes. The problem of not taking into account operation invariance however also appears in an imperative setting; see, for example, Gannon et al. [8].

6. CONCLUSIONS AND FURTHER WORK

We have highlighted a common pitfalls in subjecting implementations of purely functional abstract datatypes to property-based random testing and showed how to systematically avoid it. In particular, we have outlined a methodology for deriving from algebraic specifications properties that account for operation invariance. This methodology was presented in the context of a small and straightforward implementation of a relatively simple abstract datatype; it remains to assess and quantify its impact in the context of one or more larger case studies and real-world applications.

While the suggested invariance properties can be formulated without any specific knowledge about the concrete implementation of a datatype, the approach that we propose for arriving at a complete set of properties to test realisations of abstract datatypes against does put some responsibility on the shoulders of implementors. That is, the use of the type constructor Equiv in properties that state operation invariance requires implementors to make sure that their implementations come with a suitable instance of the class Concrete. Providing such an instance of course calls for some knowledge about the internals of a datatype—in particular, the implementation of the method from needs to be compatible with how equality is defined for the datatype. Interestingly, this required compatibility can itself be captured by a QuickCheck property. Ideally, however, the process of deriving a complete set of properties from an algebraic specification would be automated—for example, by having an (embedded) domain-specific language for algebraic specifications together with functionalilty for generating QuickCheck properties from it.

Acknowledgements

The problem of seemingly complete sets of properties turning out to be inadequate for finding certain bugs in implementations of abstract datatypes by means of property-based random testing was brought to the author's attention by Pieter Koopman during the 2012 Dutch Functional Programming Day.

Alex Gerdes, Luuk Mallens, Alexey Rodriguez Yakushev, Alexandru Şutîi, Andrei Terechko, and four anonymous reviewers gave many helpful comments on previous versions of this paper.

References

[1] Thomas Arts, John Hughes, Joakim Johansson, and Ulf T. Wiger. Testing telecoms software with Quviq Quickcheck. In Marc Feeley and Philip W. Trinder, editors, *Proceedings of the 2006 ACM SIGPLAN Workshop on Erlang, Portland, Oregon, USA, September 16, 2006*, pages 2–10. ACM Press, 2006.

[2] Thomas Arts, Laura M. Castro, and John Hughes. Testing Erlang data types with Quviq Quickcheck. In Soon Tee Teoh and Zoltán Horváth, editors, *Proceedings of the 7th ACM SIGPLAN workshop on ERLANG, Victoria, BC, Canada, September 27, 2008*, pages 1–8. ACM Press, 2008.

[3] Jan A. Bergstra. *Algebraic specification*. ACM Press, New York, NY, USA, 1989.

[4] Richard Bird. *Introduction to Functional Programming using Haskell*. Prentice Hall, London, England, UK, 2nd edition, 1998.

[5] Manuel M. T. Chakravarty, Gabriele Keller, and Simon Peyton Jones. Associated type synonyms. In Olivier Danvy and Benjamin C. Pierce, editors, *Proceedings of the 10th ACM*

SIGPLAN International Conference on Functional Programming, ICFP 2005, Tallinn, Estonia, September 26–28, 2005, pages 241–253. ACM Press, 2005.

[6] Koen Claessen and John Hughes. QuickCheck: A lightweight tool for random testing of Haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00), Montreal, Canada, September 18–21, 2000*, pages 268–279. ACM Press, 2000.

[7] Koen Claessen and John Hughes. Testing monadic code with QuickCheck. In *Proceedings of the ACM SIGPLAN Workshop on Haskell, Pittsburgh, Pennsylvania, 2002*, pages 65–77. ACM Press, 2002.

[8] John D. Gannon, Paul R. McMullin, and Richard Hamlet. Data-abstraction implementation, specification, and testing. *ACM Transactions on Programming Languages and Systems*, 3(3):211–223, 1981.

[9] Marie-Claude Gaudel and Pascale Le Gall. Testing data types implementations from algebraic specifications. In Robert M. Hierons, Jonathan P. Bown, and Mark Harman, editors, *Formal Methods and Testing, An Outcome of the FORTEST Network, Revised Selected Papers*, volume 4949 of *Lecture Notes in Computer Science*, pages 209–239. Springer-Verlag, 2008.

[10] Andy Gill and Colin Runciman. Haskell program coverage. In Gabriele Keller, editor, *Proceedings of the 2007 ACM SIGPLAN workshop on Haskell, Freiburg, Germany, September 30, 2007*, pages 1–12. ACM Press, 2007.

[11] John V. Guttag and James J. Horning. The algebraic specification of abstract data types. *Acta Informatica*, 10:27–52, 1978.

[12] Richard Hamlet. Random testing. In John J. Marciniak, editor, *Encyclopedia of Software Engineering*, pages 970–978. Wiley, New York, NY, USA, 1994.

[13] C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1:271–281, 1972.

[14] Robert Hood and Robert Melville. Real-time queue operations in pure LISP. *Information Processing Letters*, 13(2): 50–54, 1981.

[15] Johan Jeuring, Patrik Jansson, and Cláudio Amaral. Testing type class laws. In Janis Voigtländer, editor, *Proceedings of the 5th ACM SIGPLAN Symposium on Haskell, Haskell 2012, Copenhagen, Denmark, 13 September 2012*, pages 49–60. ACM Press, 2012.

[16] Pieter W. M. Koopman, Artem Alimarine, Jan Tretmans, and Rinus Plasmeijer. Gast: Generic automated software testing. In Ricardo Peña and Thomas Arts, editors, *Implementation of Functional Languages, 14th International Workshop, IFL 2002, Madrid, Spain, September 16–18, 2002, Revised Selected Papers*, volume 2670 of *Lecture Notes in Computer Science*, pages 84–100. Springer-Verlag, 2003.

[17] Pieter W. M. Koopman, Peter Achten, and Rinus Plasmeijer. Model based testing with logical properties versus state machines. In Andy Gill and Jurriaan Hage, editors, *Implementation and Application of Functional Languages, 23rd International Symposium, IFL 2011, Lawrence, KS, USA, October 3–5, 2011, Revised Selected Papers*, volume 7257 of *Lecture Notes in Computer Science*, pages 116–133. Springer-Verlag, 2012.

[18] Barbara Liskov and Stephan N. Zilles. Programming with abstract data types. In *Proceedings of the ACM SIGPLAN Symposium on Very High Level Languages*, pages 50–59. ACM Press, 1974.

[19] Robin Milner, Mads Tofte, Robin Harper, and David B. MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, Cambridge, MA, USA, 1997.

[20] Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, Cambridge, England, U.K., 1999.

[21] Simon Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, Cambridge, England, U.K., 2003.

[22] Rinus Plasmeijer and Marko van Eekelen. Concurrent Clean language report—version 1.3. Technical Report CSI-R9816, University of Nijmegen, 1998.

[23] Philip Wadler. A critique of Abelson and Sussman or why calculating is better than scheming. *SIGPLAN Notices*, 22(3): 83–94, 1987.

[24] Martin Wirsing. Algebraic specification. In Jan Van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, pages 675–788. The MIT Press, Cambridge, MA, USA, 1990.