Testing Monadic Code with QuickCheck

Koen Claessen and John Hughes
Department of Computer Science
Chalmers University of Technology
{koen,rjmh}@cs.chalmers.se

ABSTRACT

QuickCheck is a previously published random testing tool for Haskell programs. In this paper we show how to use it for testing monadic code, and in particular imperative code written using the ST monad. QuickCheck tests a program against a specification: we show that QuickCheck's specification language is sufficiently powerful to represent common forms of specifications: algebraic, model-based (both functional and relational), and pre-/post-conditional. Moreover, all these forms of specification can be used directly for testing. We define a new language of monadic properties, and make a link between program testing and the notion of observational equivalence.

1. INTRODUCTION

QuickCheck [4] is an automated testing tool for Haskell programs. It defines a *formal specification language* which programmers can use to specify the code under test, and then checks that the stated properties hold in a large number of randomly generated test cases. Specifications are thus used directly, both for test case generation and as a test oracle. The benefits of using QuickCheck are *documented* and *repeatable* testing, and production of a specification which has been machine-checked for consistency with the program. While there is no guarantee that the specified properties hold in general, we and others have found this approach to be highly effective at revealing errors quickly [21, 17].

In our first QuickCheck paper, we focussed on testing pure functions, which are easier to test than side-effecting ones since one need not take a potentially large and complex state into account. Haskell programs consist to a large extent of pure functions, and so this focus was not a major restriction. Yet even in Haskell, imperative data-structures and algorithms are sometimes important for achieving good performance; hence the popularity of monads [22], and Peyton-Jones' claim that Haskell is "the world's finest imperative programming language" [14]. Thus, in this paper, we shall show how QuickCheck can be used to test monadic Haskell code, and especially imperative code using the ST monad [15].

The key problem is constructing specifications of monadic code that can be used directly for testing. Although this is not as straightforward as for pure functions, we shall show that by borrowing concepts from programming language theory, we can construct testable specifications based on algebraic laws, abstract models, or pre- and postconditions. We introduce a new *monadic property language*, which simplifies these specifications considerably. Our main examples are an imperative implementation of queues, and the Union/Find algorithm.

The structure of the paper is as follows. In the next section, we introduce QuickCheck as previously described. In section 3 we introduce the queue example. Section 4 presents an algebraic specification of imperative queues, and shows how it can be used for testing, introducing the key idea of testing observational equivalence. The next section repeats the exercise for a model-based specification instead. Section 6 draws lessons from the queue example, in particular the desirability of extending QuickCheck with a language of monadic properties. The next three sections introduce this language, present a formal semantics, and sketch its implementation. We illustrate the monadic property language by testing the Union/Find algorithm in the next three sections, which introduce the algorithm, test it using pre- and post-conditions, and using a relational model-based specification respectively. Specifications can become quite bulky; section 13 illustrates an approach to simplifying them, by defining even higher-level combinators for model-based specification of imperative ADTs, which make such specifications very concise. Finally, section 14 discusses related work, and section 15 concludes.

2. BACKGROUND: AN OVERVIEW OF QUICKCHECK

QuickCheck is used by adding property definitions to the program code, either together with the code under test, or in separate "specification" modules. Properties are simply Haskell definitions, such as the following:

$$
\begin{aligned}
&\mathit{prop\_PlusAssoc} :: \mathit{Int} \rightarrow \mathit{Int} \rightarrow \mathit{Int} \rightarrow \mathit{Bool} \\
&\mathit{prop\_PlusAssoc}\ x\ y\ z \ = \ (x+y)+z == x+(y+z)
\end{aligned}
$$

Properties are implicitly universally quantified over their arguments, so this property states the associativity of integer addition.

Properties are tested by passing them to the function *quickCheck*. For example, using the Hugs interpreter we would test the property above by

$$Main\!>\ quickCheck\ prop\_PlusAssoc$$
$$OK,\ passed\ 100\ tests.$$

This tests the property in one hundred randomly generated cases. The function *quickCheck* is overloaded, to accept properties with any number of arguments, and we also provide a small script which invokes *quickCheck* on every property in a module (property names begin with "*prop_*" just so that this script can find them).

The QuickCheck property language also provides *conditional properties*. For example,

$$\begin{aligned}&prop\_InsertOrdered :: Int \to [Int] \to Property\\&prop\_InsertOrdered\ x\ xs = ordered\ xs \implies\\&\quad ordered\ (insert\ x\ xs)\end{aligned}$$

states that inserting an element into an ordered list produces an ordered list. Testing such a property *discards* test cases which do not satisfy the precondition, to guarantee that we test the conclusion with 100 ordered lists. Notice that the result type is different in this case: $\implies$ isn't a simple boolean operator since it affects the selection of test cases. All such operators in QuickCheck have the result type *Property*.

Alternatively, rather than quantify over all lists and then select the ordered ones, we can quantify explicitly over ordered ones:

$$\begin{aligned}&prop\_InsertOrdered :: Int \to Property\\&prop\_InsertOrdered\ x = forAll\ orderedLists\ \$\ \lambda\, xs \to\\&\quad ordered\ (insert\ x\ xs)\end{aligned}$$

This makes for more efficient testing and, often, better test coverage.

The first argument of *forAll* above, *orderedLists*, is a *test data generator*. We can think of it as representing a set that we quantify over, or, more precisely, a probability distribution over such a set. QuickCheck also provides a test data generation language for defining such sets. Test data generators have types of the form $Gen\ \tau$, where *Gen* is a monad, which enables us to use Haskell's monadic syntactic sugar and rich library of monadic operators in test data generator definitions. In addition, there are combinators for making choices between alternatives, such as

$$oneof :: [Gen\ \alpha] \to Gen\ \alpha$$

which chooses between alternatives with equal probability, or

$$frequency :: [(Int, Gen\ \alpha)] \to Gen\ \alpha$$

which attaches a weight to each alternative.

We can define a *default* test data generator for each type as an instance of the class *Arbitrary*:

$$\begin{aligned}&\mathbf{class}\ Arbitrary\ \alpha\\&\quad \mathbf{where}\\&\qquad arbitrary :: Gen\ \alpha\end{aligned}$$

QuickCheck provides instances of this class for all of Haskell's standard types. These default generators are used to construct top-level property arguments, or of course if the programmer writes *forAll arbitrary* explicitly.

QuickCheck provides several functions for making *observations* of test data, of which the most important is *collect*. For example, we could observe the lengths of lists in *prop_InsertOrdered* by redefining it as

$$\begin{aligned}&prop\_InsertOrdered\ x\ xs = ordered\ xs \implies\\&\quad collect\ (length\ xs)\ \$\ ordered\ (insert\ x\ xs)\end{aligned}$$

which causes a table showing the distribution of *length xs* in the actual test data to be displayed when testing is complete. The programmer can then decide whether the test coverage was adequate, or not.

Apart from the small script for extracting and testing properties from modules, QuickCheck is defined entirely as a collection of combinators — it is a *domain specific embedded language* [13]. To use it, the programmer need only import module QuickCheck — which is itself only 300 lines of code! Its lightweight nature makes QuickCheck easy to modify and experiment with.

3. A SIMPLE EXAMPLE: QUEUES

We shall take as an example one of the simplest imperative abstract datatypes, a queue. We assume we are given a module implementing queues, with the signature

$$\begin{aligned}&\mathbf{data}\ Queue\ s\ a = \ldots\\&empty :: ST\ s\ (Queue\ s\ a)\\&add :: Queue\ s\ a \to a \to ST\ s\ ()\\&remove :: Queue\ s\ a \to ST\ s\ ()\\&front :: Queue\ s\ a \to ST\ s\ (Maybe\ a)\end{aligned}$$

whose operations create an empty queue, add an element, remove an element, and return the front element without removing it, if there is one. Queues are implemented in the standard imperative way, so the queue operations have types in the *ST* monad, and the *Queue* type is parameterised on the state thread it belongs to. We omit the details of the implementation.

Of course, there are efficient ways to implement queues purely functionally also [3], but this is not the point: queues serve here simply as an example of an abstract datatype with an imperative implementation. In the following sections we shall see how we can specify their behaviour.

4. TESTING AN ALGEBRAIC SPECIFICATION OF QUEUES

One well established way to specify an abstract datatype is via an *algebraic specification*, in which we characterise the behaviour of our operations by giving equations between terms. For example, if we were specifying Haskell's lists, then one such equation might be

$$xs +\!\!+ (ys +\!\!+ zs) = (xs +\!\!+ ys) +\!\!+ zs$$

In this case, since our operations are monadic, we will give equations between *fragments of monadic code*, that is, program fragments which might appear as part of a **do** block. Our specification of queues is given below.

$$\boxed{\begin{array}{l} q \leftarrow \mathit{empty} \\ x \leftarrow \mathit{front}\ q \end{array}} = \boxed{\begin{array}{l} q \leftarrow \mathit{empty} \\ x \leftarrow \mathit{return}\ \mathit{Nothing} \end{array}} \tag{1}$$

$$\boxed{\begin{array}{l} q \leftarrow \mathit{empty} \\ \mathit{add}\ q\ m \\ x \leftarrow \mathit{front}\ q \end{array}} = \boxed{\begin{array}{l} q \leftarrow \mathit{empty} \\ \mathit{add}\ q\ m \\ x \leftarrow \mathit{return}\ (\mathit{Just}\ m) \end{array}} \tag{2}$$

$$\boxed{\begin{array}{l} \mathit{add}\ q\ m \\ \mathit{add}\ q\ n \\ x \leftarrow \mathit{front}\ q \end{array}} = \boxed{\begin{array}{l} \mathit{add}\ q\ m \\ x \leftarrow \mathit{front}\ q \\ \mathit{add}\ q\ n \end{array}} \tag{3}$$

$$\boxed{\begin{array}{l} q \leftarrow \mathit{empty} \\ \mathit{add}\ q\ m \\ \mathit{remove}\ q \end{array}} = \boxed{q \leftarrow \mathit{empty}} \tag{4}$$

$$\boxed{\begin{array}{l} \mathit{add}\ q\ m \\ \mathit{add}\ q\ n \\ \mathit{remove}\ q \end{array}} = \boxed{\begin{array}{l} \mathit{add}\ q\ m \\ \mathit{remove}\ q \\ \mathit{add}\ q\ n \end{array}} \tag{5}$$

Are these equations a complete specification? Do they define the behaviour of every sequence of queue operations? To see that they do, we shall argue that every such sequence can be put into a *normal form* consisting of queue creation, followed by any number of *add* operations, followed by bindings of variables to values (via $x \leftarrow \mathit{return}\ v$). Consider any such sequence, and consider the *first* use of *front* or *remove*, if one exists. If this first use is of *front*, then equations (1) to (3) allow us to remove it, by moving it earlier until it can be replaced by a *return*. Bindings using *return* can then be moved to the end of the sequence of queue operations using the monad laws, and so do not interfere with further application of these equations. If the first use is of *remove*, then equations (4) and (5) enable us to remove it by moving it earlier until it encounters the matching *add* — *provided* there are at least as many *add*s as *remove*s. Indeed, a sequence of queue operations is only *well-formed* if every prefix contains at least as many *add* operations as *remove*s.

But *what do we mean by equality* in these equations? We certainly do *not* mean that the two sides generate the same state when they are run! In particular, the left hand side of equation (4) creates and discards an internal cell in the queue, while the right hand side does not: clearly pointer values and memory contents will differ after the two sequences are executed. Yet the difference is in low-level representations, and cannot be observed by the Haskell programmer.

We therefore focus on the observable behaviour of our code. Fortunately, operational semanticists have already defined a program equivalence based on observations!

> **Definition** A *context* C is a "program with a hole". We write the result of "filling the hole" with a term e as $C[e]$.

> **Definition** We write $p \Downarrow o$ when program p computes an observable result o.

> **Definition** Two terms e and e' are *operationally equivalent* if, for every context C, then $C[e] \Downarrow o$ if and only if $C[e'] \Downarrow o$.

In our case, two program fragments are equivalent if, when they are inserted into any Haskell program, the program output is the same. Yet quantifying over all Haskell programs with holes is impractical, because Haskell is so complex. So instead, we shall restrict our attention to "queue programs", which contain *only* queue operations on a single queue of integers, and moreover are well-formed in the sense discussed above. We shall restrict our observations to the values bound to variables in uses of *front*. Thus we consider two queue-program fragments equivalent if, when they are inserted into any queue program (in a well-formed way), then the results delivered by all the calls of *front* are the same.

In restricting contexts and observations like this, we are making a number of reasonable assumptions. Since *Queue* is an abstract datatype, other operations in the *ST* monad should not interfere with queue operations — i.e. they should commute, since the *Queue* representations are hidden. We are also assuming that operations on *different* queues commute, since there is no sharing between their representations. Given these assumptions, we can ignore other monadic operations and operations on other queues when we reason about the queue of interest. Since the *Queue* operations are polymorphic, we can appeal to parametricity to argue that if the equations are satisfied for *Queue*s of integers, they are satisfied at every type. It is also reasonable to restrict our observations to the results from *front*, again since *Queue* is an abstract data type, and there is no other operator which delivers any other type of value from it. Thus, if two program fragments are operationally equivalent when we restrict our attention to queue programs, then they should be equivalent in Haskell also.

Now we just program operational equivalence using QuickCheck. We must define queue programs: they consist of queue creation followed by a sequence of *Action*s.

```
data Action = Add Int | Remove |
              Front | Return (Maybe Int)
              deriving (Eq, Show)
```

We will not need to represent variable *names* in this example: we just assume that each *Front* and *Return* binds a variable, and we shall observe the sequence of values bound. We define the semantics of action sequences via the function *perform*.

```
perform        ::
  Queue a Int -> [Action] -> ST a [Maybe Int]
perform q []        = return []
perform q (a : as) =
  case a of
    Add n    -> add n q >> perform q as
    Remove   -> remove q >> perform q as
    Front    -> liftM2 (:) (front q) (perform q as)
    Return x -> liftM (x :) (perform q as)
```

We shall need to quantify over actions, or more specifically over *well-formed action sequences*. We must therefore define a test-data generator for these. We define *actions n* as the set of action sequences which are well-formed *after n add operations*, that is, for queues already containing *n* elements. A *Remove* is only possible if the queue is non-empty.

```
actions    :: Num a => a -> Gen [Action]
actions n =
```

$$
\begin{array}{l}
\quad \mathit{oneof}\ ([\mathit{return}\ [],\\
\qquad\qquad \mathit{liftM2}\ (:)\ (\mathit{liftM}\ \mathit{Add}\ \mathit{arbitrary})\\
\qquad\qquad\qquad\qquad (\mathit{actions}\ (n+1)),\\
\qquad\qquad \mathit{liftM}\ (\mathit{Front}\ :)\ (\mathit{actions}\ n)] +\!\!+\\
\qquad\qquad \textbf{if}\ n == 0\ \textbf{then}\\
\qquad\qquad\quad []\\
\qquad\qquad \textbf{else}\\
\qquad\qquad\quad [\mathit{liftM}\ (\mathit{Remove}\ :)\ (\mathit{actions}\ (n-1))])
\end{array}
$$

We also define a function

$$\mathit{delta}\ ::\ [\mathit{Action}]\ \rightarrow\ \mathit{Int}$$

to compute the *change* in the number of queue elements wrought by a sequence of *Actions*.

Now for the definition of operational equivalence, which follows the formal definition exactly: we choose an arbitary context consisting of a prefix and suffix of queue actions, taking care that the complete program is well-formed, and then check that the observations we make are the same in each case.

$$
\begin{array}{l}
(\cong)\quad\ ::\ [\mathit{Action}]\ \rightarrow\ [\mathit{Action}]\ \rightarrow\ \mathit{Property}\\
c\ \cong\ c' =\\
\quad \mathit{forAll}\ (\mathit{actions}\ 0)\ \$\ \lambda\ \mathit{pref}\ \rightarrow\\
\quad \mathit{forAll}\ (\mathit{actions}\ (\mathit{delta}\ (\mathit{pref}\ +\!\!+\ c)))\ \$\ \lambda\ \mathit{suff}\ \rightarrow\\
\quad \textbf{let}\\
\qquad \mathit{observe}\ x\ =\\
\qquad\qquad \mathit{runST}\ (\ \textbf{do}\\
\qquad\qquad\qquad\qquad q\ \leftarrow\ \mathit{empty}\\
\qquad\qquad\qquad\qquad \mathit{perform}\ q\ (\mathit{pref}\ +\!\!+\ x\ +\!\!+\ \mathit{suff}))\\
\quad \textbf{in}\\
\quad \mathit{observe}\ c\ ==\ \mathit{observe}\ c'
\end{array}
$$

Equations (3) and (5) in the specification can now be written directly as QuickCheck properties.

$$
\begin{array}{l}
\mathit{prop\_FrontAdd}\ m\ n\ =\\
\quad [\mathit{Add}\ m,\ \mathit{Add}\ n,\ \mathit{Front}]\ \cong\ [\mathit{Add}\ m,\ \mathit{Front},\ \mathit{Add}\ n]\\
\mathit{prop\_AddRemove}\ m\ n\ =\\
\quad [\mathit{Add}\ m,\ \mathit{Add}\ n,\ \mathit{Remove}]\ \cong\ [\mathit{Add}\ m,\ \mathit{Remove},\ \mathit{Add}\ n]
\end{array}
$$

The other three equations relate fragments that begin by creating an empty queue, and can thus only appear at the start of a queue program. We therefore need a slightly different notion of operational equivalence.

$$
\begin{array}{l}
c\ \cong^{\wedge}\ c'\ =\\
\quad \mathit{forAll}\ (\mathit{actions}\ (\mathit{delta}\ c))\ \$\ \lambda\ \mathit{suff}\ \rightarrow\\
\qquad \textbf{let}\\
\qquad\quad \mathit{observe}\ x\ =\\
\qquad\qquad\quad \mathit{runST}\ (\ \textbf{do}\\
\qquad\qquad\qquad\qquad q\ \leftarrow\ \mathit{empty}\\
\qquad\qquad\qquad\qquad \mathit{perform}\ q\ (x\ +\!\!+\ \mathit{suff}))\\
\qquad \textbf{in}\\
\qquad \mathit{observe}\ c\ ==\ \mathit{observe}\ c'
\end{array}
$$

The remaining equations are now easily stated.

$$
\begin{array}{l}
\mathit{prop\_FrontEmpty}\ =\\
\quad [\mathit{Front}]\ \cong^{\wedge}\ [\mathit{Return}\ \mathit{Nothing}]\\
\mathit{prop\_FrontAddEmpty}\ m\ =\\
\quad [\mathit{Add}\ m,\ \mathit{Front}]\ \cong^{\wedge}\ [\mathit{Add}\ m,\ \mathit{Return}\ (\mathit{Just}\ m)]\\
\mathit{prop\_AddRemoveEmpty}\ m\ =\\
\quad [\mathit{Add}\ m,\ \mathit{Remove}]\ \cong^{\wedge}\ []
\end{array}
$$

The properties can now be checked by QuickCheck. As expected, they all succeed.

5. TESTING A MODEL-BASED SPECIFICATION OF QUEUES

In the previous section, we implemented an algebraic specification of queues as QuickCheck properties. But specifications come in other flavours too. In this section, we implement a specification based on an *abstract model* of queues. The popular Z specification language, for example, is based on such models [11].

We shall model the state of a queue as a list of the stored elements. With this model, the queue operations are very easy to define. We give them names subscripted by S, since they will serve as specifications. Note that each (except *empty*) returns both a "result" and a new queue.

$$
\begin{array}{ll}
\mathit{empty}_S & = []\\
\mathit{add}_S\ a\ q & = ((),\ q\ +\!\!+\ [a])\\
\mathit{remove}_S\ (\_ : q) & = ((),\ q)\\
\mathit{front}_S\ [] & = (\mathit{Nothing},\ [])\\
\mathit{front}_S\ (a : q) & = (\mathit{Just}\ a,\ a : q)
\end{array}
$$

Now, to formulate the correctness of the implementation, we must relate the implementation state to the abstract model. A standard way to do so is to define an *abstraction function*, which maps the implementation state to the abstract value that it represents. Our abstraction function must, of course, be monadic.

$$\mathit{abstract}\ ::\ \mathit{Queue}\ s\ a\ \rightarrow\ \mathit{ST}\ s\ [a]$$

We omit the (easy) definition; note only that *abstract* must not *change* the implementation state in any way.

An implementation is *correct* if it *commutes with abstract*: that is, if the answer delivered is the same answer that the specification delivers on the abstraction of the initial state, and the final state, when abstracted, is the same as the final state produced by the specification from the abstraction of the initial state.

$$
\begin{array}{l}
\mathit{commutes}\qquad ::\ \mathit{Eq}\ a\ \Rightarrow\\
\quad \mathit{Queue}\ s\ \mathit{Int}\ \rightarrow\ (\mathit{Queue}\ s\ \mathit{Int}\ \rightarrow\ \mathit{ST}\ s\ a) \rightarrow\\
\qquad ([\mathit{Int}]\ \rightarrow\ (a, [\mathit{Int}]))\ \rightarrow\ \mathit{ST}\ s\ \mathit{Bool}\\
\mathit{commutes}\ q\ a\ f\ =\ \textbf{do}\\
\qquad\qquad\qquad\qquad \mathit{old}\ \leftarrow\ \mathit{abstract}\ q\\
\qquad\qquad\qquad\qquad x\ \leftarrow\ a\ q\\
\qquad\qquad\qquad\qquad \mathit{new}\ \leftarrow\ \mathit{abstract}\ q\\
\qquad\qquad\qquad\qquad \mathit{return}\ ((x, \mathit{new}) == f\ \mathit{old})
\end{array}
$$

Of course, this condition must hold in all states. But what do we mean by "all states"? Certainly not all memory states: the references that make up a queue representation must satisfy a strong invariant — they must be linked together in a linear chain, without loops, the head node must point at the first and last element nodes, and so on. Trying to generate a random heap state, with a rats' nest of references, and then select those that represent queues, would be both difficult and hopeless in practice. Let us instead quantify over *reachable states*, that is, states which can actually be produced by a sequence of queue operations. These automatically satisfy the queue invariant, they can be represented naturally by the sequence of operations which constructs them, and they are the only states of interest anyway! Fortunately, we have already defined a generator for well-formed sequences of queue operations in the previous section, so now it is easy to define when an operation correctly implements a specification.

```
implements      :: Eq a ⇒
  (∀ s. Queue s Int → ST s a) →
    ([Int] → (a, [Int])) → Property
a `implements` f =
  forAll (actions 0) $ λ as →
    runST ( do
              q ← empty
              perform q as
              commutes q a f)
```

The correctness properties for *add* and *front* are now direct.

```
prop_Add n  =  add n `implements` add_S n
prop_Front  =  front `implements` front_S
```

The *empty* operation does not quite fit this framework, since it *creates* a queue rather than modifying an existing one. We cannot reuse *implements*, but of course it is correct if the representation it constructs abstracts to the empty queue in the specification.

```
prop_Empty =
  runST ( do
            q ← empty
            q' ← abstract q
            return (q' == (empty_S :: [Int])))
```

Finally, the *remove* operation does not quite fit either, because it has a precondition: it can only be applied to non-empty queues. Thus we need a version of *implements* which quantifies over states satisfying a precondition.

```
implementsIf :: Eq a ⇒
  (∀ s. Queue s Int → ST s Bool) →
  (∀ s. Queue s Int → ST s a) →
  ([Int] → (a, [Int])) →
      Property
implementsIf pre a f =
  forAll (actions 0) $ λ as →
    runST ( do
              q ← empty
              perform q as
              pre q) ⟹
    runST ( do
              q ← empty
              perform q as
              commutes q a f)
```

Now we can complete our specification:

```
prop_Remove =
  implementsIf (liftM isJust ∘ front) remove remove_S
```

Once again, all properties succeed.

6. LESSONS FROM THE QUEUE EXAMPLE

We have shown how to represent two popular kinds of specification as QuickCheck properties, and thus use them directly for testing. The key idea for coping with the monadic nature of the implementation was to define a "queue program language", represented as a Haskell datatype, and quantify over contexts. This enabled us to implement directly the definition of operational equivalence for testing the algebraic specification, and to generate random reachable states to test the model-based one.

The reader may wonder why we did not represent program fragments as monadic values — as semantics, rather than abstract syntax. In principle this may seem attractive, but there are major advantages in using a datatype.

- When tests fail, the values of quantified variables are displayed. If we quantify over contexts, then we see the abstract syntax tree, which is, of course, very useful. The semantics of a context is a function, however: if we quantified over this instead then we would see no useful information on a test failure.
- As well as running program fragments, we may wish to compute some of their properties by *static analysis*, which requires an abstract syntax tree. A simple example of this is the *delta* function, used above to predict the changes in the number of elements in the queue, when a queue program fragment is run.
- When we test code using the ST monad, then computations have polymorphic types, and functions over them must have rank-2 types. Examples are the *implements* and *implementsIf* functions in the previous section. These types can be quite complex, and moreover must always be stated explicitly. By passing abstract syntax trees instead, we avoid the need for most rank-2 types.

Indeed, there is an even more severe problem. If we were to quantify over the semantics of contexts in the ST monad, by writing *forAll contexts*$(\lambda\ c\ \rightarrow\ \dots)$, then since c must have a polymorphic type, the λ-expression must have a rank-2 type, and *forAll* must be used at an instance with a rank-3 type! This goes beyond what today's implementations can support: Hugs allows only rank-2 types, and although GHC now supports rank-k types (using Odersky and Läufer's work [20]), the type system is still *predicative*, which means that type variables cannot be instantiated to anything other than monotypes. To quantify over values involving the ST monad, we would need to define a special version of *forAll*, with an explicitly stated rank-3 type. Moreover, we would need a *different* version of *forAll* for each type of value quantified over, so these versions cannot reasonably be placed in the QuickCheck library; they must be defined by the *user* of QuickCheck, which is unacceptable. Thus representing contexts by abstract syntax is essential to making our approach work at all.

So far, we have used only QuickCheck as described in our original paper [4], and it has worked pretty well; we have needed *no* extensions specific to monads. However, the shoe does pinch a little. Look back at the definition of *implementsIf* (for testing an operation with a precondition): in order to check that the state generated by the random context satisfies the precondition, we had to run the code generating it twice! We cannot write, for example,

```
pre q ⟹ commutes q a f
```

inside a single call of *runST*, because $\Longrightarrow$ is a *property* combinator, and has the wrong type to appear in an ST computation. We cannot either write

```
b ← pre q
if b then
  commutes q a f
else
  return True
```

because this has a different meaning: it counts a test in which the precondition is not satisfied as a *successful test*, which is not what we want at all!

The problem is that we cannot use *property operators* in the midst of a monadic computation — and sometimes, that is exactly what we want to do. Other examples would be quantifying over the elements of a list produced by a monadic computation, or collecting values generated in monadic code. This motivates extending QuickCheck with a language of *monadic properties*: the subject of the next section.

7. A LANGUAGE OF MONADIC SPECIFICATIONS

Our goal is to extend QuickCheck with a new kind of property, which can contain monadic computations in an underlying monad m. We therefore define a *property monad* $PropertyM\ m\ a$, whose elements may mix property operations and m-computations. This is really just a monad transformer [16], whose lifting operation we call

$$run :: Monad\ m \Rightarrow m\ a \rightarrow PropertyM\ m\ a$$

Non-monadic properties can be embedded in monadic ones using

$$\begin{aligned}&assert :: (Monad\ m,\ Testable\ a) \Rightarrow\\ &\quad a \rightarrow PropertyM\ m\ ()\end{aligned}$$

(where *Testable* types are those corresponding to properties in vanilla QuickCheck). An assertion must hold when the monadic property is tested.

Preconditions can be included in monadic properties using

$$pre :: Monad\ m \Rightarrow Bool \rightarrow PropertyM\ m\ ()$$

Test cases in which preconditions fail are discarded, just like when using the implication combinator $\Longrightarrow$.

Using these operations, we can represent a Hoare triple $\{p\}x \leftarrow e\{q\}$ as

$$\begin{aligned}&pre\ p\\ &x \leftarrow run\ e\\ &assert\ q\end{aligned}$$

We can also think of *run* as a monadic *weakest precondition* operator: we could define

$$\begin{aligned}&wp \quad :: Monad\ m \Rightarrow\\ &\quad m\ a \rightarrow (a \rightarrow PropertyM\ m\ b) \rightarrow PropertyM\ m\ b\\ &wp\ m\ k = run\ m \gg\!= k\end{aligned}$$

and represent the weakest precondition $\mathsf{wp}(x \leftarrow e, p)$ as $wp\ e\ \$\ \lambda\,x \rightarrow p$.

We represent quantification in monadic properties by either using

$$\begin{aligned}&pick :: (Monad\ m,\ Show\ a) \Rightarrow\\ &\quad Gen\ a \rightarrow PropertyM\ m\ a\end{aligned}$$

or, for more familiar notation,

$$forAllM\ gen\ k\ =\ pick\ gen \gg\!= k$$

The choice between *pick*/*forAllM* and *run*/*wp* is a matter of taste: the latter operations resemble mathematical notation more closely, while the former let us take advantage of Haskell's **do** syntactic sugar.

We can make observations of test data using

$$\begin{aligned}&monitor :: Monad\ m \Rightarrow\\ &\quad (Property \rightarrow Property) \rightarrow PropertyM\ m\ ()\end{aligned}$$

For example, $monitor\ (collect\ e)$ collects the distribution of values of e. Finally, we can convert monadic properties back to ordinary ones, given a "run function" for the underlying monad, using

$$\begin{aligned}&monadic \quad :: Monad\ m \Rightarrow\\ &\quad (m\ Property \rightarrow Property) \rightarrow\\ &\qquad PropertyM\ m\ () \rightarrow Property\\ &imperative :: (\forall\ b.\ PropertyM\ (ST\ b)\ ()) \rightarrow Property\end{aligned}$$

imperative is equivalent to $monadic\ runST$, except that the latter would need impredicative rank-3 types and so cannot be written.

Using these operations, we can revisit *implementsIf* and rewrite it as follows:

$$\begin{aligned}&implementsIf\ p\ a\ f\ =\ imperative\ (\\ &\quad forAllM\ (actions\ 0)\ \$\ \lambda\ as\ \rightarrow\\ &\qquad \mathbf{do}\\ &\qquad\quad q \leftarrow run\ empty\\ &\qquad\quad run\ (perform\ q\ as)\\ &\qquad\quad ok \leftarrow run\ (p\ q)\\ &\qquad\quad pre\ ok\\ &\qquad\quad b \leftarrow run\ (commutes\ q\ a\ f)\\ &\qquad\quad assert\ b)\end{aligned}$$

The repeated execution needed in the original version to test the precondition is gone.

8. SEMANTICS OF MONADIC PROPERTIES

QuickCheck properties enjoy both a computational and a declarative reading, in which generators really denote sets, $\Longrightarrow$ is true implication, and *forAll* is true quantification. In the declarative reading, a property just denotes a truth value (not necessarily computable). Of course, non-termination in a property may make a declarative reading impossible, but we restrict ourselves here to terminating programs whose semantics can be modelled using sets and functions, rather than domains and continuous functions. Even if there is a mismatch here, the declarative reading is the "intended semantics" which our Haskell implementation approximates.

But what is the declarative reading of a monadic property? What is the logic which we are trying to represent? Of course, monadic properties may be based on *any* monad, not just the familiar state one. The meaning of properties when the underlying monad permits backtracking through preconditions, for example, or concurrency, is far from obvious. In this section, we give a formal semantics to the monadic property language which answers such questions.

We model monadic properties over a monad M as *non-empty sets of computations of the type $M\ Bool$*. We use sets to model quantification: a property $forAllM\ s\ p$ is modelled by a set constructed from s. Given a satisfaction test for the monad, $test_M :: M\ Bool \rightarrow Bool$, a monadic property is satisfied if *every computation in the set delivers True when it is tested.* Different choices for $test_M$ lead to different interpretations of properties — for example, if M is the list monad (representing backtracking computations), then $test_M$ might require that *all* possible results are *True*, that *the first* result is *True*, or that *some* result is *True*. If M is the *Maybe* monad, then $test_M$ might interpret *Nothing* as *True* (testing for *partial correctness*) or *False* (*total correctness*). We require only that $test_M\ (return\ b)\ =\ b$.

Now, without loss of generality, we can assume (because of their type) that monadic properties end in *return* (). Such a property is trivially satisfied. (Note that it is solely the assertions made during a monadic computation which introduce truth values to be checked; the result of the computation plays no role and is therefore irrelevant.)

$$[\![return()]\!] \ = \ \{return\ True\}$$

Otherwise, a property takes the form of $m \gg\!\!= k$ for some m and k. An assertion returns *False* if it is not satisfied:

$$\begin{aligned}[\![assert\ \ True \gg p]\!] &= [\![p]\!] \\ [\![assert\ \ False \gg p]\!] &= \{return\ False\}\end{aligned}$$

A precondition returns *True* if it is *False*.

$$\begin{aligned}[\![pre\ \ True \gg p]\!] &= [\![p]\!] \\ [\![pre\ \ False \gg p]\!] &= \{return\ True\}\end{aligned}$$

Quantification derives a set of computations from each element of the set quantified over, and merges them — unless the set we quantify over is empty, when it succeeds at once (to ensure that the meaning of the property is a *non-empty* set).

$$\begin{aligned}[\![pick\ \emptyset \gg\!\!= k]\!] &= \{return\ True\} \\ [\![pick\ s \gg\!\!= k]\!] &= \{m \mid x \in s, m \in [\![k\ x]\!]\},\ \text{if } s \neq \emptyset\end{aligned}$$

Finally, running a computation of type $M\ \tau$ is interpreted as

$$[\![run\ m \gg\!\!= k]\!] \ = \ \{m \gg\!\!= k' \mid \ k' \in \tau \to M\ Bool,\ \forall x.k'\ x \in [\![k\ x]\!]\}$$

Here k represents a function from τ to a *set* of computations, and k' is a function which makes a choice from each such set. It is to make this possible that we require the meaning of a property to be a *non-empty* set. The effect of this definition is that, if there is quantification in k (perhaps depending on the result delivered by m), then every possible choice is represented by some $m \gg\!\!= k'$ in the resulting set.

With these definitions, monadic properties have a well-defined meaning no matter *what* the underlying monad is.

To understand these definitions a little bit better, it is useful to instantiate them for a particular monad. It is easy to see for example that taking the identity monad simply reverts to basic QuickCheck properties. Let us also look at a more elaborate example: As the underlying monad, we take the list monad, and we look at the following somewhat artificial property which depends on an unknown predicate p:

```
do
  x ← run [1,2]
  y ← pick (elements [x + 3, x + 4])
  assert (p x y)
```

(For convenience, we have left out the finishing *return* ().)

The operational reading is: We make two computations, one where x equals 1, and the other where x equals 2. In both computations, we pick an arbitrary element from the list $[x+3, x+4]$ as the value for y, and finally we return the two-list of the results.

The declarative reading of the two simple subexpressions should also be clear:

$$[\![assert(p\ x\ y)]\!] \ = \ \{[p\ x\ y]\}$$

$$[\![pick(elements[x+3, x+4]) \gg\!\!= \lambda y \to assert(p\ x\ y)]\!] \ = \ \{[p\ x\ (x+3)], [p\ x\ (x+4)]\}$$

The declarative reading of the whole property is:

$$\{k'\ 1\ +\!\!+\ k'\ 2 \mid \forall x.k'\ x \in \{[p\ x\ (x+3)], [p\ x\ (x+4)]\}$$

This last expression simplifies to:

$$\{[p\ 1\ y_1,\ p\ 2\ y_2] \mid y_1 \in \{1+3, 1+4\},\ y_2 \in \{2+3, 2+4\}\}$$

Which is the same as:

$$\{[p\ 1\ 4,\ p\ 2\ 5],\ [p\ 1\ 4,\ p\ 2\ 6],\ [p\ 1\ 5,\ p\ 2\ 5],\ [p\ 1\ 5,\ p\ 2\ 6]\}$$

Exactly what we expected! It then depends on the chosen $test_{[]}$ function (and on p of course) which of the elements in the set actually pass the test or not, and thus whether all elements in the set pass the test.

9. IMPLEMENTING MONADIC PROPERTIES

The implementation of QuickCheck is based on the monad *Gen*, an abstract type defined by

$$\textbf{newtype}\ Gen\ a\ =\ Gen\ (Int\ \to\ StdGen\ \to\ a)$$

Essentially a *Gen a* is a function from a random number seed to an a: the *Int* parameter is used to control the size of generated data and need not concern us here. QuickCheck properties are just generators for test results

$$\textbf{newtype}\ Property\ =\ Prop\ (Gen\ Result)$$

where the *Result* type collects quantified variables, preconditions, and monitoring information as well as representing success or failure.

Monadic properties are built by combining *Gen* and a CPS monad with the underlying monad m.

$$\begin{aligned}&\textbf{newtype}\ PropertyM\ m\ a\ = \\ &\quad Monadic\ ((a\ \to\ Gen\ (m\ Result))\ \to \\ &\qquad\qquad Gen\ (m\ Result))\end{aligned}$$

Using CPS enables *pre* and *assert* to discard the rest of a property when their argument is false.

Given this type, the rest of the implementation is mostly straightforward, and follows the semantics closely; indeed, we added only about 30 lines of code to QuickCheck, and did not need to change any existing code at all. The only tricky part is the definition of *run*:

$$run\ m\ =\ Monadic\ (\lambda\, k\ \to\ liftM\ (m\ \gg\!\!=)\ (promote\ k))$$

Here the continuation k is of type $a\ \to\ Gen\ (m\ Result)$, but before we apply $liftM\ (m\ \gg\!\!=)$ to it, we must convert it to a $Gen\ (a\ \to\ m\ Result)$. Because of the way we defined *Gen* this is simple to do: the *promote* function need only swap the arguments of the function it is passed, to take the random number seed and size first, rather than the a. But this kind of promotion is quite impossible for most monads: indeed, for the monad *Set* (which *Gen* is supposed to represent), *promote* corresponds to applying the Axiom of Choice! No wonder this seemingly simple definition is somewhat counter-intuitive.

```
data Element s a = Element a (STRef s (Link s a))
data Link s a = Weight Int | Next (Element s a)

newElement :: a -> ST s (Element s a)
newElement a = do
                 r <- newSTRef (Weight 1)
                 return (Element a r)
findElement :: Element s a -> ST s (Element s a)
findElement (Element a r) =
  do
    e <- readSTRef r
    case e of
      Weight w -> return (Element a r)
      Next next -> do
                     last <- findElement next
                     writeSTRef r (Next last)
                     return last
unionElements ::
  Element s a -> Element s a -> ST s ()
unionElements e1 e2 =
  do
    Element a1 r1 <- findElement e1
    Element a2 r2 <- findElement e2
    Weight w1 <- readSTRef r1
    Weight w2 <- readSTRef r2
    if w1 <= w2 then
      do
        writeSTRef r1 (Next (Element a2 r2))
        writeSTRef r2 (Weight (w1 + w2))
    else
      do
        writeSTRef r2 (Next (Element a1 r1))
        writeSTRef r1 (Weight (w1 + w2))
instance Eq (Element s a)
  where
    Element _ r == Element _ r' = r == r'
```

Figure 1: The Union-Find Algorithm.

10. ANOTHER EXAMPLE: THE UNION/ FIND ALGORITHM

As an example which makes extensive use of monadic properties, we shall test the Union/Find algorithm. This is a very efficient way to represent an equivalence relation. Elements of the relation are organised into trees representing equivalence classes, with each element containing a pointer to its parent. By following these pointers to the root of each tree, we can find a distinguished element of each equivalence class; the operation which does so is called *find*. We can test whether two elements are equivalent by comparing the results of *find* on each one. Equivalence classes can also be merged by declaring two elements to be equivalent: this is done by the function *union*, and achieved by making the root of one tree point at the root of the other.

The Union/Find algorithm owes its great efficiency to two optimisations:

- After *find* has traversed a path to the root of a tree, it updates all the elements in the path to point directly at the root. This speeds up subsequent *find*s.
- When trees are merged, the root of the *lighter* tree is made to point at the root of the *heavier*, where the weight of a tree is the number of elements in it. This also speeds up subsequent *find*s.

With these optimisations, a sequence of *union* and *find* operations is executed in almost linear time (where "almost" involves the inverse of the Ackermann function, so for all practical purposes we can consider the time to be linear).

A Haskell implementation of the Union/Find algorithm is very simple; one appears in Figure 1. Elements are represented by the type *Element*, contain a value (so we can represent equivalence relations on other types), and are created by the function *newElement*. The *find* and *union* operations are implemented by *findElement* and *unionElements*. Finally, *Element*s can be compared, so we can decide whether two results of *findElement* are the same. Elements contain an updateable *Link*, which in the case of root nodes contains a weight, and for other nodes contains the parent.

11. TESTING PRE- AND POSTCONDITIONS FOR UNION/FIND

We shall test our Union/Find implementation using yet a third method: be specifying pre- and post-conditions for each operation. With this approach, we need neither an abstract model, nor algebraic laws. But we will still need to quantify over reachable states. As before, we define a language of union/find programs.

```
data Action = New | Find Var | Union Var Var
  deriving Show
type Var = Int
```

A program is a list of *Action*s, which may create, find, or unite elements. The arguments of *findElement* and *unionElements* may be any element previously created by *newElement*; we use natural numbers to refer to them in order of creation. The semantics of action sequences is defined by

```
exec :: [Action] -> [Element a ()] ->
          ST a [Element a ()]
```

which delivers as its result a list of the *Element*s created by *newElement*.

Of course, only certain union/find programs are *well-formed*: we must not use an *Element* which has not been created. We therefore define a generator for the set of programs well-formed in the context of k elements.

```
actions   :: Int -> Gen [Action]
actions 0 =
  frequency [(25, liftM (New :) (actions 1)),
             (1, return [])]
actions n =
  frequency
    [(2, liftM (New :) (actions (n + 1))),
     (2, liftM2 (:) (liftM Find element)
                    (actions n)),
     (2, liftM2 (:) (liftM2 Union element
                                  element)
                    (actions n)),
     (1, return [])]
  where
    element = choose (0, n - 1)
```

When the number of elements is zero, the only possible action is *New*: we give this a high probability, to avoid a large number of tests in the initial state. Similarly, we assign a higher probability to choosing an operation than to returning the empty list: we can expect to generate action sequences with an average length of 7 using this definition.

Now we can define a combinator for quantifying over all states.

```
forAllStates ::
  (∀ s. [Element s ()] → PropertyM (ST s) a) →
    Property
forAllStates p =
  forAll (actions 0) $ λ as →
    imperative ( do
                   vars ← run (exec as [])
                   p vars)
```

We pass the property p a list of all created *Elements*; in most properties we need to quantify over the elements of this list.

This quantification poses a problem, though. QuickCheck's quantification operators can only quantify over types in class *Show*, since the value chosen must be displayed when a test fails. But *Elements* cannot be *shown*, since they contain *STRefs*, and this is an abstract type for which *show* is not defined. Of course, we could define our own *Show* instance to display references as "< STRef >", but this would not be useful! We *want* to know which element was chosen when a test fails!

Our solution to the "abstract type quantification" problem is to quantify over an element's *position in a list* instead: as long as we know how the list is constructed, we can infer which element was used. In this case, we use the list of created *Elements* passed to properties by *forAllStates*. We define a function

```
pickElement :: Monad m ⇒ [a] → PropertyM m a
pickElement vars =
  do
    pre (not (null vars))
    i ← pick (choose (0, length vars − 1))
    return (vars !! i)
```

which quantifies over this list, and imposes a precondition that it be non-empty.

Now we just need to characterise the behaviour of *findElement* and *unionElements* using pre- and postconditions. We will need to refer to the distinguished representative of each equivalence class, so we define

```
representative :: Element a b → ST a (Element a b)
```

to find it. Of course, this function delivers the same result as *findElement*, *but without a side effect.* It is just for use in formulating properties.

Let us begin! Firstly, *findElements* returns the representative of its argument.

```
prop_FindReturnsRep =
  forAllStates (λ vars →
    do
      v ← pickElement vars
      r ← run (representative v)
      r′ ← run (findElement v)
      assert (r == r′))
```

Secondly, *findElement* does not change the representative of any element.

```
prop_FindPreservesReps =
  forAllStates (λ vars →
    do
      (v, v′) ← two (pickElement vars)
      r0 ← run (representative v)
      r′ ← run (findElement v′)
      r1 ← run (representative v)
      assert (r0 == r1))
```

Thirdly, *unionElements* does not change the representatives of elements which were *not* previously equivalent to one of its arguments.

```
prop_UnionPreservesOtherReps =
  forAllStates (λ vars →
    do
      (v0, v1, v2) ← three (pickElement vars)
      [r0, r1, r2] ←
          run (mapM representative [v0, v1, v2])
      pre (r0 ≠ r1 ∧ r0 ≠ r2)
      run (unionElements v1 v2)
      r0′ ← run (representative v0)
      assert (r0 == r0′))
```

Finally, *unionElements* really does unite equivalence classes. We express this by stating that all the elements of the equivalence class of either argument have the same representative afterwards.

```
prop_UnionUnites =
  forAllStates (λ vars →
    do
      (v1, v2) ← two (pickElement vars)
      c1 ← run (equivClass vars v1)
      c2 ← run (equivClass vars v2)
      run (unionElements v1 v2)
      c1′ ← run (mapM representative c1)
      c2′ ← run (mapM representative c2)
      assert (length (nub (c1′ ++ c2′)) == 1))
  where
    equivClass vars v = filterM (≡ v) vars
    e1 ≡ e2 = liftM2 (==) (representative e1)
                          (representative e2)
```

We claim that these properties are easy to read and write. Moreover, note that we have taken great advantage of the monadic property language: preconditions, quantifications, and computations are thoroughly mixed in these properties.

Let us test one more property: the "weight invariant" stating that each root node contains a weight equal to the number of elements which it represents.

```
prop_WeightInvariant =
  forAllStates (λ vars →
    do
      v ← pickElement vars
      r@(Element _ link) ← run (representative v)
      Weight w ← run (readSTRef link)
      rs ← run (mapM representative vars)
      assert (w == length (filter (== r) rs)))
```

This property is not necessary for correctness, but it is for efficiency. Surprisingly, when we *quickCheck* it, it fails! After a few tries to find a small counter-example, we find

```
UnionFind > quickCheck prop_WeightInvariant
Falsifiable, after 3 tests :
```

```
[NewElement, UnionElements 0 0]
0
```

This tells us that the weight of element 0 is wrong after it is unioned with itself. Inspecting the code of *unionElements*, we quickly see why: we forgot to consider the case when the two arguments *are already equivalent*. In that case, we need do nothing — and in particular, the weight should not be updated. Adding this special case makes all properties go through.

12. TESTING A MODEL-BASED SPECIFICATION OF UNION/FIND

Just as we tested queues using a specification based on an abstract model, we can test the Union/Find algorithm in the same way. We shall model elements by natural numbers in the range $0 \ldots k$, and the state by a function *repr* from $\{0 \ldots k\}$ to itself, which maps elements to their representative. We can conveniently represent such a function in Haskell by a list (so we apply it to an element x by writing *repr* !! x). We define an abstraction function to recover the abstract state.

```
abstract      :: [Element a b] -> ST a [Int]
abstract vars = mapM abs vars
  where
    abs v = do
              r <- representative v
              return (position vars r)
```

where *position* returns the position of an element in a list.

The abstract state must satisfy an invariant: *repr* ∘ *repr* must equal *repr*. We write

```
prop_Invariant = forAllStates (\vars ->
  do
    repr <- run (abstract vars)
    assert (repr == map (repr!!) repr))
```

Now, notice that (as far as correctness is concerned) it does not matter whether *union* makes its first argument point to its second, or vice versa. Rather than specify this behaviour exactly, we shall use *relational* specifications which leave some freedom to the implementor. Thus we specify our operations via a *predicate* which must hold on the inputs, initial state, output, and final state, rather than by giving a function from the former to the latter. The specifications of *find* and *union* are easy to write:

```
findS x repr y repr' =
  repr == repr' && y == repr !! x

unionS x y repr () repr' =
  let
    z = repr' !! x
  in
  (z == repr !! x || z == repr !! y) &&
  repr' == [ if z' == repr !! x || z' == repr !! y then
               z
             else
               z' | z' <- repr]
```

These specifications closely resemble Z schemas [11].

We define a combinator expressing that a monadic computation implements such a specification:

```
implements vars m s =
  do
    repr  <- run (abstract vars)
    ans   <- run m
    repr' <- run (abstract vars)
    assert (s repr ans repr')
```

Now it only remains to state that *findElement* and *unionElements* implement the specifications above, for all choices of elements. The only (slight) complication is that we must convert elements from their concrete to their abstract representation (using *position vars*) before we can compare implementation and specification.

```
prop_Find = forAllStates (\vars ->
  do
    v <- pickElement vars
    implements vars
        (liftM (position vars) (findElement v))
        (findS (position vars v)))
prop_Union = forAllStates (\vars ->
  do
    (v, v') <- two (pickElement vars)
    implements vars
        (unionElements v v')
        (unionS (position vars v) (position vars v')))
```

This completes the model-based specification: it is pleasingly simple. Indeed, model-based specifications are often simpler than pre- and postcondition specifications such as we gave in the previous section, since the latter are couched in terms of the (generally more complex) implementation state. So why not always use model-based specifications? Firstly, it is useful to be *able* to test pre- and postconditions, since in some cases one may just wish to test a few such properties without going to the trouble of defining a complete abstract model. Secondly, because the pre- and post-condition style is expressed entirely in terms of the implementation state, these properties can often be tested more efficiently than those in the model-based style (although speed is not a problem in these examples).

13. A GENERAL MODEL-BASED SPECIFICATION FRAMEWORK

Fully formal specifications can become quite complex (this is true whether they are used for testing or any other purpose). An advantage of representing them in a language like Haskell, with powerful abstraction mechanisms, is that we can hope to find "higher-level combinators" which make specifications easier to write. In this section we sketch an initial step in this direction: a library for model-based specification of imperative ADTs, which we apply to the queue example once more.

The library is based on two abstract types, the first of which is

```
data Action m spec impl = ...
```

An element of this type represents a concrete operation in the monad m, that works on an implementation type *impl*, and has an abstract functional counterpart of type *spec*. For example, in the case of queues, m is the monad *ST s*, *spec* is the type [*Int*], and *impl* is the type *Queue s Int*; an action might represent the operation *add* 23. However, an action contains both the specification and implementation of an operation, and when executed, tests if the observational outputs of the action are the same. In the queue example, we check that all calls to the implementation of *front* produce the same answer as the specification — this is our correctness criterion.

The function

```
sameOutput :: [Action m spec impl] -> m Bool
```

executes a list of actions in sequence, thus checking that the observable outputs of all actions are the same.

The second abstract type is

```
data Method m spec impl = ...
```

A *Method* represents an *Action* generator — for example, corresponding to *add*, from which the *Action add* 23 can be generated. Methods are constructed using method combinators, as in this example, which specifies the queue methods:

```
methods_Queue :: [Method (ST s) [Int] (Queue s Int)]
methods_Queue =
  [name "empty" $ methodInit [] empty,
   name "add" $ arg arbitrary $ \x ->
        method1 add_S add,
   name "front" $ method1 front_S front,
   name "remove" $
        method1Pre (not . null) remove_S remove]
```

Here, *name* specifies the name of an operator (for debugging output). The method constructor *methodInit* specifies a method that creates an object, *method*1 specifies a method that transforms one object, *method*1*Pre* specifies a method that has a precondition, and there are other method constructors. The method combinator *arg* is used to specify an argument to a method.

Given such a list of methods, we can generate random sequences of actions which correspond to calls of the methods. This is done by the generator *actions*:

```
actions :: [Method m spec impl] ->
                        Gen [Action m spec impl]
```

This function keeps track of the abstract state when generating the list of actions, and at any point in time only picks methods whose precondition is satisfied. The current version of *actions* makes a property fail if it gets to a state where there are no methods which can be performed — when it is "stuck". Of course, it depends on the application if this really is an error or not.

Note that the list of the list of methods denotes a *choice* of methods, whereas the list in the list of actions denotes a *sequence* of actions.

Finally, we check that all generated action sequences produce the same output in the abstract and concrete semantics. This is done by the function *commutes*:

```
commutes :: [Method m spec impl] -> PropertyM m ()
commutes methods =
  forAllM (actions methods) $ \acts ->
    do
      b <- sameOutput acts
      assert b
```

Notice that, in monadic properties, we *can* quantify over the semantics of actions — the "rank-3" problem discussed in section 6 is avoided.

Using the library, a full correctness specification of the queue example looks like this:

```
prop_Queue = imperative (commutes methods_Queue)
```

Together with the definition of *methods_Queue*, this is only a few lines.

14. RELATED WORK

There are two other automated testing tools for Haskell. HUnit is a unit testing framework based on the JUnit framework for Java, which permits test cases to be structured hierarchically into tests which can be run automatically [12]. HUnit allows the programmer to define "assertions", but these apply only to a particular test case, and so do not make up a specification. There is no automatic generation of test cases.

Auburn [18] is a tool primarily intended for benchmarking abstract data types. Auburn generates random "datatype usage graphs" (dugs), corresponding to our "queue programs" etc, and measures the time to evaluate them. Auburn can produce dug generators and evaluators automatically, given the signature of the ADT. It avoids generating ill-formed dugs by tracking an abstract state, or "shadow", for each value of the ADT, and checking preconditions expressed in terms of it before applying an operator. Dug generators are parameterised on the desired frequency of the different operations, size of data to generate, degree of sharing etc, so that benchmarking corresponds as closely as possible to real conditions. Benchmarking can reveal errors in the ADT implementation, but since there is no specification or other test oracle then they are discovered only if they lead to run-time failure.

The Hat tracer for Haskell [25] is not a testing tool, but enables the programmer to browse a computation once it has failed. We are investigating integrating it with QuickCheck, so that the tracer can be invoked when QuickCheck discovers a fault.

The more general testing literature is voluminous.

Random testing dates from the 1960s, and is now used commercially, especially when the distribution of random data can be chosen to match that of the real data. It compares surprisingly favourably in practice with systematic choice of test cases. In 1984, Duran and Ntafos compared the fault detection probability of random testing with partition testing, and discovered that the differences in effectiveness were small [6]. Hamlet and Taylor corroborated the original results [10]. Although partition testing is slightly more effective at exposing faults, to quote Hamlet's excellent survey [9], *"By taking 20% more points in a random test, any advantage a partition test might have had is wiped out."* Our philosophy is to apply random testing at a fine grain, by specifying properties of most functions under test. So even when QuickCheck is used to test a large program, we always test a small part at a time, and are therefore likely to exercise each part of the code thoroughly.

Invoking sequences of operations to test abstract data types is a standard approach (how else could it be done?). Generating random sequences of operations, while still fulfilling all preconditions, is not so common. Our test data generation language, embedded in Haskell, makes this easy. The connection we have drawn between random sequences of operations and the definition of observational equivalence is new.

Algebraic specifications have been used by many authors as a foundation for testing. The first system based on this idea was DAISTS [8], which tested abstract data types by evaluating and comparing the left and right hand sides of equations in the specification, in test cases supplied by the user. Although the language used was imperative, abstract data type operations were forbidden to side-effect their arguments, so the programs to be tested were essentially restricted to be functional.

Later work aims to relax this restriction: Antoy and Hamlet describe a technique for testing C++ classes against an algebraic specification, which is animated in order to predict the correct result [1]. The specification language must be somewhat restricted in order to guarantee that specifications *can* be animated. The concrete and abstract states are related by a programmer-defined abstraction function, just as in this paper. Antoy and Hamlet do not address test case generation, leaving that as a problem for a separate tool.

Bernot, Gaudel, and Marre developed a theory of testing, which formalises the assumptions on which selection of test cases is based [2]. They developed a tool for test case selection based on an algebraic specification.

One unusual feature of the algebraic specifications in this paper is that they relate monadic terms, in which the underlying state is implicit. More commonly in algebraic specifications, the state is an explicit argument and result. (Perhaps this is because algebraic specification frameworks tend to be first order.) Relating *programs* rather than *states* lets us write equations which apply directly to the imperative implementation. We believe we are the first to directly verify such equations by testing: recall that DAISTS was limited to testing pure functions, and Antoy and Hamlet used their equational specification to derive rewrite rules, rather than testing the equations in it directly.

Model-based specifications have also been used as a foundation for testing. Stocks and Carrington developed a framework for deriving *test frames* (characterising a class of test cases) from a Z specification [24]. They derived test frames manually, but Donat has developed an automatic tool for doing so [5]. Model-based specifications have also been used as test oracles. A tool for instrumenting C++ classes to check pre- and post-conditions derived from a model-based specification has been developed by Edwards [7]. Mueller and Korel test C code against a formal specification by translating the specification into code which checks the results of the test, and generating test cases either randomly or using existing test case generators [19]. The case studies used are rather small though — the most complex is the C string copy function.

All of this work requires some preprocessing or analysis of specifications before they can be used for testing. QuickCheck is unique in using specifications *directly*, both for test case generation and as a test oracle. The other side of the coin is that the QuickCheck specification language is necessarily more restrictive than, for example, predicate calculus, since properties must be directly testable.

Pitts' *evaluation logic* bears some resemblance to our monadic property language [23]. It is also parameterised on a monad, and permits properties to be stated which hold after a computation. Pitts writes $[x \Leftarrow e]P$ where we write $run\ e \gg\!= \lambda\, x\ \rightarrow\ P$. However, the two languages differ in essential ways. For example, Pitts can write $[x \Leftarrow e]P \wedge [x' \Leftarrow e']P'$, meaning that if we compute e, then P will hold, but if we compute e', then P' will hold. We cannot express this — indeed, we have no conjunction operator, but the reason is deep seated. To test this property, we would have to compute both e and e' *in some order*! But Pitts' property talks about the state after computing one or the other, *but not both*.

QuickCheck's main limitation as a testing tool is that it provides no information on the structural coverage of the program under test: there is no check, for example, that every part of the code is exercised. We leave this as the responsibility of an external coverage tool. Unfortunately, no such tool exists for Haskell! It is possible that Hat could be extended to play this rôle.

15. CONCLUSIONS

In this paper, we have shown how QuickCheck can be used for specification-based testing of imperative operations. The main contributions are:

- We have made a link between testing of imperative code and the concept of observational equivalence.
- We have shown how equations between imperative code fragments can be tested directly, by running each fragment in the same context. Representing contexts explicitly by data structures was a key step here.
- We have defined and given the semantics of a new kind of monadic properties, parameterised over any monad.
- We have shown that the QuickCheck property language, despite its limitations, is sufficiently powerful to represent many common specification formalisms (algebraic specifications, functional models, relational models, pre- and post-conditions).
- We have shown that each formalism so represented can be used directly for testing imperative code.

It will be exciting to formulate further formal systems using QuickCheck.

16. REFERENCES

[1] S. Antoy and R. Hamlet. Automatically checking an implementation against its formal specification. In *Irvine Software Symposium*, pages 29–48, March 1992.

[2] Gilles Bernot, Marie Claude Gaudel, and Bruno Marre. Software Testing based on Formal Specifications: a theory and a tool. *Software Engineering Journal*, 6(6):387–405, Nov 1991.

[3] F. Warren Burton. An efficient functional implementation of FIFO queues. *Information Processing Letters*, 14(5):205–206, July 1982.

[4] Koen Claessen and John Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. In *International Conference on Functional Programming*, pages 268–279. ACM, 2000.

[5] M. Donat. Automating Formal Specification Based Testing. In M. Bidoit and M. Dauchet, editors, *Proc. Conf. on Theory and Practice of Sw Development (TAPSOFT 97)*, volume 1214, pages 833–847, Lille, France, 1997. Springer-Verlag, Berlin.

[6] J. Duran and S. Ntafos. An evaluation of random testing. *Transactions on Software Engineering*, 10(4):438–444, July 1984.

[7] Stephen H. Edwards. A framework for practical, automated black-box testing of component-base software. *Software Testing, Verification and Reliability*, 11(2), June 2001.

[8] J. Gannon, R. Hamlet, and P. McMullin. Data abstraction implementation, specification, and testing. *Trans. Prog. Lang. and Systems*, (3):211–223, 1981.

[9] D. Hamlet. Random testing. In J. Marciniak, editor, *Encyclopedia of Software Engineering*, pages 970–978. Wiley, 1994.

[10] R. Hamlet and R. Taylor. Partition testing does not inspire confidence. *Transactions on Software Engineering*, 16(12):1402–1411, December 1990.

[11] I. J. Hayes, editor. *Specification Case Studies.* Prentice Hall International Series in Computer Science, 2nd edition, 1993.

[12] Dean Herington. HUnit 1.0 User's Guide, 2002. http://hunit.sourceforge.net/HUnit-1.0/Guide.html.

[13] P. Hudak. Modular Domain Specific Languages and Tools. In *Proceedings of Fifth International Conference on Software Reuse.* IEEE Computer Society, Jun 1999.

[14] Simon Peyton Jones. Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell. In Tony Hoare, Manfred Broy, and Ralf Steinbruggen, editors, *Engineering theories of software construction*, pages 47–96. IOS Press, 2001. Presented at the 2000 Marktoberdorf Summer School.

[15] J. Launchbury and S. Peyton Jones. State in Haskell. *Lisp and Symbolic Computation*, 8(4):293–341, December 1995.

[16] S. Liang, P. Hudak, and M. Jones. Monad transformers and modular interpreters. In *Conference Record of POPL'95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 333–343, San Francisco, California, 1995.

[17] Tom Moertel. Seven Lessons from the ICFP Programming Contest. http://www.kuro5hin.org/story/2001/7/31/0102/11014, 2001.

[18] Graeme E. Moss and Colin Runciman. Automated benchmarking of functional data structures. In *Practical Aspects of Declarative Languages*, pages 1–15, 1999.

[19] C. Mueller and B. Korel. Automated evaluation of cots components. In *First International Workshop on Automated Program Analysis, Testing and Verification*, Limerick, Ireland, 2000. In conjunction with ICSE 2000.

[20] Martin Odersky and Konstantin Läufer. Putting type annotations to work. In *Conference Record of POPL '96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, St. Petersberg Beach, Florida*, pages 54–67, New York, NY, 1996.

[21] Chris Okasaki. An Overview of Edison. In *Haskell Workshop*, pages 34–54, September 2000.

[22] S. L. Peyton Jones and P. Wadler. Imperative Functional programming. In *Proceedings 1993 Symposium Principles of Programming Languages*, Charleston, N.Carolina, 1993.

[23] Andrew M. Pitts. Evaluation logic. In G. Birtwistle, editor, *Proceedings of the IVth Higher Order Workshop*, pages 162–189. Springer-Verlag, 1990.

[24] Phil Stocks and David Carrington. A Framework for Specification Based Testing. *Transactions on Software Engineering*, 22(11), Nov 1996.

[25] Malcolm Wallace, Olaf Chitil, Thorsten Brehm, and Colin Runciman. Multiple-View Tracing for Haskell: a New Hat. In *Haskell Workshop.* ACM, September 2001.