Automatic Testing of Operation Invariance

Tobias Gödderz and Janis Voigtländer

University of Bonn, Germany, {goedderz,jv}@cs.uni-bonn.de

Abstract. We present an approach to automatically generating operation invariance tests for use with Haskell's random testing framework QuickCheck. The motivation stems from a paper by Holdermans [8] which showed how to address certain shortcomings of straightforward testing of implementations of an abstract datatype. While effective, his solution requires extra generation work from the test engineer. Also, it may not even be doable if the person responsible for testing has no knowledge about, and program-level access to, the internals of the concrete datatype implementation under test. We propose and realize a refinement to Holdermans' solution that improves on both aspects: Required operation invariance tests can be formulated even in ignorance of implementation internals, and can be automatically generated using Template Haskell.

1 Introduction

It is good software engineering practice to test one's, and possibly other's, code. In declarative languages, QuickCheck [4] and related tools [1, 3, 10] are an attractive option, combining convenient ways of generating test input data and a DSL for expressing properties to be tested.

One recurring situation where property-based testing is desirable is in connection with implementations of an abstract datatype (ADT) specification. The scenario is that some interface (API) is given as a collection of type signatures of operations along with a collection of equational axioms that are expected to hold of those operations. Then there is an, or possibly several, implementations of that specification. The user of such an implementation should not need to be concerned with its internals, but would still like to be convinced of its correctness. It seems very natural to simply turn the given axioms into QuickCheck-like properties, and to accept an implementation as correct if it passes all those axioms-become-tests. However, it has been known for a while that such an approach is not enough to uncover all possible errors [6, 9]. Subtle bugs can remain hidden, due to an unfortunate interplay of buggy implementation and programmed equality (while actual semantic equality is impossible to test in general). Recently, Holdermans [8] presented a solution that works by adding operation invariance tests, to ensure that the assumed notion of equality is not just an equivalence relation, but actually a congruence relation. His solution has its own problems though. Specifically, it requires hand-writing a certain kind of additional test input data generator. But not only is that something which may depend on implementation internals (so is not necessarily something that a user having access only to the datatype's external API could do); it is also extra work and an additional source of potential errors: get that generator wrong by not covering enough ground, and even the additional tests will not be enough to really establish overall correctness. We set out to improve on these aspects.

2 The Need for Operation Invariance Tests

This section is largely a recap of material from Holdermans' paper. Specifically, we also use his very compelling example of how the approach of simply (and only) testing the axioms from the specification of an ADT may lead to a false sense of security.

Assume a set of people is interested in integer FIFO queues, and in particular in separately specifying, implementing, and using them. The specification is just a mathematical entity, consisting of a listing of signatures of desirable operations:

$$
\begin{aligned}
&\mathit{empty} &&:: \mathsf{Queue}\\
&\mathit{enqueue} &&:: \mathsf{Int} \to \mathsf{Queue} \to \mathsf{Queue}\\
&\mathit{isEmpty} &&:: \mathsf{Queue} \to \mathsf{Bool}\\
&\mathit{dequeue} &&:: \mathsf{Queue} \to \mathsf{Queue}\\
&\mathit{front} &&:: \mathsf{Queue} \to \mathsf{Int}
\end{aligned}
$$

and of axioms expected to hold:

$$
\begin{aligned}
&\mathsf{Q}_1\colon \mathit{isEmpty}\ \mathit{empty} = \mathsf{True} &&\\
&\mathsf{Q}_2\colon \mathit{isEmpty}\ (\mathit{enqueue}\ x\ q) = \mathsf{False} &&\\
&\mathsf{Q}_3\colon \mathit{front}\ (\mathit{enqueue}\ x\ \mathit{empty}) = x &&\\
&\mathsf{Q}_4\colon \mathit{front}\ (\mathit{enqueue}\ x\ q) = \mathit{front}\ q && \textbf{if}\ \mathit{isEmpty}\ q = \mathsf{False}\\
&\mathsf{Q}_5\colon \mathit{dequeue}\ (\mathit{enqueue}\ x\ \mathit{empty}) = \mathit{empty} &&\\
&\mathsf{Q}_6\colon \mathit{dequeue}\ (\mathit{enqueue}\ x\ q) = \mathit{enqueue}\ x\ (\mathit{dequeue}\ q) && \textbf{if}\ \mathit{isEmpty}\ q = \mathsf{False}
\end{aligned}
$$

An implementation in Haskell would be a module

```
module Queue (Queue, empty, enqueue, isEmpty, front, dequeue) where
...
```

that contains some definition for Queue as well as definitions for the operations adhering to the type signatures from the specification. Importantly, the type Queue is exported without any data constructors, so from the outside of the module, Queue values can only be created, manipulated, and inspected using the provided operations. A prospective user of the implementation can import the above module and call those operations in application code. Now, the user would like to have some certainty that the implementation is correct. The appropriate notion of correctness is adherence to the axioms from the mathematical specification, on which user and implementer should have agreed beforehand. One way for the implementer to convince the user of the correctness is to do extensive property-based testing to establish validity of the specification's axioms for the provided implementation. Using QuickCheck, the test suite would consist of the following properties:

$$
\begin{aligned}
&q_1 = \mathit{property}\ (\mathit{isEmpty}\ \mathit{empty} == \mathsf{True})\\
&q_2 = \mathit{property}\ (\lambda x\ q \to \mathit{isEmpty}\ (\mathit{enqueue}\ x\ q) == \mathsf{False})\\
&q_3 = \mathit{property}\ (\lambda x \to \mathit{front}\ (\mathit{enqueue}\ x\ \mathit{empty}) == x)
\end{aligned}
$$

$$
\begin{aligned}
q_4 &= \mathit{property}\ (\lambda x\ q \to \mathit{isEmpty}\ q == \mathsf{False} \\
&\qquad\qquad \Longrightarrow \\
&\qquad\qquad \mathit{front}\ (\mathit{enqueue}\ x\ q) == \mathit{front}\ q) \\
q_5 &= \mathit{property}\ (\lambda x \to \mathit{dequeue}\ (\mathit{enqueue}\ x\ \mathit{empty}) == \mathit{empty}) \\
q_6 &= \mathit{property}\ (\lambda x\ q \to \mathit{isEmpty}\ q == \mathsf{False} \\
&\qquad\qquad \Longrightarrow \\
&\qquad\qquad \mathit{dequeue}\ (\mathit{enqueue}\ x\ q) == \mathit{enqueue}\ x\ (\mathit{dequeue}\ q))
\end{aligned}
$$

These could even be written down by the user person in ignorance of any internals of the implementation. However, the use of $==$ on Queue values requires availability of an appropriate instance of the Eq type class (which is how Haskell organizes overloading of $==$). Let us assume the implementer provides such an instance. Moreover, we have to assume that the implementer also provides an appropriate random generator for Queue values (the q values quantified via the lambda-abstractions above – whereas for the Int values quantified as x we can take for granted that a random generator already exists), because otherwise the properties cannot be tested. In general, the implementer may even have to provide several random Queue generators for different purposes, for example since otherwise certain preconditions in axioms might be too seldom fulfilled, thus preventing effective testing.[1] But in the example here this is not an issue, since both empty and nonempty queues will appear with reasonable likelihood among the generated test data.

So if a Queue implementation passes all the above tests, it should be correct, right? Unfortunately not. As Holdermans [8] demonstrates, it can happen, even under all the assumptions above, that an implementation passes all the tests but is still harmfully incorrect. Let us repeat that faulty implementation in full as well (also since we will later want to refer to it in checking the adequacy of our own testing solution to the overall problem). Here it is:

$$
\begin{aligned}
&\mathbf{data}\ \mathsf{Queue} = \mathsf{BQ}\ [\mathsf{Int}]\ [\mathsf{Int}] \\[4pt]
&\mathit{bq} :: [\mathsf{Int}] \to [\mathsf{Int}] \to \mathsf{Queue} \\
&\mathit{bq}\ [\,]\ r = \mathsf{BQ}\ (\mathit{reverse}\ r)\ [\,] \\
&\mathit{bq} f\ \ r = \mathsf{BQ} f\ r \\[4pt]
&\mathit{empty} :: \mathsf{Queue} \\
&\mathit{empty} = \mathit{bq}\ [\,]\ [\,] \\[4pt]
&\mathit{enqueue} :: \mathsf{Int} \to \mathsf{Queue} \to \mathsf{Queue} \\
&\mathit{enqueue}\ x\ (\mathsf{BQ} f\ r) = \mathit{bq} f\ (x : r) \\[4pt]
&\mathit{isEmpty} :: \mathsf{Queue} \to \mathsf{Bool} \\
&\mathit{isEmpty}\ (\mathsf{BQ} f\ \_) = \mathit{null} f
\end{aligned}
$$

[1] QuickCheck's restricting conditional operator $\Longrightarrow$, written as ==>, does not count "*A*" being false as evidence for "*A* implies *B*" being true. Thus, in order to reach a certain number of positive test cases for "*A* implies *B*", that many cases with both "*A*" and "*B*" being true must be encountered (and, of course, not a single case with "*A*" true but "*B*" false). Consequently, QuickCheck gives up testing if not enough cases with "*A*" being true are encountered.

```
front :: Queue → Int
front (BQ f _) = last f

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

This implementation uses the “smart constructor” *bq* to preserve the invariant that, for every BQ *f r*, it holds that *null f* implies *null r*, which makes *front* simpler to implement. Ironically, in the implementation above the error nevertheless lies in the definition of *front*, which should have been *head f* rather than *last f*.

As already mentioned, in order to test properties q_1–q_6, we need a definition of $==$ for Queue and a random generator for Queue values. The implementer is kind enough to provide both (and both are perfectly sane) within the Queue module:

```
instance Eq Queue where
  q == q′ = toList q == toList q′

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

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

Now we can run tests *quickCheck* q_1 through *quickCheck* q_6, and find that all are satisfied. And this is not just happenstance, by incidentally not hitting a counterexample during the random generation of test input data. No, it is a mathematical fact that the implementation above satisfies the $==$-equations in properties q_1–q_6. And yet the implementation is incorrect. To see this, consider:

```
> front (dequeue (enqueue 3 (enqueue 2 (enqueue 1 empty))))
3
```

Clearly, a correct implementation of a FIFO queue should have output 2 here. In fact, Q_2–Q_6 can be used to prove, by equational reasoning, that

$$\mathit{front}\ (\mathit{dequeue}\ (\mathit{enqueue}\ 3\ (\mathit{enqueue}\ 2\ (\mathit{enqueue}\ 1\ \mathit{empty})))) = 2$$

The key to what went wrong here (the implementation being incorrect and yet passing all the tests derived from the specification axioms) is that properties q_1–q_6 do not, and in fact cannot, test for semantic equivalence. They can only be formulated using the programmed equivalence $==$. That would be fine if $==$ were not just *any* equivalence relation that satisfies q_1–q_6 (which we know it does), but actually a *congruence relation* that does so. For otherwise, $==$ cannot correspond to the semantic equivalence $=$ intuitively used in Q_1–Q_6. Being a congruence relation means to, in addition to being an equivalence relation, be compatible with all operations from the ADT specification. And that is not the case for the implementation given above. For example,

```
> let q = dequeue (enqueue 3 (enqueue 2 (enqueue 1 empty)))
> let q′ = enqueue 3 (dequeue (enqueue 2 (enqueue 1 empty)))
> q == q'
True
```

but

```
> front q == front q'
False
```

That does not necessarily mean that, morally, the implementation of $==$ given for Queue (and used in the tests) is wrong. The real bug here resides in the implementation of *front*, but it cannot be detected by just checking q_1–q_6; one additionally needs appropriate compatibility axioms.[2]

So Holdermans [8] observes that one should check additional axioms like

$$q_7 = \mathit{property}\ (\lambda q\ q' \rightarrow q == q' \Longrightarrow \mathit{front}\ q == \mathit{front}\ q')$$

Actually, he wisely adds a non-emptiness check to the precondition to prevent both *front q* and *front q′* from leading to a runtime error. But even then, the new axiom (or any of the other added compatibility axioms) does not – except in very lucky attempts – actually uncover the bug in the implementation. The problem is that it is very unlikely to hit a case with $q == q'$ for random q and q'. And even if QuickCheck hits one such case every now and then, it is not very likely that it is *also* a counterexample to $\mathit{front}\ q == \mathit{front}\ q'$. So in the vast majority of test runs QuickCheck simply gives up, and we are none the wiser about whether the compatibility axioms do hold for the given implementation or do not.

Holdermans' solution to *this* problem is to invest manual effort into randomly generating pairs of q and q' that ought to be considered equivalent queues. Concretely,

```
data Equiv a = a :≡: a
instance Arbitrary (Equiv Queue) where
  arbitrary = do z ← arbitrary
                 x ← from z
                 y ← from z
                 return (x :≡: y)
    where
      from xs = do i ← choose (0, length xs − 1)
                   let (xs₁, xs₂) = splitAt i xs
                   return (bq xs₁ (reverse xs₂))
```

Given some Show instances for Queue and Equiv Queue, testing the newly formulated property

$$q_7 = \mathit{property}\ (\lambda (q :\equiv: q') \rightarrow \mathit{not}\ (\mathit{isEmpty}\ q) \Longrightarrow \mathit{front}\ q == \mathit{front}\ q')$$

would yield, for example:

[2] In general, one should *also* convince oneself in the first place that $==$ is indeed an equivalence relation as well (reflexive, symmetric, transitive), but often this will be a triviality. For example, any definition of the form $x == x' = f\ x == f\ x'$ for some function f, like in the Eq instance for Queue, is already guaranteed to give an equivalence relation if $==$ on the target type of f is known to be an equivalence relation.

```
> quickCheck q7
*** Failed! Falsifiable (after 5 tests):
BQ [2,0] [] :≡: BQ [2] [0]
```

That, finally, is useful information which can be used to figure out the bug in the implementation of *front*.

But this success depends on the implementer having provided a good definition for the Arbitrary (Equiv Queue) instance above, specifically, a good *from*-function for "perturbing" a randomly generated queue in different ways. Note, this is not something the user could do themselves, or exercise any control over, since implementing that instance crucially depends on having access to the internals of the queue implementation module. If the implementer makes a mistake in writing that function/instance, then the bug in the original implementation of *front* will possibly remain unnoticed. For example, in an extreme case, the implementer may accidentally write the Arbitrary (Equiv Queue) instance in such a way that *from* is semantically equivalent to *return*, in which case q_7 and all other compatibility axioms will be tested positively, despite the implementation of *front* still being incorrect. Holdermans notes that one might test the chosen perturbation functionality itself to ensure not having introduced an error there. But this, which would correspond to a test $quickCheck\ (\lambda(q :\equiv: q') \rightarrow q == q')$ does not help with judging the suitability of the perturbation in terms of having "enough randomness"; clearly $from\ xs = return\ xs$ would make this test succeed as well. And the inverse property, that any pair of equivalent queues $q == q'$ has indeed a chance of also being generated as $q :\equiv: q'$, is unfortunately *not* expressible as a QuickCheck test.

Our purpose now is to avoid the need of defining an Arbitrary (Equiv Queue) instance. Thus, we will empower the user to detect operation invariance violations in an ADT implementation without having access to internals or relying on the implementer to generate the required pairs of equivalent values for tests.

3 Thoughts on Random Terms

A simple idea would be to work purely symbolically as long as possible. For example, the values

$$q = dequeue\ (enqueue\ 3\ (enqueue\ 2\ (enqueue\ 1\ empty)))$$

and

$$q' = enqueue\ 3\ (dequeue\ (enqueue\ 2\ (enqueue\ 1\ empty)))$$

that we first mentioned as evidence for a case with $q == q'$ but $front\ q \mathrel{/=} front\ q'$[3], are semantically equivalent according to Q_1–Q_6. Naturally, *any* pair of conceptually equivalent queues can be shown (mathematically) to be equivalent by using those original axioms. After all, that is the whole point of having the ADT specification comprising of those axioms in the first place.

[3] We have $/=$ as the negation of programmed equivalence $==$.

So a suitable strategy for generating terms q and q' that would be eligible as $q :\equiv: q'$ pairs might be to start from the given signatures of operations, randomly generate a well-typed term q from those operations, then randomly apply a random selection of the axioms Q_1–Q_6 at random places in the term q to obtain another term q'. Using pairs of such q and q' for tests like $\mathit{front}\ q == \mathit{front}\ q'$ (now actually computing the result, no longer working symbolically) would indeed, in principle, have the power to detect the bugs that violate operation invariance. In fact, this is exactly the strategy we initially pursued.

However, it does not work out in practice. Already generating random terms is known to be non-trivial, though solutions exist [5]. Moreover applying random axioms at random places is difficult in general to get right in the sense of ending up with a probability distribution that is effective in terms of hitting cases that uncover bugs. For example, given an integer queue, only the root node can be of type Bool, which will lead to a low probability of axiom Q_2 to be applicable at a randomly selected node and an even lower probability of it to be selected for application, especially for large terms. Conversely first deciding on an axiom to apply and then randomly generating a (sub)term at which it is indeed applicable leads to similar uncertainties about coverage.

Additionally, there is a problem with "infeasible" terms. For integer queues, that is every term where *dequeue* or *front* is applied to a subterm t with $\mathit{isEmpty}\ t$ being True. Assume for now that x in each term $\mathit{enqueue}\ x\ q$ might only be an integer (and not a more complex subterm $\mathit{front}\ q'$).[4] Then the tree of each term is a path. Its leaf will be *empty*, its root node one of *enqueue*, *isEmpty*, *front*, or *dequeue*. All inner nodes can only be one of *enqueue* or *dequeue*. If at any node in the path more of its descendants are of type *dequeue* than *enqueue*, then the term is not valid. The probability that a term of length n is feasible is $\frac{1}{2}$ for $n = 1$, $\frac{252}{1024}$ for $n = 10$, and about 0.125 for $n = 40$. This might be much worse for ADTs with more complex terms.

Without prior knowledge about which operations are problematic and which axioms contribute to situations of equivalent values (queues, in the example) whose equivalence is not preserved by applying an operation, there is not much hope with the naive approach described above. But of course we precisely want to avoid having to invest any such knowledge about a specific ADT or implementation, since our goal is a generic solution to the problem. Also, working purely symbolically would not be practical for another reason: some axioms, like Q_4 and Q_6, have preconditions that need to be established before application. So in the hypothetical rewriting of q into q' by random applications of axioms described above, we would either have to symbolically prove such preconditions along the way (arbitrarily hard in general), or resort to actually computing values of subterms of q to establish whether some axiom is applicable there or not.

4 Our Solution

So what can we do instead? We want to avoid having to apply arbitrarily many rewrites at random places deep in some terms. Toward a solution, let us discuss the case of a binary operation f with two argument positions to be filled by a value of the abstract

[4] This could be seen as having an operation $\mathit{enqueue}_x :: \mathsf{Queue} \to \mathsf{Queue}$ for every $x :: \mathsf{Int}$.

datatype (unlike in the case of Queue, where no operation takes more than one queue as input).[5] The required operation invariance test would be that $t_1 :\equiv: t'_1$ and $t_2 :\equiv: t'_2$ imply $f\ t_1\ t_2 == f\ t'_1\ t'_2$, where $t_i :\equiv: t'_i$ means that t_i and t'_i are convertible, i.e., there is a sequence of axiom applications that leads from term t_i to term t'_i. We claim that it would be enough to focus on the two argument positions of f separately, and to consider only single axiom applications. Indeed, assume the above test leads to a counterexample, i.e., $f\ t_1\ t_2 \mathrel{/\!\!=} f\ t'_1\ t'_2$. Then there are already t, t', and t'' with $f\ t\ t' \mathrel{/\!\!=} f\ t\ t''$ or $f\ t'\ t \mathrel{/\!\!=} f\ t''\ t$ and where t' and t'' are exactly one axiom application apart. Why? By the given assumptions, we can form a sequence $f\ t_1\ t_2 = \ldots = f\ t'_1\ t_2 = \ldots = f\ t'_1\ t'_2$ of single axiom applications, where in the first part of that sequence axioms are only applied inside the first argument position of f, later only inside the second one. Now if t, t', and t'' with the claimed properties would not exist, then it would have to be the case that $f\ t_1\ t_2 == \ldots == f\ t'_1\ t_2 == \ldots == f\ t'_1\ t'_2$ and thus $f\ t_1\ t_2 == f\ t'_1\ t'_2$ by transitivity of $==$[6]. But this is a contradiction to $f\ t_1\ t_2 \mathrel{/\!\!=} f\ t'_1\ t'_2$.

So, generalizing from the case of a binary operation, we have that in order to establish operation invariance overall, it suffices to test that for every operation f it holds $f \ldots t' \ldots == f \ldots t'' \ldots$ for every argument position and all pairs t' and t'' of terms related by exactly one axiom application and where the other argument positions are appropriately filled with random input (but with the same on left and right; this is what the "..." indicate in $f \ldots t' \ldots == f \ldots t'' \ldots$). Still, the axiom application relating t' and t'' could be somewhere deeply nested, i.e., t' could be $f_1 \ldots (f_2 \ldots (\ldots (f_n \ldots\ lhs \ldots) \ldots) \ldots) \ldots$ and t'' be $f_1 \ldots (f_2 \ldots (\ldots (f_n \ldots\ rhs \ldots) \ldots) \ldots) \ldots$, where *lhs* and *rhs* are the two sides of an instantiated axiom, while all the other parts ("...") in the two nestings agree. We could generate tests arising from this, an important observation being that for the "..." parts, since they are equal on both sides (not only equivalent), we can simply use random values – no random symbolic *terms* are necessary, which is an important gain. Note that, as Haskell is purely functional, it is not necessary to model dependencies between arguments to obtain completeness. Actually, we restrict to a subset of all tests, namely ones where the axiom application in fact is at the root of t' and t''. That is, we only employ tests $f \ldots\ lhs \ldots == f \ldots\ rhs \ldots$, not e.g. $f \ldots (f_1 \ldots\ lhs \ldots) \ldots == f \ldots (f_1 \ldots\ rhs \ldots) \ldots$ – a pragmatic decision, but also one that has turned out well so far. We have not encountered a situation where this narrowing of test cases has missed some bug, except in very artificial examples manufactured just for that purpose. The intuitive reason seems to be that by doing all tests $f \ldots\ lhs \ldots == f \ldots\ rhs \ldots$, which of course also includes the tests $f_1 \ldots\ lhs \ldots == f_1 \ldots\ rhs \ldots$, for varying *lhs*/*rhs* as obtained from all axioms, one casts a fine enough net – situations where these all go through, along with all standard instantiations $lhs == rhs$ obtained from the axioms, but where $f \ldots (f_1 \ldots\ lhs \ldots) \ldots == f \ldots (f_1 \ldots\ rhs \ldots) \ldots$ would fail, appear to be extremely rare.

To summarize, we propose to test operation invariance via properties of the following form, for each combination of one operation, one argument position, and one axiom:

$$f\ x_1 \ldots (axiom_{lhs}\ y_1 \ldots y_m) \ldots x_n == f\ x_1 \ldots (axiom_{rhs}\ y_1 \ldots y_m) \ldots x_n$$

[5] The consideration of binary operations here is without loss of generality. Dealing with unary operations is simpler, and dealing with operations of higher arity than two is analogous to dealing with binary operations, just requires more index fiddling in the discussion.

[6] Note that while the operation invariance property of $==$ is still under test, we were willing to simply take for granted that $==$ is at least an equivalence relation. See also Footnote 2

where like the x_i (alluded to as the "other" values above), the y_j – which are meant to fill any variable positions in a symbolic axiom $lhs = rhs$ – can be simply values. No need for symbolic terms, since neither in the x_i nor in the y_j positions we need to assume any further possible axiom rewrites. Also, no need to generate terms/values in an implementation-dependent way, since the signature and axiom information from the ADT specification suffices. The discussion in the paragraphs preceding this one implies that the proposed approach is sound (if a counterexample is found, then there is a genuine problem with the ADT implementation under test), but not in general complete (even if the axioms and our subset of operation invariance properties are tested exhaustively, some bug in the ADT implementation might be missed – due to our pragmatic decision not to consider deeper nested rewrites like $f\ x_1 \ldots (f_1\ y_1 \ldots (axiom_{lhs}\ z_1 \ldots z_l) \ldots y_m) \ldots x_n ==$ $f\ x_1 \ldots (f_1\ y_1 \ldots (axiom_{rhs}\ z_1 \ldots z_l) \ldots y_m) \ldots x_n$).

The full set of tests of the proposed form for integer queues follows. An index $enqueue_1$ or $enqueue_2$ indicates that the first or second argument position of $enqueue$ is filled by the particular axiom, respectively. Parameters with a prime (i.e. q' and x') fill the other arguments of the tested operation as opposed to being variables of the relevant axiom.

$$
\begin{array}{l}
enqueue_1\_q_3 = property\ (\lambda x\ q' \rightarrow \textbf{let}\ lhs = front\ (enqueue\ x\ empty) \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad rhs = x \\
\qquad\qquad\qquad\qquad\qquad\quad\ \ \textbf{in}\ enqueue\ lhs\ q' == enqueue\ rhs\ q') \\
enqueue_1\_q_4 = property\ (\lambda x\ q\ q' \rightarrow isEmpty\ q == \mathsf{False} \\
\qquad\qquad\qquad\qquad\qquad\qquad \Longrightarrow \\
\qquad\qquad\qquad\qquad\qquad\qquad \textbf{let}\ lhs = front\ (enqueue\ x\ q) \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad rhs = front\ q \\
\qquad\qquad\qquad\qquad\qquad\qquad \textbf{in}\ enqueue\ lhs\ q' == enqueue\ rhs\ q') \\
enqueue_2\_q_5 = property\ (\lambda x\ x' \rightarrow \textbf{let}\ lhs = dequeue\ (enqueue\ x\ empty) \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad rhs = empty \\
\qquad\qquad\qquad\qquad\qquad\quad\ \ \textbf{in}\ enqueue\ x'\ lhs == enqueue\ x'\ rhs) \\
isEmpty\_q_5 = property\ (\lambda x \rightarrow \textbf{let}\ lhs = dequeue\ (enqueue\ x\ empty) \\
\qquad\qquad\qquad\qquad\qquad\quad rhs = empty \\
\qquad\qquad\qquad\qquad\qquad \textbf{in}\ isEmpty\ lhs == isEmpty\ rhs) \\
enqueue_2\_q_6 = property\ (\lambda x\ q\ x' \rightarrow isEmpty\ q == \mathsf{False} \\
\qquad\qquad\qquad\qquad\qquad\qquad \Longrightarrow \\
\qquad\qquad\qquad\qquad\qquad\qquad \textbf{let}\ lhs = dequeue\ (enqueue\ x\ q) \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad rhs = enqueue\ x\ (dequeue\ q) \\
\qquad\qquad\qquad\qquad\qquad\qquad \textbf{in}\ enqueue\ x'\ lhs == enqueue\ x'\ rhs) \\
isEmpty\_q_6 = property\ (\lambda x\ q \rightarrow isEmpty\ q == \mathsf{False} \\
\qquad\qquad\qquad\qquad\qquad \Longrightarrow \\
\qquad\qquad\qquad\qquad\qquad \textbf{let}\ lhs = dequeue\ (enqueue\ x\ q) \\
\qquad\qquad\qquad\qquad\qquad\quad rhs = enqueue\ x\ (dequeue\ q) \\
\qquad\qquad\qquad\qquad\qquad \textbf{in}\ isEmpty\ lhs == isEmpty\ rhs) \\
dequeue\_q_6 = property\ (\lambda x\ q \rightarrow isEmpty\ q == \mathsf{False} \\
\qquad\qquad\qquad\qquad\qquad \Longrightarrow
\end{array}
$$

$$
\begin{aligned}
&\textbf{let}\ \mathit{lhs} = \mathit{dequeue}\ (\mathit{enqueue}\ x\ q)\\
&\quad\ \mathit{rhs} = \mathit{enqueue}\ x\ (\mathit{dequeue}\ q)\\
&\textbf{in}\ \mathit{not}\ (\mathit{isEmpty}\ \mathit{lhs}) \Longrightarrow\\
&\quad \mathit{dequeue}\ \mathit{lhs} == \mathit{dequeue}\ \mathit{rhs})
\end{aligned}
$$

$$
\begin{aligned}
\mathit{front\_q}_6 = \mathit{property}\ (\lambda x\ q \rightarrow\ & \mathit{isEmpty}\ q == \mathsf{False}\\
&\Longrightarrow\\
&\textbf{let}\ \mathit{lhs} = \mathit{dequeue}\ (\mathit{enqueue}\ x\ q)\\
&\quad\ \mathit{rhs} = \mathit{enqueue}\ x\ (\mathit{dequeue}\ q)\\
&\textbf{in}\ \mathit{not}\ (\mathit{isEmpty}\ \mathit{lhs}) \Longrightarrow\\
&\quad \mathit{front}\ \mathit{lhs} == \mathit{front}\ \mathit{rhs})
\end{aligned}
$$

There are two additional tests that syntactically belong to the others, but do not really add anything:

$$
\begin{aligned}
\mathit{dequeue\_q}_5 = \mathit{property}\ (\lambda x \rightarrow\ & \textbf{let}\ \mathit{lhs} = \mathit{dequeue}\ (\mathit{enqueue}\ x\ \mathit{empty})\\
&\quad\ \mathit{rhs} = \mathit{empty}\\
&\textbf{in}\ \mathit{not}\ (\mathit{isEmpty}\ \mathit{lhs}) \Longrightarrow\\
&\quad \mathit{dequeue}\ \mathit{lhs} == \mathit{dequeue}\ \mathit{rhs})\\
\mathit{front\_q}_5 = \mathit{property}\ (\lambda x \rightarrow\ & \textbf{let}\ \mathit{lhs} = \mathit{dequeue}\ (\mathit{enqueue}\ x\ \mathit{empty})\\
&\quad\ \mathit{rhs} = \mathit{empty}\\
&\textbf{in}\ \mathit{not}\ (\mathit{isEmpty}\ \mathit{lhs}) \Longrightarrow\\
&\quad \mathit{front}\ \mathit{lhs} == \mathit{front}\ \mathit{rhs})
\end{aligned}
$$

This is because both sides of Q_5 are empty queues, but neither *dequeue* nor *front* work on those.

5 A Practical Implementation

The tests shown in the previous section can (almost) strictly syntactically be derived from the specification. For every operation, every argument the operation has, and every axiom, one test can be obtained – by applying the operation at the selected argument to the axiom if the types allow it. In addition, constraints may have to be added per operation to prevent their application to invalid values, like the constraints $\mathit{not}\ (\mathit{isEmpty}\ \mathit{lhs})$ in $\mathit{dequeue\_q}_6$ and $\mathit{front\_q}_6$ (and in $\mathit{dequeue\_q}_5$ and $\mathit{front\_q}_5$).

A tool or library that generates these tests automatically needs type information about both operations and axioms. Details about the implementation of the datatype and operations, or the specific terms in the axioms, are not necessary. As relatively arbitrarily typed code must be generated, it seems to be at least very tricky to do this with plain Haskell and without a lot of manual help. Thus, to automate our solution as much as possible, we used Template Haskell [11]. As a result, none of the shown tests need to be hand-written by the user.

As a case study, we demonstrate here different ways how our tool/library (in a new module Test.OITestGenerator) can be used to generate appropriate QuickCheck tests. First, the axioms have to be specified. For this, OITestGenerator exports a datatype AxiomResult *a*, its constructor =!=, and a restricting conditional operator $\Rrightarrow$[7] that works

[7] which is written as ===>

akin to $\Longrightarrow$ as known from QuickCheck. Axioms with variables are written as functions with corresponding arguments, returning an $\mathsf{AxiomResult}\ a$ where a is the codomain of the axiom's left- and right-hand side.

$$
\begin{aligned}
&q_1 :: \mathsf{AxiomResult\ Bool}\\
&q_1 = \mathit{isEmpty\ empty} =!= \mathsf{True}\\
&q_2 :: \mathsf{Int} \to \mathsf{Queue} \to \mathsf{AxiomResult\ Bool}\\
&q_2 = \lambda x\ q \to \mathit{isEmpty}\ (\mathit{enqueue\ x\ q}) =!= \mathsf{False}\\
&q_3 :: \mathsf{Int} \to \mathsf{AxiomResult\ Int}\\
&q_3 = \lambda x \to \mathit{front}\ (\mathit{enqueue\ x\ empty}) =!= x\\
&q_4 :: \mathsf{Int} \to \mathsf{Queue} \to \mathsf{AxiomResult\ Int}\\
&q_4 = \lambda x\ q \to \mathit{not}\ (\mathit{isEmpty\ q}) \Rrightarrow \mathit{front}\ (\mathit{enqueue\ x\ q}) =!= \mathit{front\ q}\\
&q_5 :: \mathsf{Int} \to \mathsf{AxiomResult\ Queue}\\
&q_5 = \lambda x \to \mathit{dequeue}\ (\mathit{enqueue\ x\ empty}) =!= \mathit{empty}\\
&q_6 :: \mathsf{Int} \to \mathsf{Queue} \to \mathsf{AxiomResult\ Queue}\\
&q_6 = \lambda x\ q \to \mathit{not}\ (\mathit{isEmpty\ q}) \Rrightarrow \mathit{dequeue}\ (\mathit{enqueue\ x\ q}) =!= \mathit{enqueue\ x}\ (\mathit{dequeue\ q})
\end{aligned}
$$

This is already enough preparation to generate the basic tests, i.e., direct translations of the axioms, with the provided function *generate_basic_tests*. It gets one argument, a list of Axioms. Axiom is a container holding a) the name of an axiom, which (the axiom) must be a function returning an AxiomResult *a*, and b) possibly custom generators for the function's arguments. It has one constructor function *axiom*, which takes a Name – custom generators can be assigned to an Axiom via *withGens*, which is explained later. Then *generate_basic_tests* returns an expression of type [Property]. Property is the type of QuickCheck tests, as returned by the function *property* in the earlier given versions of q_1–q_6.

In using *generate_basic_tests* to generate the basic tests from the above functions returning AxiomResults, two of Template Haskell's syntactic constructs will be needed: The first are *splices*. A splice is written $(…), where … is an expression. A splice may occur instead of an expression. The splice will be evaluated at compile time and the syntax tree returned by it will be inserted in its place. The second construct used is '…, where … is a name of a function variable or data constructor. Then '… is of type Name and its value represents the name of the … that was quoted. Using these constructs, we can write:

$$
\begin{aligned}
&\mathit{adt\_basic\_tests} :: [\mathsf{Property}]\\
&\mathit{adt\_basic\_tests} = \$(\mathbf{let}\ \mathit{axs} = \mathit{map\ axiom}\ [\text{'}q_1, \text{'}q_2, \text{'}q_3, \text{'}q_4, \text{'}q_5, \text{'}q_6]\\
&\qquad\qquad\qquad\quad \mathbf{in}\ \mathit{generate\_basic\_tests\ axs})
\end{aligned}
$$

Now, *adt_basic_tests* can be executed via *mapM_ quickCheck adt_basic_tests*.

Before the, for our purposes more interesting, operation invariance tests can be generated, constraints for *dequeue* and *front* must be specified. Such a constraint function has the purpose of deciding whether a set of arguments is valid for the respective operation. Thus it takes the same arguments, but returns a Bool independently of the operation's return type.

$\mathit{may\_dequeue} :: \mathsf{Queue} \rightarrow \mathsf{Bool}$
$\mathit{may\_dequeue} = \mathit{not} \circ \mathit{isEmpty}$

$\mathit{may\_front} :: \mathsf{Queue} \rightarrow \mathsf{Bool}$
$\mathit{may\_front} = \mathit{not} \circ \mathit{isEmpty}$

Given these, the operation invariance tests can be generated by the provided function *generate_oi_tests*. For operations there exists a datatype Op similar to Axiom, with one constructor function *op*. The function *withConstraint* may be used to add a constraint to an operation.[8] It may also be used multiple times on the same operation, in which case the constraints are connected with a logical "and".

```
adt_oi_tests :: [Property]
adt_oi_tests = $(let ops = [op 'empty
                           ,op 'enqueue
                           ,op 'isEmpty
                           ,withConstraint (op 'dequeue) 'may_dequeue
                           ,withConstraint (op 'front) 'may_front]
                     axs = map axiom ['q1,'q2,'q3,'q4,'q5,'q6]
                 in generate_oi_tests axs ops)
```

Note that the repeated local definition of *axs* (in both *adt_basic_tests* and *adt_oi_tests*) is necessary due to Template Haskell's stage restrictions. It is not possible to refer to a top-level declaration in the same file, because it is still being compiled when the splices are executed. Note also that *empty* could be omitted here from the list of operations as it takes no arguments.

Running the tests automatically generated above is enough to detect the buggy implementation of *front*!

```
+++ OK, passed 100 tests (100% Queue.enqueue@1/Main.q3).
+++ OK, passed 100 tests (100% Queue.enqueue@1/Main.q4).
+++ OK, passed 100 tests (100% Queue.enqueue@2/Main.q5).
+++ OK, passed 100 tests (100% Queue.isEmpty@1/Main.q5).
*** Gave up! Passed only 0 tests.
*** Gave up! Passed only 0 tests.
+++ OK, passed 100 tests (100% Queue.enqueue@2/Main.q6).
+++ OK, passed 100 tests (100% Queue.isEmpty@1/Main.q6).
+++ OK, passed 100 tests (100% Queue.dequeue@1/Main.q6).
*** Failed! Falsifiable (after 5 tests):
3
BQ [4] [4,-3]
```

The test that fails here is $\mathit{front\_q}_6$ (which would have appeared in the output as `Queue.front@1/Main.q6`). Note that two tests were generated that are correctly typed but have no valid input (as already observed further above when writing down properties by hand); namely $\mathit{dequeue\_q}_5$ and $\mathit{front\_q}_5$ alias `Queue.dequeue@1/Main.q5` and

[8] As opposed to constraints for axioms, which are specified using $\Rrightarrow$ in the function whose name is passed to *axiom*.

`Queue.front@1/Main.q5`. They could be avoided by using other functions exported by OITestGenerator and explained below.

Generators can be passed to an operation via *withGens* in a list, with one generator name for each argument, as in *withGens* (*op* '*enqueue*) ['*arbitrary*, '*arbitrary*]. It is not allowed to omit generators when using *withGens*; instead *arbitrary* must be passed if no special generator should be used for some position. The function *withGens* can be intermingled with *withConstraint* and may also be used on Axioms.

Also, there is a convenience function *generate_axiom's_tests* which takes only one Axiom and a list of Ops. It is useful when certain combinations of axioms and operations should be excluded. It can also be used when only specific argument positions of an operation should be excluded for an axiom. The function $but :: \mathsf{Op} \to \mathsf{Arg} \to \mathsf{Op}$, when called as o ‘*but*‘ i, excludes the ith argument from o when generating tests. Arg has a single constructor function $arg :: \mathsf{Int} \to \mathsf{Arg}$. The function *but* may be called multiple times on the same operation to exclude multiple arguments. To supplement it, there also is $only :: \mathsf{Op} \to \mathsf{Arg} \to \mathsf{Op}$ to include only one argument. For illustration:

```
all_q5 :: [Property]
all_q5 = $(let ops = [op 'empty
                     ,op 'enqueue
                     ,op 'isEmpty
                     ,op 'dequeue `but` arg 1
                     ,op 'front `but` arg 1]
           in generate_axiom's_tests (axiom 'q5) ops)
```

Of course, in this case, *dequeue* and *front* could simply be omitted completely as they do only have one argument.

Another convenience function is *generate_single_test*, which again works similarly to *generate_oi_tests*, but takes only one Axiom and one Op instead of lists, and generates only a single test. It may be used when more control is needed.

```
enqueue1_q3 = $(generate_single_test (axiom 'q3) (op 'enqueue `only` 1))
enqueue1_q4 = $(generate_single_test (axiom 'q4) (op 'enqueue `only` 1))
enqueue2_q5 = $(generate_single_test (axiom 'q5) (op 'enqueue `only` 2))
isEmpty1_q5 = $(generate_single_test (axiom 'q5) (op 'isEmpty `only` 1))
enqueue2_q6 = $(generate_single_test (axiom 'q6) (op 'enqueue `only` 2))
isEmpty1_q6 = $(generate_single_test (axiom 'q6) (op 'isEmpty `only` 1))
dequeue1_q6 = $(generate_single_test (axiom 'q6) (op 'dequeue `only` 1))
front1_q6   = $(generate_single_test (axiom 'q6) (op 'front `only` 1))
```

No constraints are passed here because the superfluous tests $dequeue_1\_q_5$ and $front_1\_q_5$ were purposefully omitted, and because no axiom but Q_5 can result in an empty queue.

As writing such a list of tests is cumbersome, there is a function $show\_all\_tests :: \mathsf{Maybe}\ (\mathsf{String} \to \mathsf{Int} \to \mathsf{String} \to \mathsf{String}) \to [\mathsf{Name}] \to [\mathsf{Name}] \to \mathsf{ExpQ}$ which takes an optional formatting function, a list of axiom names, and a list of operation names, and produces a String-typed expression whose content is exactly the code above (plus $dequeue_1\_q_5$ and $front_1\_q_5$, which were removed manually from the output).

$$\begin{aligned} single\_test\_str = \$(\textbf{let}\ & ops = [\text{'}empty, \text{'}enqueue, \text{'}isEmpty, \text{'}dequeue, \text{'}front] \\ & axs = [\text{'}q_1, \text{'}q_2, \text{'}q_3, \text{'}q_4, \text{'}q_5, \text{'}q_6] \\ \textbf{in}\ & show\_all\_tests\ \mathsf{Nothing}\ axs\ ops) \end{aligned}$$

If constraints have to be added to the generated code, they must be added manually. On the other hand, all *'only' n* could be omitted in the case above, since there is no operation with two arguments of the same type.[9] Instead of Nothing above, a custom formatting function can be passed that generates the names of the properties.

The implementation is available as a Cabal package at `http://hackage.haskell.org/package/qc-oi-testgenerator`. Insights into the implementation, as well as more discussion of formal aspects of the overall approach, can be found in the first author's diploma thesis [7].

6 Conclusion and Discussion

We have presented and implemented an approach for automatically checking operation invariance for Haskell implementations of abstract datatypes. The user writes down axioms by closely following standard QuickCheck syntax, and both the ordinary QuickCheck tests as well as additional operation invariance tests are derived from that.

It might have been more desirable to obtain the necessary information about axioms directly from existing QuickCheck tests, freeing the user from possibly having to rewrite them. For this it would be necessary, in the implementation, to obtain the syntax tree of a given Haskell declaration and find the axiom by looking for a call to $==$. Then, type information for the left- and right-hand side would be needed. As of today, neither is possible with Template Haskell: The Info data structure returned by *reify* has a field of type Maybe Dec to hold the syntax tree of the right-hand side of a declaration. However, according to the documentation of Template Haskell's most current version 2.9.0.0, there is no implementation for this and the field always contains Nothing. Another way to obtain the syntax tree would be to write the axioms in expression quotations. In both cases, though, Template Haskell offers no way of obtaining the necessary type information, as those can only be retrieved for Names using *reify*, but not for arbitrary syntax trees. Also due to Template Haskell not offering a way of calling the type checker or other convenient ways to help generate correctly typed code yet, the implementation does not currently support polymorphic types. A workaround making it possible to test polymorphic abstract datatypes is to construct a (suitably chosen, cf. [2]) monomorphic instance and rename all participating functions. That way, *reify* returns the monomorphic types.

On the conceptual side, it would be attractive to gain more insight into how effective the kind of tests we generate are in finding bugs in general. This might be achieved by a formalization of our approach and/or collecting experimental evidence from more case studies. Here we discuss only a small variation on the running example, which illustrates an interesting aspect concerning the implementation of equality:

[9] The function *generate_single_test* throws a compile time error unless there is exactly one way to combine the given Axiom and Op.

The error in the shown version of *front* is hidden from the basic tests only because the implemented equality compares two values by how the implementer thinks they *should* behave. An observational equality like the following one, which even does not touch the internals, would not so be fooled.

$$\begin{array}{ll} q == q' & \mid \mathit{isEmpty}\ q \mathrel{/=} \mathit{isEmpty}\ q' = \mathsf{False} \\ & \mid \mathit{isEmpty}\ q \qquad\qquad\quad\; = \mathsf{True} \\ & \mid \mathit{otherwise} \qquad\qquad\quad = \mathit{front}\ q == \mathit{front}\ q' \wedge \mathit{dequeue}\ q == \mathit{dequeue}\ q' \end{array}$$

Still, the basic tests will not suffice in general. As a very artificial example, consider the queue implementation used so far, but now without the error in *front*, and replace the following two functions:

$$\mathit{bq}\ f\ r = \mathsf{BQ}\ (f \mathbin{+\!\!+} \mathit{reverse}\ r)\ [\,]$$

$$\begin{array}{ll} \mathit{enqueue}\ x\ q@(\mathsf{BQ}\ f\ r) & \mid \mathit{isEmpty}\ q = \mathit{bq}\ f\ (r \mathbin{+\!\!+} [x]) \\ & \mid \mathit{otherwise} = \mathsf{BQ}\ f\ (r \mathbin{+\!\!+} [x]) \end{array}$$

This error will never be found with the basic tests and using the above observational equality. So using operation invariance tests is still a good idea.

Acknowledgments

We thank the anonymous reviewers for their comments and suggestions.

References

[1] C. Amaral, M. Florido, and V.S. Costa. PrologCheck – Property-based testing in Prolog. In *FLOPS*, pages 1–17. Springer, 2014.

[2] J.-P. Bernardy, P. Jansson, and K. Claessen. Testing polymorphic properties. In *ESOP*, pages 125–144. Springer, 2010.

[3] J. Christiansen and S. Fischer. EasyCheck – Test data for free. In *FLOPS*, pages 322–336. Springer, 2008.

[4] K. Claessen and J. Hughes. QuickCheck: A lightweight tool for random testing of Haskell programs. In *ICFP*, pages 268–279. ACM, 2000.

[5] J. Duregård, P. Jansson, and M. Wang. Feat: Functional enumeration of algebraic types. In *Haskell Symposium*, pages 61–72. ACM, 2012.

[6] J. Gannon, P. McMullin, and R. Hamlet. Data abstraction, implementation, specification, and testing. *ACM Trans. Program. Lang. Syst.*, 3(3):211–223, 1981.

[7] T. Gödderz. Verification of abstract data types: Automatic testing of operation invariance. Diploma thesis, University of Bonn, Germany, 2014. `http://tobias.goedderz.info/dt-inf.pdf`.

[8] S. Holdermans. Random testing of purely functional abstract datatypes: Guidelines for dealing with operation invariance. In *PPDP*, pages 275–284. ACM, 2013.

[9] P. Koopman, P. Achten, and R. Plasmeijer. Model based testing with logical properties versus state machines. In *IFL*, pages 116–133. Springer, 2012.

[10] C. Runciman, M. Naylor, and F. Lindblad. SmallCheck and Lazy SmallCheck: Automatic exhaustive testing for small values. In *Haskell Symposium*, pages 37–48. ACM, 2008.

[11] T. Sheard and S. Peyton Jones. Template meta-programming for Haskell. In *Haskell Workshop*, pages 1–16. ACM, 2002.