How to Specify It!: A Guide to Writing Properties of Pure Functions

How to Specify it!

A Guide to Writing Properties of Pure Functions.

John Hughes

Chalmers University of Technology and Quviq AB, Göteborg, Sweden.

Abstract. Property-based testing tools test software against a *specification*, rather than a set of examples. This tutorial paper presents five generic approaches to writing such specifications (for purely functional code). We discuss the costs, benefits, and bug-finding power of each approach, with reference to a simple example with eight buggy variants. The lessons learned should help the reader to develop effective property-based tests in the future.

1 Introduction

Property-based testing (PBT) is an approach to testing software by defining general properties that ought to hold of the code, and using (usually randomly) generated test cases to test that they do, while reporting minimized failing tests if they don't. Pioneered by QuickCheck[1] in Haskell [9], the method is now supported by a variety of tools in many programming languages, and is increasingly popular in practice. Searching for "property-based testing" on Youtube finds many videos on the topic—most of the top 100 recorded at developer conferences and meetings, where (mostly) other people than this author present ideas, tools and methods for PBT, or applications that make use of it. Clearly, property-based testing is an idea whose time has come. But equally clearly, it is also poorly understood, requiring explanation over and over again!

We have found that many developers trying property-based testing for the first time find it difficult to identify *properties to write*—and find the simple examples in tutorials difficult to generalize. This is known as the *oracle problem* [3], and it is common to all approaches that use test case generation.

In this paper, therefore, we take a simple—but non-trivial—example of a purely functional data structure, and present five different approaches to writing properties (invariants, postconditions, metamorphic properties and the preservation of equivalence, inductive properties, and model-based properties). We show the necessity of testing the random generators and shrinkers that property-based testing depends on. We discuss the pitfalls to keep in mind for each kind of property, and we compare and constrast their effectiveness, with the help of eight buggy implementations. We hope that the concrete advice presented here will

[1] `http://hackage.haskell.org/package/QuickCheck`

enable readers to side-step the "where do I start?" question, navigate the zoo of different kinds of property, and quickly derive the benefits that property-based testing has to offer.

2 A Primer in Property-Based Testing

Property-based testing is an approach to random testing pioneered by QuickCheck[2] in Haskell [9], in which universally quantified properties are evaluated as tests in randomly generated cases, and failing tests are simplified by a search for similar, smaller cases. There is no precise definition of the term, however: indeed, MacIver writes[3]

> 'Historically the definition of property-based testing has been "The thing that QuickCheck does".'

The basic idea has been reimplemented many times—Wikipedia in 2019 lists more than 50 implementations, in 36 different programming languages[4], of all programming paradigms. Among contemporary PBT tools are, for example, ScalaCheck [20] for the JVM, FsCheck[5] for .NET, Quviq QuickCheck [2, 16] and Proper [21, 18] for the BEAM, Hypothesis[6] for Python, PrologCheck [1] for Prolog, and SmallCheck [24], SmartCheck [22] and LeanCheck [4] for Haskell, among many others. These implementations vary in quality and features, but the ideas in this paper—while presented using Haskell QuickCheck—should be relevant to a user of any of them.

Suppose, then, that we need to test the *reverse* function on lists. Any developer will be able to write a unit test such as the following:

$$test\_Reverse = reverse\ [1,2,3] == [3,2,1]$$

Here the $(==)$ operator is an equality comparison for use in tests, which displays a message including the compared values if the comparison is *False*.

This test is written in the same form as most test cases worldwide: we apply the function under test (*reverse*) to known arguments ($[1,2,3]$), and then compare the result to a known expected value ($[3,2,1]$). Developers are practiced in coming up with these examples, and predicting expected results. But what happens when we try to write a property instead?

$$prop\_Reverse :: [Int] \to Property$$
$$prop\_Reverse\ xs = reverse\ xs == ???$$

The property is parameterised on xs, which will be randomly generated by QuickCheck; we state a monomorphic type signature explicitly, even though the *reverse* function is polymorphic, to tell QuickCheck what type of test data

[2] `http://hackage.haskell.org/package/QuickCheck`
[3] `https://hypothesis.works/articles/what-is-property-based-testing/`
[4] `https://en.wikipedia.org/wiki/QuickCheck`
[5] `https://fscheck.github.io/FsCheck/`
[6] `https://pypi.org/project/hypothesis/`

to generate. The result type is *Property*, not *Bool*, because this is what (==) returns—*Property*s are not pure booleans, because they can generate diagnostic output, among other things.

The property can clearly test *reverse* in a much wider range of cases than the unit test—*any* randomly generated list, rather than just the list $[1, 2, 3]$—which is a great advantage. But the question is: *what is the expected result?* That is, what should we replace ??? by in the definition above? Since the argument to *reverse* is not known in advance, we cannot precompute the expected result. We could write test code to *predict* it, as in

$$prop\_Reverse :: [Int] \to Property$$
$$prop\_Reverse\ xs = reverse\ xs == predictRev\ xs$$

but *predictRev is not easier to write than reverse*—it is *exactly the same function*!

This is the most obvious approach to writing properties—to replicate the implementation in the test code—and it is deeply unsatisfying. It is both an *expensive* approach, because the replica of the implementation may be as complex as the implementation under test, and of *low value*, because there is a grave risk that misconceptions in the implementation will be replicated in the test code. "Expensive" and "low value" is an unfortunate combination of characteristics for a software testing method!

"Avoid replicating your code in your tests."

We can finesse this problem by rewriting the property so that it does not refer to an expected result, instead checking some *property* of the result. For example, *reverse* is its own inverse:

$$prop\_Reverse :: [Int] \to Property$$
$$prop\_Reverse\ xs = reverse\ (reverse\ xs) == xs$$

Now we can pass the property to QuickCheck, to run a series of random tests (by default 100):

```
*Examples> quickCheck prop_Reverse
+++ OK, passed 100 tests.
```

We have met our goal of testing *reverse* on 100 random lists, but this property is not very strong—if we had accidentally defined

$$reverse\ xs = xs$$

then it would still pass (whereas the unit test above would report a bug).

We can define another property that this *buggy* implementation of *reverse* passes, but the correct definition fails:

$$prop\_Wrong :: [Int] \to Property$$
$$prop\_Wrong\ xs = reverse\ xs == xs$$

Since *reverse* is actually correctly implemented, this allows us to show what happens when a property fails:

```
*Examples> quickCheck prop_Wrong
*** Failed! Falsified (after 5 tests and 3 shrinks):
[0,1]
[1,0] /= [0,1]
```

Here the first line after the failure message shows the value of *xs* for which the test failed ($[0,1]$), while the second line is the message generated by ($==$), telling us that the result of *reverse* (that is, $[1,0]$) was not the expected value ($[0,1]$).

Interestingly, the counterexample QuickCheck reports for this property is almost always $[0,1]$, and occasionally $[1,0]$. These are not the random counterexamples that QuickCheck finds first; they are the result of *shrinking* the random counterexamples via a systematic greedy search for a simpler failing test. Shrinking lists tries to remove elements, and numbers shrink towards zero; the reason we see these two counterexamples is that *xs* must contain at least two different elements to falsify the property, and 0 and 1 are the smallest pair of different integers. Shrinking is one of the most useful features of property-based testing, resulting in counterexamples which are usually easy to debug, because *every part* of the counterexample is relevant to the failure.

Now we have seen the benefits of property-based testing—random generation of very many test cases, and shrinking of counterexamples to minimal failing tests—and the major pitfall: the temptation to replicate the implementation in the tests, incurring high costs for little benefit. In the remainder of this paper, we present systematic ways to define properties *without* falling into this trap. We will (largely) ignore the question of how to generate *effective* test cases—that are good at reaching buggy behaviour in the implementation under test—even though this is an active research topic in its own right (see, for example, the field of *concolic testing* [12, 25]). While generating good test cases is important, in the absence of good properties, they are of little value.

3 Our Running Example: Binary Search Trees

The code we shall develop properties for is an implementation of finite maps (from keys to values) as binary search trees. The definition of the tree type is shown in Figure 1; a tree is either a *Leaf*, or a *Branch* containing a left subtree, a key, a value, and a right subtree. The operations we will test are those that create trees (*nil*, *insert*, *delete* and *union*), and that *find* the value associated with a key in the tree. We will also use auxiliary operations: *toList*, which returns a sorted list of the key-value pairs in the tree, and *keys* which is defined in terms of it. The implementation itself is standard, and is not included here.

Before writing properties of binary search trees, we must define a *generator* and a *shrinker* for this type. We use the definitions in Figure 2, which generate trees by creating a random list of keys and values and inserting them into the empty tree, and shrink trees using a generic method provided by QuickCheck. The type restriction in the definition of *arbitrary* is needed to fix *kvs* to be a

```
data BST k v = Leaf | Branch (BST k v) k v (BST k v)
  deriving (Eq, Show, Generic)

  -- the operations under test
find    :: Ord k ⇒ k → BST k v → Maybe v
nil     :: BST k v
insert :: Ord k ⇒ k → v → BST k v → BST k v
delete :: Ord k ⇒ k       → BST k v → BST k v
union :: Ord k ⇒ BST k v → BST k v → BST k v

  -- auxiliary operations
toList :: BST k v → [(k, v)]
keys   :: BST k v → [k]
```

Fig. 1. The API under test: binary search trees.

```
instance (Ord k, Arbitrary k, Arbitrary v) ⇒ Arbitrary (BST k v) where
  arbitrary = do
    kvs ← arbitrary
    return $ foldr (uncurry insert) nil (kvs :: [(k, v)])
  shrink = genericShrink
```

Fig. 2. Generating and shrinking binary search trees.

list, because *foldr* is overloaded to work over any *Foldable* collection. We shall revisit both these definitions later, but they will do for now.

We need to *fix an instance type* for testing; for the time being, we choose to let both keys and values be integers, and define

```
type Key = Int
type Val  = Int
type Tree = BST Int Int
```

Int is usually an acceptably good choice as an instance for testing polymorphic properties, although we will return to this choice later. In the rest of this article *we omit type signatures on properties* for brevity, although in reality they must be given, to tell QuickCheck to use the types above.

4 Approaches to Writing Properties

4.1 Validity Testing

"Every operation should return valid results."

Many data-structures need to satisfy invariant properties, above and beyond being well-typed, and binary search trees are no exception: the keys in the tree

$$
\begin{aligned}
&\mathit{prop\_NilValid} = \mathit{valid}\ (\mathit{nil} :: \mathit{Tree})\\
&\mathit{prop\_InsertValid}\ k\ v\ t = \mathit{valid}\ (\mathit{insert}\ k\ v\ t)\\
&\mathit{prop\_DeleteValid}\ k\ t = \mathit{valid}\ (\mathit{delete}\ k\ t)\\
&\mathit{prop\_UnionValid}\ t\ t' = \mathit{valid}\ (\mathit{union}\ t\ t')
\end{aligned}
$$

Fig. 3. Validity properties.

should be ordered. In this section, we shall see how to write properties that check that this invariant is preserved by each operation.

We can capture the invariant by the following function:

$$
\begin{aligned}
&\mathit{valid}\ \mathit{Leaf} = \mathit{True}\\
&\mathit{valid}\ (\mathit{Branch}\ l\ k\ \_v\ r) =\\
&\quad \mathit{valid}\ l \wedge \mathit{valid}\ r \wedge\\
&\quad \mathit{all}\ (<k)\ (\mathit{keys}\ l) \wedge \mathit{all}\ (>k)\ (\mathit{keys}\ r)
\end{aligned}
$$

That is, all the *keys* in a left subtree must be less than the key in the node, and all the *keys* in the right subtree must be greater.

This definition is obviously correct, but it is an inefficient implementation of the validity checking function; it is quadratic in the size of the tree in the worst case. A more efficient implementation would exploit the validity of the left and right subtrees, and compare only the *last* key in the left subtree, and the *first* key in the right subtree, against the key in a *Branch* node. But the equivalence of these two definitions depends on reasoning, and we prefer to *avoid reasoning that is not checked by tests*—if it turns out to be wrong, or is invalidated by later changes to the code, then tests using the more efficient definition might fail to detect some bugs. Testing that two definitions are equivalent would require testing a property such as

$$\mathit{prop\_ValidEquivalent}\ t = \mathit{valid}\ t == \mathit{fastValid}\ t$$

and to do so, we would need a generator that can produce *both valid and invalid trees*, so this is not a straightforward extension. We prefer, therefore, to use the obvious-but-inefficient definition, at least initially. The trees we are generating are relatively small, so quadratic complexity is not a problem.

"Test your tests."

Now it is straightforward to define properties that check that every operation that constructs a tree, constructs a valid one (see Figure 3). However, these properties, by themselves, do not provide good testing for validity. To see why, let us plant a bug in *insert*, so that it creates duplicate entries when inserting a key that is already present (bug (2) in section 5). *prop_InsertValid* fails as it should, but so do *prop_DeleteValid* and *prop_UnionValid*:

```
=== prop_InsertValid from BSTSpec.hs:19 ===
*** Failed! Falsified (after 6 tests and 8 shrinks):
0
```

```
0
Branch Leaf 0 0 Leaf

=== prop_DeleteValid from BSTSpec.hs:22 ===
*** Failed! Falsified (after 8 tests and 7 shrinks):
0
Branch Leaf 1 0 (Branch Leaf 0 0 Leaf)

=== prop_UnionValid from BSTSpec.hs:25 ===
*** Failed! Falsified (after 7 tests and 9 shrinks):
Branch Leaf 0 0 (Branch Leaf 0 0 Leaf)
Leaf
```

Thus, at first sight, there is nothing to indicate that the bug is in *insert*; all of *insert*, *delete* and *union* can return invalid trees! However, *delete* and *union* are *given* invalid trees as inputs in the tests above, and we cannot expect them to return valid trees in this case, so these reported failures are "false positives."

The problem here is that the *generator* for trees is producing invalid ones (because it is defined in terms of *insert*). We could add a precondition to each property, requiring the tree to be valid, as in:

$$prop\_DeleteValid\ k\ t = valid\ t \Longrightarrow valid\ (delete\ k\ t)$$

which would discard invalid test cases (not satisfying the precondition) without running them, and thus make the properties pass. This is potentially inefficient (we might spend much of our testing time discarding test cases), but it is also really just applying a sticking plaster: what we *want* is that all generated trees should be valid! We can test this by defining an additional property:

$$prop\_ArbitraryValid\ t = valid\ t$$

which at first sight seems to be testing that *all* trees are valid, but in fact tests that *all trees generated by the Arbitrary instance* are valid. If this property fails, then it is the generator that needs to be fixed—there is no point in looking at failures of other properties, as they are likely caused by the failing generator.

Usually the generator for a type *is* intended to fulfill its invariant, but—as in this case—is defined independently. A property such as *prop_ArbitraryValid* is essential to check that these definitions are mutually consistent.

It is also possible for the *shrink* function to violate a datatype invariant. For this reason, we should also write a property requiring all the smaller test cases returned by *shrink* to be valid:

$$prop\_ShrinkValid\ t = all\ valid\ (shrink\ t)$$

Unfortunately, with the definitions given so far, this property fails:

```
=== prop_ShrinkValid from BSTSpec.hs:28 ===
*** Failed! Falsified (after 6 tests and 8 shrinks):
Branch (Branch Leaf 0 0 Leaf) 0 1 Leaf
```

Inspection reveals that this argument to *shrink* is *already* invalid—and so it is no surprise that *shrink* might include invalid trees in its result. The problem here is that, even though QuickCheck initially found a *valid* tree with an invalid shrink, *it shrunk the test case before reporting it using the invalid shrink function*, resulting in an invalid tree with invalid shrinks. What we want to see, when debugging, is a *valid* tree with an invalid shrink; to ensure that this is what QuickCheck reports, we must add a $valid\ t \Longrightarrow$ precondition to this property. This precondition should always hold for a randomly generated test (provided *arbitrary* is correct), but prevents such a test case being shrunk to an invalid case when the property fails; thus, we avoid the potential inefficiency discussed on page 7, whereby preconditions cause many randomly generated tests to be discarded.

We can also reexpress the check in a slightly different, but equivalent form, so that when a failing test is reported we see both the valid original tree, and the invalid tree that it is shrunk to:

$$prop\_ShrinkValid\ t = valid\ t \Longrightarrow filter\ (not \circ valid)\ (shrink\ t) == [\,]$$

With these changes the failing test is easy to interpret:

```
=== prop_ShrinkValid from BSTSpec.hs:28 ===
*** Failed! Falsified (after 7 tests and 8 shrinks):
Branch (Branch Leaf 0 0 Leaf) 1 0 Leaf
[Branch (Branch Leaf 0 0 Leaf) 0 0 Leaf] /= []
```

We see that shrinking the key 1 to 0 invalidated the invariant.

We must thus redefine shrinking for the *BST* type to enforce the invariant. There are various ways of doing so, but perhaps the simplest is to continue to use *genericShrink*, but discard smaller trees where the invariant is broken:

$$shrink = filter\ valid \circ genericShrink$$

This section illustrates well the importance of *testing our tests*; it is vital to test generators and shrinkers *independently* of the operations under test, because a bug in either can result in many very-hard-to-debug failures in other properties.

> **Summary:** *Validity testing consists of defining a function to check the invariants of your datatypes, writing properties to test that your generators and shrinkers only produce valid results, and writing a property for each function under test that performs a single random call, and checks that the return value is valid.*

Validity properties are important to test, whenever a datatype has an invariant, but they are far from sufficient by themselves. Consider this: *if every function returning a BST were defined to return nil in every case*, then all the properties written so far would pass. *insert* could be defined to delete the key instead, or *union* could be defined to implement set difference—as long as the invariant is preserved, the properties will still pass. Thus, we must move on to properties that better capture the *intended behaviour* of each operation.

4.2 Postconditions

"Postconditions relate return values to arguments of a single call."

A *postcondition* is a property that should be *True* after a call, or (equivalently, for a pure function) *True* of its result. Thus, we can define properties by asking ourselves "What should be *True* after calling f?". For example, after calling *insert*, then we should be able to *find* the key just inserted, and any previously inserted keys with unchanged values.

$$\begin{aligned}&\mathit{prop\_InsertPost}\ k\ v\ t\ k' =\\ &\quad \mathit{find}\ k'\ (\mathit{insert}\ k\ v\ t) == \textbf{if}\ k \equiv k'\ \textbf{then}\ \mathit{Just}\ v\ \textbf{else}\ \mathit{find}\ k'\ t\end{aligned}$$

One may wonder whether it is best to parameterize this property on *two different* keys, or just on one: after all, for the type chosen, independently generated keys k and k' are equal in only around 3.3% of cases, so most test effort is devoted to checking the **else**-branch in the property, namely that *other* keys than the one inserted are preserved. However, using the *same* key for k and k' would weaken the property drastically—for example, an implementation of *insert* that discarded the original tree entirely would still pass. Moreover, nothing hinders us from defining and testing a specialized property:

$$\mathit{prop\_InsertPostSameKey}\ k\ v\ t = \mathit{prop\_InsertPost}\ k\ v\ t\ k$$

Testing this property devotes *all* test effort to the case of finding a newly inserted key, but does not require us to replicate the code in the more general postcondition.

We can write similar postconditions for *delete* and *union*; writing the property for *union* forces us to specify that *union* is left-biased (since union of finite maps cannot be commutative).

$$\mathit{prop\_UnionPost}\ t\ t'\ k = \mathit{find}\ k\ (\mathit{union}\ t\ t') == (\mathit{find}\ k\ t <|> \mathit{find}\ k\ t')$$

(where $(<|>)$ is the operation that chooses one of two *Maybe* values, choosing the first argument if it is of the form $\mathit{Just}\ x$, and the second argument otherwise).

Postconditions are not always as easy to write. For example, consider a postcondition for *find*. The return value is either *Nothing*, in case the key is not found in the tree, or $\mathit{Just}\ v$, in the case where it is present with value v. So it seems that, to write a postcondition for *find*, we need to be able to determine whether a given key is present in a tree, and if so, with what associated value. *But this is exactly what find does!* So it seems we are in the awkward situation discussed in the introduction: in order to test *find*, we need to reimplement it.

We can finesse this problem using a very powerful and general idea, that of *constructing a test case whose outcome is easy to predict.* In this case, we *know* that a tree must contain a key k, *if we have just inserted it.* Likewise, we know that a tree cannot contain a key k, if we have just deleted it. Thus we can write *two* postconditions for *find*, covering the two cases:

$$\begin{aligned}&\mathit{prop\_FindPostPresent}\ k\ v\ t = \mathit{find}\ k\ (\mathit{insert}\ k\ v\ t) == \mathit{Just}\ v\\ &\mathit{prop\_FindPostAbsent}\ k\ t = \mathit{find}\ k\ (\mathit{delete}\ k\ t) == \mathit{Nothing}\end{aligned}$$

But there is a risk, when we write properties in this form, that we are only testing very special cases. Can we be certain that *every* tree, containing key k with value v, can be expressed in the form *insert* k v t? Can we be certain that every tree *not* containing k can be expressed in the form *delete* k t? If not, then the postconditions we wrote for *find* may be less effective tests than we think.

Fortunately, for this data structure, every tree *can* be expressed in one of these two forms, because inserting a key that is already present, or deleting one that is not, is a no-op. We express this as another property to test:

```
prop_InsertDeleteComplete k t = case find k t of
  Nothing → t == delete k t
  Just v → t == insert k v t
```

Summary: *A postcondition tests a single function, calling it with random arguments, and checking an expected relationship between its arguments and its result.*

4.3 Metamorphic Properties

"Related calls return related results."

Metamorphic testing is a successful approach to the oracle problem in many contexts [7]. The basic idea is this: even if the expected result of a function call such as *insert* k v t may be difficult to predict, we may still be able to express an expected *relationship* between this result, and the result of a related call. In this case, if we insert an additional key into t before calling *insert* k v, then we expect the additional key to appear in the result also. We formalize this as the following *metamorphic property*:

```
prop_InsertInsert (k, v) (k', v') t =
  insert k v (insert k' v' t) == insert k' v' (insert k v t)
```

A metamorphic property, like this one, (almost) always *relates two calls* to the function under test. Here the function under test is *insert*, and the two calls are *insert* k v t and *insert* k v (*insert* k' v' t). The latter is constructed by *modifying* the argument, in this case also using *insert*, and the property expresses an expected relationship between the values of the two calls. Metamorphic testing is a fruitful source of property ideas, since if we are given $O(n)$ operations to test, each of which can also be used as a modifier, then there are potentially $O(n^2)$ properties that we can define.

However, the property above is not true: testing it yields

```
=== prop_InsertInsert from BSTSpec.hs:78 ===
*** Failed! Falsified (after 2 tests and 5 shrinks):
(0,0)
(0,1)
Leaf
Branch Leaf 0 0 Leaf /= Branch Leaf 0 1 Leaf
```

This is not surprising. The property states that the order of insertions does not matter, while the failing test case inserts the same key twice with different values—of course the order of insertion matters in this case, because "the last insertion wins". A first stab at a metamorphic property may often require correction; QuickCheck is good at showing us what it is that needs fixing. We just need to consider two equal keys as a special case:

$$\begin{array}{l} \mathit{prop\_InsertInsert}\ (k, v)\ (k', v')\ t = \\ \quad \mathit{insert}\ k\ v\ (\mathit{insert}\ k'\ v'\ t) \\ \quad \equiv \\ \quad \textbf{if}\ k \equiv k'\ \textbf{then}\ \mathit{insert}\ k\ v\ t\ \textbf{else}\ \mathit{insert}\ k'\ v'\ (\mathit{insert}\ k\ v\ t) \end{array}$$

Unfortunately, this property *still* fails:

```
=== prop_InsertInsert from BSTSpec.hs:78 ===
*** Failed! Falsified (after 2 tests):
(1,0)
(0,0)
Leaf
Branch Leaf 0 0 (Branch Leaf 1 0 Leaf) /=
Branch (Branch Leaf 0 0 Leaf) 1 0 Leaf
```

Inspecting the two resulting trees, we can see that changing the order of insertion results in trees with *different shapes*, but containing the *same* keys and values. Arguably this does not matter: we should not care what shape of tree each operation returns, provided it contains the right information[7]. To make our property pass, we must make this idea explicit. We therefore define an equivalence relation on trees that is true if they have the same contents,

$$t1 \mathrel{\hat{=}} t2 = \mathit{toList}\ t1 \equiv \mathit{toList}\ t2$$

and re-express the property in terms of this equivalence:

$$\begin{array}{l} \mathit{prop\_InsertInsert}\ (k, v)\ (k', v')\ t = \\ \quad \mathit{insert}\ k\ v\ (\mathit{insert}\ k'\ v'\ t) \\ \quad \mathrel{\hat{=}} \\ \quad \textbf{if}\ k \equiv k'\ \textbf{then}\ \mathit{insert}\ k\ v\ t\ \textbf{else}\ \mathit{insert}\ k'\ v'\ (\mathit{insert}\ k\ v\ t) \end{array}$$

Now, at last, the property passes. (We discuss why we need *both* this equivalence, and structural equality on trees, in section 7).

There is a different way to address the first problem—that the order of insertions *does* matter, when inserting the same key twice. That is to *require* the keys to be different, via a precondition:

$$\begin{array}{l} \mathit{prop\_InsertInsertWeak}\ (k, v)\ (k', v')\ t = \\ \quad k \not\equiv k' \Longrightarrow \mathit{insert}\ k\ v\ (\mathit{insert}\ k'\ v'\ t) \mathrel{\hat{=}} \mathit{insert}\ k'\ v'\ (\mathit{insert}\ k\ v\ t) \end{array}$$

[7] Recall that we have not imposed any *balance condition* on our trees. If we were to repeat this entire exercise for balanced trees, then we would need a stronger invariant to capture the balance condition, but we would still face the same problem in this property, since balance conditions don't require a *unique* tree shape. Both trees in this example are balanced—they are just different balanced representations of the same information.

This lets us keep the property in a simpler form, but is weaker, since it no longer captures that "the last insert wins". We will return to this point later.

We can go on to define further metamorphic properties for *insert*, with different modifiers—*delete* and *union*:

$$
\begin{aligned}
&\mathit{prop\_InsertDelete}\ (k, v)\ k'\ t = \\
&\quad \mathit{insert}\ k\ v\ (\mathit{delete}\ k'\ t) \\
&\quad \mathrel{\hat{=}} \\
&\quad \textbf{if}\ k \equiv k'\ \textbf{then}\ \mathit{insert}\ k\ v\ t\ \textbf{else}\ \mathit{delete}\ k'\ (\mathit{insert}\ k\ v\ t) \\
&\mathit{prop\_InsertUnion}\ (k, v)\ t\ t' = \\
&\quad \mathit{insert}\ k\ v\ (\mathit{union}\ t\ t') \mathrel{\hat{=}} \mathit{union}\ (\mathit{insert}\ k\ v\ t)\ t'
\end{aligned}
$$

and, in a similar way, metamorphic properties for the other functions in the API under test. We derived sixteen different properties in this way, which are listed in Appendix A. The trickiest case is *union* which, as a binary operation, can have *either* argument modified—or both. We also found that some properties could be motivated in more than one way. For example, *prop_InsertUnion* (above) can be motivated as a metamorphic test for *insert*, in which the argument is modified by *union*, or as a metamorphic test for *union*, in which the argument is modified by *insert*. Likewise, the metamorphic tests we wrote for *find* replicated the postconditions we wrote above for *insert*, *delete* and *union*. We do not see this as a problem: that there is more than one way to motivate a property does not make it any less useful, or any harder to come up with!

> **Summary:** *A metamorphic property tests a single function by making (usually) two related calls, and checking the expected relationship between the two results.*

Preservation of Equivalence Now that we have an equivalence relation on trees, we may wonder whether the operations under test *preserve* it. For example, we might try to test whether *insert* preserves equivalence as follows:

$$
\begin{aligned}
&\mathit{prop\_InsertPreservesEquiv}\ k\ v\ t\ t' = \\
&\quad t \mathrel{\hat{=}} t' \Longrightarrow \mathit{insert}\ k\ v\ t \mathrel{\hat{=}} \mathit{insert}\ k\ v\ t'
\end{aligned}
$$

This kind of property is important, since many of our metamorphic properties only allow us to conclude that two expressions are equivalent; to use these conclusions in further reasoning, we need to know that equivalence is preserved by each operation.

Unfortunately, testing the property above does not work; it is very, very unlikely that two randomly generated trees t and t' will be equivalent, and thus almost all generated tests are discarded. To test this kind of property, we need to *generate equivalent pairs of trees* together. We can do so be defining a *type* of equivalent pairs, with a custom generator and shrinker—see Figure 4. This generator constructs two equivalent trees by inserting the *same* list of keys and values in two different orders; the shrinker is omitted for brevity. The properties using this type appear in Figure 5, along with properties to test the new generator and shrinker.

```
data Equivs k v = BST k v :≐: BST k v deriving Show
instance (Arbitrary k, Arbitrary v, Ord k) ⇒ Arbitrary (Equivs k v) where
  arbitrary = do
    kvs ← L.nubBy ((≡) `on` fst) <$> arbitrary
    kvs′ ← shuffle kvs
    return (tree kvs :≐: tree kvs′)
    where tree = foldr (uncurry insert) nil
  shrink (t1 :≐: t2) = ...
```

Fig. 4. Generating equivalent trees.

```
prop_InsertPreservesEquiv k v (t :≐: t′) = insert k v t ≐ insert k v t′
prop_DeletePreservesEquiv k (t :≐: t′) = delete k t ≐ delete k t′
prop_UnionPreservesEquiv (t1 :≐: t1′) (t2 :≐: t2′) = union t1 t2 ≐ union t1′ t2′
prop_FindPreservesEquiv k (t :≐: t′) = find k t == find k t′
prop_Equivs (t :≐: t′) = t ≐ t′
prop_ShrinkEquivs (t :≐: t′) =
  t ≐ t′ ⟹ all (λ(t :≐: t′) → t ≐ t′) (shrink (t :≐: t′))
  where t ≐ t′ = toList t ≡ toList t′
```

Fig. 5. Preservation of equivalence.

4.4 Inductive Testing

"Inductive proofs inspire inductive tests."

Metamorphic properties do not, in general, *completely* specify the behaviour of the code under test. However, in some cases, a subset of metamorphic properties *does* form a complete specification. Consider, for example, the following two properties of *union*:

```
prop_UnionNil1 t = union nil t == t
prop_UnionInsert t t′ (k, v) =
  union (insert k v t) t′ ≐ insert k v (union t t′)
```

We can argue that these two properties characterize the behaviour of *union* precisely (up to equivalence of trees), *by induction on the size of union's first argument*. This idea is due to Claessen [8].

However, there is a hidden assumption in the argument above—namely, that *any non-empty tree t can be expressed in the form insert k v t′, for some smaller tree t′*, or equivalently, that any tree can be constructed using insertions only. There is no reason to believe this *a priori*—it might be that some tree shapes can only be constructed by *delete* or *union*. So, to confirm that these two properties uniquely characterize *union*, we must test this assumption.

One way to do so is to define a function that maps a tree to a list of insertions that recreate it. It is sufficient to insert the key in each node before the keys in its subtrees:

```
insertions Leaf = []
insertions (Branch l k v r) = (k, v) : insertions l ++ insertions r
```

Now we can write a property to check that *every* tree can be reconstructed from its list of insertions:

```
prop_InsertComplete t = t == foldl (flip $ uncurry insert) nil (insertions t)
```

However, this is not sufficient! Recall that the generator we are using, defined in section 3, generates a tree by *performing a list of insertions*! It is clear that any *such* tree can be built using only *insert*, and so the property above can never fail, but what we need to know is that the same is true of trees returned by *delete* and *union*! We must thus define additional properties to test this:

```
prop_InsertCompleteForDelete k t = prop_InsertComplete (delete k t)
prop_InsertCompleteForUnion t t' = prop_InsertComplete (union t t')
```

Together, these properties also justify our choice of generator—they show that we really *can* generate any tree constructible using the tree API. If we could *not* demonstrate that trees returned by *delete* and *union* can also be constructed using *insert*, then we could define a more complex generator for trees that uses all the API operations, rather than just *insert*—a workable approach, but considerably trickier, and harder to tune for a good distribution of test data.

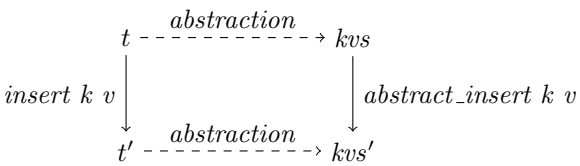
Finally, we note that in these completeness properties, it is vital to check *structural equality* between trees, and not just equivalence. The whole point is to show that *delete* and *union* cannot construct otherwise unreacheable *shapes* of trees, which might provoke bugs in the implementation.

> **Summary:** *Inductive properties relate a call of the function-under-test to calls with smaller arguments. A set of inductive properties covering all possible cases together test the base case(s) and induction step(s) of an inductive proof-of-correctness. If all the properties hold, then we know the function is correct–inductive properties together make up a complete test.*

4.5 Model-based Properties

"Abstract away from details to simplify properties."

In 1972, Hoare published an approach to proving the correctness of data representations [14], by relating them to abstract data using an *abstraction function*. Hoare defines a concrete and abstract implementation for each operation, and then proves that diagrams such as this one commute:



$$
\begin{aligned}
&\mathit{prop\_NilModel} = \mathit{toList}\ (\mathit{nil} :: \mathit{Tree}) == [\,]\\
&\mathit{prop\_InsertModel}\ k\ v\ t =\\
&\quad \mathit{toList}\ (\mathit{insert}\ k\ v\ t) == \mathit{L.insert}\ (k, v)\ (\mathit{deleteKey}\ k\ \$\ \mathit{toList}\ t)\\
&\mathit{prop\_DeleteModel}\ k\ t = \mathit{toList}\ (\mathit{delete}\ k\ t) == \mathit{deleteKey}\ k\ (\mathit{toList}\ t)\\
&\mathit{prop\_UnionModel}\ t\ t' =\\
&\quad \mathit{toList}\ (\mathit{union}\ t\ t') == \mathit{L.sort}\ (\mathit{L.unionBy}\ ((\equiv)\ `\mathit{on}`\ \mathit{fst})\ (\mathit{toList}\ t)\ (\mathit{toList}\ t'))\\
&\mathit{prop\_FindModel}\ k\ t = \mathit{find}\ k\ t == \mathit{L.lookup}\ k\ (\mathit{toList}\ t)\\
&\mathit{deleteKey}\ k = \mathit{filter}\ ((\not\equiv k) \circ \mathit{fst})
\end{aligned}
$$

Fig. 6. Model-based properties.

In this case we abstract trees t (the concrete implementation) as *ordered lists of key–value pairs kvs* (the abstract data), using an abstraction function which is just *toList*. The diagram says that both paths from top left to bottom right should yield the same result: applying the concrete version of insertion to a tree, and then abstracting the result to a list of key–value pairs, yields the *same* list as the abstract version of insertion, applied to the abstracted input. If a similar diagram commutes for every operation in an API, then it follows that any sequence of concrete operations behaves in the same way as the same sequence of abstract ones.

We can use the same idea for testing. Since *Data.List* already provides an insertion function for ordered lists, it is tempting to define

$$\mathit{prop\_InsertModel}\ k\ v\ t = \mathit{toList}\ (\mathit{insert}\ k\ v\ t) == \mathit{L.insert}\ (k, v)\ (\mathit{toList}\ t)$$

(in which *Data.List* is imported under the name L). However, this property fails:

```
*** Failed! Falsified (after 5 tests and 6 shrinks):
0
0
Branch Leaf 0 0 Leaf
[(0,0)] /= [(0,0),(0,0)]
```

The problem is that the insertion function in *Data.List* may create duplicate elements, but *insert* for trees does not. So it is not quite the correct abstract implementation; we can correct this by deleting the key if it is initially present—see the correct properties in Figure 6.

We refer to these properties as "model-based" properties, and we refer to the abstract datatype, in this case an ordered list of keys and values, as the "model". The model can be thought of as a kind of *reference implementation* of the operations under test, though with a much simpler representation. Model-based properties are very powerful: they make up a complete specification of the behaviour of the operations under test, with only a single property per operation. On the other hand, they do require us to construct a model, which in more complex situations may be quite expensive, or may resemble the actual implementation more than is healthy.

Summary: *A model-based property tests a single function by making a single call, and comparing its result to the result of a related "abstract operation" applied to related abstract arguments. An abstraction functions maps the real, concrete arguments and results to abstract values, which we also call the "model".*

4.6 A Note on Generation

Throughout this paper, we have used integers as test data, for both keys and values. This is generally an acceptable choice, although not necessarily ideal. It is useful to *measure the distribution* of test data, to judge whether or not tests are likely to find bugs efficiently. In this case, many properties refer to one or more keys, and a tree, generated independently. We may therefore wonder, how often does such a key actually occur in an independently generated tree?

To find out, we can define a property just for *measurement*. QuickCheck allows properties to label test cases with one or more strings; the labelling strings are collected as tests are run, and their distribution displayed in a table afterwards. In this case, we measure how often k appears in t, and also *where* among the keys of t it appears:

```
prop_Measure k t =
  label (if k ∈ keys t then "present" else "absent") $
  label (if t ≡ nil then "empty" else
    if keys t ≡ [k] then "just k" else
    if (all (⩾ k) (keys t)) then "at start" else
    if (all (⩽ k) (keys t)) then "at end" else
    "middle") $
  True
```

Two tables are generated by testing this property, one for each of the calls of the *label* function. After a million tests, we saw the following distributions:

```
79.1973% absent
20.8027% present

75.0878% middle
 9.6716% at end
 9.6534% at start
 5.1782% empty
 0.4090% just k
```

From the second table, we can see that k appears at the beginning or end of the keys in t about 10% of the time for each case, while it appears somewhere in the middle of the sequences of keys 75% of the time. This looks quite reasonable. On the other hand, *in almost 80% of tests, k is not found in the tree at all!*

For some of the properties we defined, this will result in quite inefficient testing. For example, consider the postcondition for *insert*:

```
prop_InsertPost k v t k' =
  find k' (insert k v t) == if k ≡ k' then Just v else find k' t
```

In almost 80% of tests k' will not be present in t, and since k' is rarely equal to k, then in most of these cases both sides of the equation will be *Nothing*. In effect, we spend most of our effort testing that inserting key k does not insert an *unrelated key* k' into the tree! While this would be a serious bug if it occurred, it seems disproportionate to devote so much test effort to this kind of case.

More reasonable would be to divide our test effort roughly equally between cases in which the given key *does* occur in the random tree, and cases in which it does not. We can achieve this by *changing the generation of keys*. If we choose keys *from a smaller set*, then we will generate equal keys more often. For example, we might define a **newtype** of keys containing a smaller non-negative integer:

```
newtype Key = Key Int deriving (Eq, Ord, Show)

instance Arbitrary Key where
  arbitrary = do
    NonNegative n ← scale (`div`2) arbitrary
    return $ Key n
  shrink (Key k) = Key <$> shrink k
```

Here *scale* adjusts QuickCheck's internal size parameter in the generation of n, resulting in random values whose average is half that of QuickCheck's normal random non-negative integers. Testing *prop_Measure* using this type for keys results in the following, much better, distribution:

```
 55.3881% present
 44.6119% absent

 70.6567% middle
 11.6540% at end
 10.8601% at start
  5.1937% empty
  1.6355% just k
```

This example illustrates that "collisions" (that is, cases in which we randomly choose the same value in two places) can be important test cases. Indeed, consider the following (obviously false) property:

```
prop_Unique x y = x ≢ y
```

If we were to choose x and y uniformly from the entire range of 64-bit integers, then QuickCheck would never be able to falsify it, in practice. If we use QuickCheck's built-in *Int* generator, then the property fails in around 3.3% of cases. Using the *Key* generator we have just defined, the property fails in 9.3% of cases. The choice of generator should be made on the basis of how important collisions are as test cases.

5 Bug Hunting

To evaluate the properties we have written, we created eight buggy implementations of binary search trees, with bugs ranging from subtle to blatant. These implementations are listed in Figure 7.

Bug #	Description
1	*insert* discards the existing tree, returning a single-node tree just containing the newly inserted value.
2	*insert* fails to recognize and update an existing key, inserting a duplicate entry instead.
3	*insert* fails to update an existing key, leaving the tree unchanged instead.
4	*delete* fails to rebuild the tree above the key being deleted, returning only the remainder of the tree from that point on (an easy mistake for those used to imperative programming to make).
5	Key comparisons reversed in *delete*; only works correctly at the root of the tree.
6	*union* wrongly assumes that all the keys in the first argument precede those in the second.
7	*union* wrongly assumes that if the key at the root of t is smaller than the key at the root of t', then all the keys in t will be smaller than the key at the root of t'.
8	*union* works correctly, except that when both trees contain the same key, the left argument does not always take priority.

Fig. 7. The eight buggy implementations.

The results of testing each property for each buggy version are shown in Figure 8. We make the following observations.

5.1 Bug finding effectiveness

Validity properties miss many bugs (five of eight), as do "preservation of equivalence" and "completeness of insertion" properties. In contrast, every bug is found by at least one postcondition, metamorphic property, and model-based property.

Invalid test data provokes false positives. Bug #2, which causes invalid trees to be generated as test cases, causes many properties that *do not use insert* to fail. This is why *prop_ArbitraryValid* is so important—when it fails, we need not waste time debugging false positives in properties unrelated to the bug. Because of these false positives, we ignore bug #2 in the rest of this discussion.

Model-based properties are effective at finding bugs; each property tests just one operation, and finds every bug in that operation. In fact, the model-based properties together form a complete specification of the code, and so should be expected to find every bug.

Postconditions are quite effective; each postcondition for a buggy operation finds all the bugs we planted in it, but some postconditions are less effective than we might expect. For example, *prop_FindPostPresent* uses both *find* and *insert*, so we might expect it to reveal the three bugs in *insert*, but it reveals only two of them.

	insert bugs			*delete* bugs		*union* bugs		
Property	#1	#2	#3	#4	#5	#6	#7	#8
Validity properties								
prop_ArbitraryValid		✗						
prop_NilValid								
prop_InsertValid		✗						
prop_DeleteValid		✗						
prop_UnionValid		✗				✗	✗	
prop_ShrinkValid								
Postconditions								
prop_InsertPost	✗	✗	✗					
prop_DeletePost		✗		✗	✗			
prop_FindPostPresent		✗	✗					
prop_FindPostAbsent		✗			✗			
prop_InsertDeleteComplete		✗		✗				
prop_UnionPost						✗	✗	✗
Metamorphic properties								
prop_InsertInsertWeak	✗							
prop_InsertInsert	✗	✗	✗					
prop_InsertDeleteWeak				✗				
prop_InsertDelete		✗	✗	✗				
prop_InsertUnion	✗	✗	✗			✗	✗	✗
prop_DeleteNil								
prop_DeleteInsertWeak				✗				
prop_DeleteInsert	✗	✗		✗	✗			
prop_DeleteDelete				✗	✗			
prop_DeleteUnion		✗		✗	✗	✗	✗	✗
prop_UnionNil1								
prop_UnionNil2								
prop_UnionDeleteInsert	✗	✗	✗	✗	✗	✗	✗	✗
prop_UnionUnionIdem						✗		
prop_UnionUnionAssoc						✗	✗	✗
prop_FindNil								
prop_FindInsert	✗	✗	✗					
prop_FindDelete		✗		✗	✗			
prop_FindUnion						✗	✗	✗
Preservation of equivalence								
prop_InsertPreservesEquivWeak								
prop_InsertPreservesEquiv								
prop_DeletePreservesEquiv	✗			✗	✗			
prop_UnionPreservesEquiv	✗						✗	✗
prop_FindPreservesEquiv	✗							
Completeness of insertion								
prop_InsertComplete								
prop_InsertCompleteForDelete								
prop_InsertCompleteForUnion	✗					✗	✗	
Model-based properties								
prop_NilModel								
prop_InsertModel	✗	✗	✗					
prop_DeleteModel		✗		✗	✗			
prop_UnionModel		✗				✗	✗	✗
prop_FindModel								
Total failures	12	17	8	12	9	10	10	8

Fig. 8. Failing properties for each bug.

Metamorphic properties are less effective individually, but powerful in combination. Weak properties miss bugs (compare each line ending in *Weak* with the line below), because their preconditions to exclude tricky test cases result in tricky bugs escaping detection. But even stronger-looking properties that we might expect to find bugs miss them—*prop_InsertDelete* misses bug #1 in *insert*, *prop_DeleteInsert* misses bug #3 in *insert*, and so on. Degenerate metamorphic properties involving *nil* are particularly ineffective. Metamorphic properties are essentially an axiomatization of the API under test, and there is no guarantee that this axiomitization is complete, so some bugs might be missed altogether.

5.2 Bug finding performance

Property type	**Min**	**Max**	**Mean**
Postcondition	7.1	245	77
Metamorphic	2.4	714	56
Model-based	3.1	9.8	5.8

Fig. 9. Average mean number of tests required to make a property of each type fail.

Hitherto we have discussed which properties *can* find bugs, given enough testing time. But it also matters *how quickly* a property can find a bug. For seven of our eight bugs (omitting bug #2, which causes invalid test cases to be

generated), and for each postcondition, metamorphic property, and model-based property that detects the bug, we found a counterexample to the property using QuickCheck 1,000 times with different random seeds, and recorded the *mean number of tests* needed to make that property fail for that bug. Note that finding a counterexample 1,000 times requires running far more than 1,000 random tests: we ran over 700,000 tests of the hardest-to-falsify property in total, in order to find a counterexample 1,000 times. We then averaged the mean-time-to-failure across all bugs, and all properties of the same type. The results are summarized in Figure 9.

In this example model-based properties find bugs far faster than postconditions or metamorphic properties, while metamorphic properties find bugs a little faster than postconditions on average, but their mean time to failure varies more.

Digging a little deeper, for the same bug in *union*, *prop_UnionPost* fails after 50 tests on average, while *prop_UnionModel* fails after only 8.4 tests, *even though they are logically equivalent.* The reason is that after computing a *union* that is affected by the bug, the model-based property checks that the *model* of the result is correct—which requires *every* key and value to be correct. The post-condition, on the other hand, checks that a *random* key has the correct value in the result. Thus *prop_UnionPost* may exercise the bug many times without detecting it. Each model-based test may take a little longer to run, because it validates the result of *union* more thoroughly, but this is not significant compared to the enormous difference in the *number* of tests required to find the bug—the entire test case must be generated, and the *union* computed, in either case, so the difference in validation time is not really important.

5.3 Lessons

These results suggest that, if time is limited, then writing model-based properties may offer the best return on investment, in combination with validity properties to ensure we don't encounter confusing failures caused by invalid data. In situations where the model is complex (and thus expensive) to define, or where the model resembles the implementation so closely that the same bugs are likely in each, then metamorphic properties offer an effective alternative, at the cost of writing many more properties.

6 Related work

Pre- and post-conditions were introduced by Hoare [15] for the purpose of proving programs correct, inspired by Floyd [11]. The notion of a data representation invariant, which we use here for "validity testing", comes from Hoare's 1972 paper on proving data representations correct [14]. Pre- and post-conditions and invariants also form an integral part of Meyer's "Design by Contract" approach to designing software [19], in which an invariant is specified for each class, and

pre- and post-conditions for each class method, and these can optionally be checked at run-time—for example during testing.

Metamorphic testing was introduced by Chen, Cheung and Yiu as a way of deriving tests that do not require an oracle [6]. They consider, for example, an algorithm to find shortest-paths in a graph. While it is difficult to check whether a path found by the algorithm is actually shortest, it is easy to compare the path found from a node with the paths found from its neighbours, and check that it is no longer than the shortest path via a neighbour. As in this case, the key idea is to compare results from multiple invocations of the code-under-test, and check that an appropriate "metamorphic relation" holds between them. We have used equalities and equivalences as metamorphic relations in this paper, but the idea is much more general—for example, one might test that *insert* does not *reduce the size* of a tree, which would catch bugs that accidentally discard part of the structure. Metamorphic testing is useful in many contexts, and is now the subject of an annual workshop series[8].

Metamorphic properties which are equations or equivalences are a form of *algebraic specification* [13]. Guttag and Horning divide the operations into those that return the type of interest (*nil*, *insert*, *delete*, and *union*, in our case), and *observations* that return a different type (*find*). They give conditions for "sufficient completeness", meaning that the specification precisely determines the value of any observation.

We already saw that the idea behind model-based properties comes from Hoare's seminal paper [14]. Using an abstract model as a specification is also at the heart of the Z specification language [26], and the field of model-based testing [5], an active research area with two workshop series devoted to it[9][10].

The title of the paper is of course inspired by Polya's classic book [23].

7 Discussion

We have discussed a number of different kinds of properties that a developer can try to formulate to test an implementation: invariant properties, postconditions, metamorphic properties, inductive properties, and model-based properties. Each kind of property is based on a widely applicable idea, usable in many different settings. When writing metamorphic properties, we discovered the need to define *equivalence* of data structures, and thus also to define properties that test for preservation of equivalence. We discussed the importance of *completeness*—our test data generator should be able to generate any test case—and saw how to test this. We saw the importance of testing both our generators and our shrinkers, to ensure that other properties are tested with valid data. We saw how to *measure* the distribution of test data, to ensure that test effort is well spent.

Model-based testing seemed the most effective approach overall, revealing all our bugs with a small number of properties, and generally finding bugs fast. But

[8] `http://metwiki.net/MET19/`
[9] `http://mbt-workshop.org/`
[10] `https://conf.researchr.org/series/a-most`

metamorphic testing was a fertile source of ideas, and was almost as effective at revealing bugs, so is a useful alternative, especially in situations where a model is expensive to construct.

We saw that some properties must use equivalence to compare values, while other properties must use structural equality. Thus, we need *two* notions of "equality" for the data structures under test. In fact, it is the *equivalence* which ought to be exported as the equality instance for binary search trees, because structural equality distinguishes representations that ought to be considered equal outside the abstraction barrier of the abstract data type. Yet we need to use structural equality in some properties, and of course, we want to use the derived *Eq* instance for the representation datatype for this. So we appear to need *two* *Eq* instances for the same type! The solution to this conundrum is to define *two* types: a data type of representations with a derived structural equality, which is *not* exported to clients, and a **newtype** isomorphic to this datatype, which is exported, with an *Eq* instance which defines equality to be equivalence. This approach does mean that some properties must be *inside* the abstraction barrier of the data type, and thus must be placed in the same module as the implementation, which may not be desirable as it mixes test code and implementation code. An alternative is to define an *Internals* module which exports the representation type, and can be imported by test code, but is not used by client modules.

The choice of properties (and generators) may also depend on whether the tester takes a "white box" or "black box" view of the code. From the perspective of an *implementor*, it makes sense to use properties such as validity properties, that depend on the representation of the data. From the perspective of a *user*, properties should use only the API exported by the implementor—as do metamorphic and model-based properties. In this paper we generated random trees using the exported API, but of course we could also have generated the representation directly. This is certainly possible, but more complicated and error-prone, and often no more effective.

The ideas in this paper are applicable to testing any *pure* code, but code with side-effects demands a somewhat different approach. In this case, every operation has an implicit "state" argument, and an invisible state result, making properties harder to formulate. Test cases are *sequences* of operations, to set up the state for each operation under test, and to observe changes made to the state afterwards. Nevertheless, the same ideas can be adapted to this setting; in particular, there are a number of state-machine modelling libraries for property-based testing tools that support a "model-based" approach in a stateful setting. State machine modelling is heavily used at Quviq AB[11] for testing customer software, and an account of some of these examples can be found in [17].

We hope the reader will find the ideas in this paper helpful in developing effective property-based tests in the future.

[11] A company founded in 2006 by the author and Thomas Arts, to commercialize property based testing. See `http://quviq.com`.

Acknowledgements

I'm grateful to the anonymous referees for many useful suggested improvements, and to Vetenskapsrådet for funding this work under the SyTeC grant.

References

1. Cláudio Amaral, Mário Florido, and Vítor Santos Costa. Prologcheck – property-based testing in prolog. In Michael Codish and Eijiro Sumii, editors, *Functional and Logic Programming*, pages 1–17, Cham, 2014. Springer International Publishing.
2. 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, 2006.
3. E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo. The oracle problem in software testing: A survey. *IEEE Trans. on Soft. Eng.*, 41(5):507–525, May 2015.
4. Rudy Matela Braquehais. *Tools for discovery, refinement and generalization of functional properties by enumerative testing.* PhD thesis, University of York, UK, 2017.
5. Manfred Broy, Bengt Jonsson, J-P Katoen, Martin Leucker, and Alexander Pretschner. Model-based testing of reactive systems. In *Volume 3472 of Springer LNCS*. Springer, 2005.
6. Tsong Y Chen, Shing C Cheung, and Shiu Ming Yiu. Metamorphic testing: a new approach for generating next test cases. Technical report, Technical Report HKUST-CS98-01, Department of Computer Science, Hong Kong, 1998.
7. Tsong Yueh Chen, Fei-Ching Kuo, Huai Liu, Pak-Lok Poon, Dave Towey, T. H. Tse, and Zhi Quan Zhou. Metamorphic testing: A review of challenges and opportunities. *ACM Comput. Surv.*, 51(1):4:1–4:27, January 2018.
8. Koen Claessen. Inductive testing. Private communication; see slides at `https://docs.google.com/presentation/d/1pejW9foV4ZAw5e03kYR3urNQsIPobomY_5HshxZQpLc/edit?usp=drivesdk`
9. Koen Claessen and John Hughes. Quickcheck: A lightweight tool for random testing of haskell programs. In *Proc. 5th ACM SIGPLAN Int. Conf. on Functional Programming*, ICFP '00, 2000.
10. Lindley *et al.*, editor. *A List of Successes That Can Change the World - Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*, volume 9600 of *Lecture Notes in Computer Science*. Springer, 2016.
11. Robert W Floyd. Assigning meanings to programs. In *Program Verification*, pages 65–81. Springer, 1993.
12. Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: directed automated random testing. In *ACM Sigplan Notices*, volume 40, pages 213–223. ACM, 2005.
13. John V. Guttag and James J. Horning. The algebraic specification of abstract data types. *Acta informatica*, 10(1):27–52, 1978.
14. C. A. Hoare. Proof of correctness of data representations. *Acta Inf.*, 1(4):271–281, December 1972.
15. Charles Antony Richard Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.
16. John Hughes. Experiences with quickcheck: Testing the hard stuff and staying sane. In *et al.* [10], pages 169–186.

17. John Hughes. Experiences with quickcheck: Testing the hard stuff and staying sane. In *et al.* [10], pages 169–186.
18. Andreas Löscher and Konstantinos Sagonas. Targeted property-based testing. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 46–56. ACM, 2017.
19. Bertrand Meyer. Applying'design by contract'. *Computer*, 25(10):40–51, 1992.
20. Rickard Nilsson. Scalacheck: the definitive guide. 2014.
21. Manolis Papadakis and Konstantinos Sagonas. A proper integration of types and function specifications with property-based testing. In *Proceedings of the 10th ACM SIGPLAN workshop on Erlang*, pages 39–50. ACM, 2011.
22. Lee Pike. Smartcheck: automatic and efficient counterexample reduction and generalization. In Wouter Swierstra, editor, *Proceedings of the 2014 ACM SIGPLAN symposium on Haskell, Gothenburg, Sweden, September 4-5, 2014*, pages 53–64. ACM, 2014.
23. G Polya. *How to solve it! A system of thinking which can help you solve any problem.* Princeton University Press, 1945.
24. Colin Runciman, Matthew Naylor, and Fredrik Lindblad. Smallcheck and lazy smallcheck: automatic exhaustive testing for small values. In Andy Gill, editor, *Proceedings of the 1st ACM SIGPLAN Symposium on Haskell, Haskell 2008, Victoria, BC, Canada, 25 September 2008*, pages 37–48. ACM, 2008.
25. Koushik Sen, Darko Marinov, and Gul Agha. Cute: a concolic unit testing engine for c. In *ACM SIGSOFT Software Engineering Notes*, volume 30, pages 263–272. ACM, 2005.
26. J Michael Spivey. *Understanding Z: a specification language and its formal semantics*, volume 3. Cambridge University Press, 1988.

A Metamorphic properties

$$prop\_InsertInsertWeak\ (k, v)\ (k', v')\ t = k \not\equiv k' \Longrightarrow insert\ k\ v\ (insert\ k'\ v'\ t) \mathrel{\hat{=}} insert\ k'\ v'\ (insert\ k\ v\ t)$$

$$prop\_InsertInsert\ (k, v)\ (k', v')\ t = insert\ k\ v\ (insert\ k'\ v'\ t) \mathrel{\hat{=}} \textbf{if}\ k \equiv k'\ \textbf{then}\ insert\ k\ v\ t\ \textbf{else}\ insert\ k'\ v'\ (insert\ k\ v\ t)$$

$$prop\_InsertDeleteWeak\ (k, v)\ k'\ t = k \not\equiv k' \Longrightarrow insert\ k\ v\ (delete\ k'\ t) \mathrel{\hat{=}} delete\ k'\ (insert\ k\ v\ t)$$

$$prop\_InsertDelete\ (k, v)\ k'\ t = insert\ k\ v\ (delete\ k'\ t) \mathrel{\hat{=}} \textbf{if}\ k \equiv k'\ \textbf{then}\ insert\ k\ v\ t\ \textbf{else}\ delete\ k'\ (insert\ k\ v\ t)$$

$$prop\_InsertUnion\ (k, v)\ t\ t' = insert\ k\ v\ (union\ t\ t') \mathrel{\hat{=}} union\ (insert\ k\ v\ t)\ t'$$

$$prop\_DeleteInsertWeak\ k\ (k', v')\ t = k \not\equiv k' \Longrightarrow delete\ k\ (insert\ k'\ v'\ t) \mathrel{\hat{=}} insert\ k'\ v'\ (delete\ k\ t)$$

$$prop\_DeleteNil\ k = delete\ k\ nil == (nil :: Tree)$$

$$prop\_DeleteInsert\ k\ (k', v')\ t = delete\ k\ (insert\ k'\ v'\ t) \mathrel{\hat{=}} \textbf{if}\ k \equiv k'\ \textbf{then}\ delete\ k\ t\ \textbf{else}\ insert\ k'\ v'\ (delete\ k\ t)$$

$$prop\_DeleteDelete\ k\ k'\ t = delete\ k\ (delete\ k'\ t) \mathrel{\hat{=}} delete\ k'\ (delete\ k\ t)$$

$$
\begin{aligned}
&\mathit{prop\_DeleteUnion}\ k\ t\ t' = \\
&\quad \mathit{delete}\ k\ (\mathit{union}\ t\ t') \doteq \mathit{union}\ (\mathit{delete}\ k\ t)\ (\mathit{delete}\ k\ t') \\
&\mathit{prop\_UnionNil1}\ t = \mathit{union}\ \mathit{nil}\ t == t \\
&\mathit{prop\_UnionNil2}\ t = \mathit{union}\ t\ \mathit{nil} == t \\
&\mathit{prop\_UnionDeleteInsert}\ t\ t'\ (k, v) = \\
&\quad \mathit{union}\ (\mathit{delete}\ k\ t)\ (\mathit{insert}\ k\ v\ t') \doteq \mathit{insert}\ k\ v\ (\mathit{union}\ t\ t') \\
&\mathit{prop\_UnionUnionIdem}\ t = \mathit{union}\ t\ t \doteq t \\
&\mathit{prop\_UnionUnionAssoc}\ \mathit{t1}\ \mathit{t2}\ \mathit{t3} = \\
&\quad \mathit{union}\ (\mathit{union}\ \mathit{t1}\ \mathit{t2})\ \mathit{t3} == \mathit{union}\ \mathit{t1}\ (\mathit{union}\ \mathit{t2}\ \mathit{t3}) \\
&\mathit{prop\_FindNil}\ k = \mathit{find}\ k\ (\mathit{nil} :: \mathit{Tree}) == \mathit{Nothing} \\
&\mathit{prop\_FindInsert}\ k\ (k', v')\ t = \\
&\quad \mathit{find}\ k\ (\mathit{insert}\ k'\ v'\ t) == \textbf{if}\ k \equiv k'\ \textbf{then}\ \mathit{Just}\ v'\ \textbf{else}\ \mathit{find}\ k\ t \\
&\mathit{prop\_FindDelete}\ k\ k'\ t = \\
&\quad \mathit{find}\ k\ (\mathit{delete}\ k'\ t) == \textbf{if}\ k \equiv k'\ \textbf{then}\ \mathit{Nothing}\ \textbf{else}\ \mathit{find}\ k\ t \\
&\mathit{prop\_FindUnion}\ k\ t\ t' = \mathit{find}\ k\ (\mathit{union}\ t\ t') == (\mathit{find}\ k\ t <|> \mathit{find}\ k\ t')
\end{aligned}
$$