QuickSpec: Guessing Formal Specifications using Testing

Koen Claessen[1], Nicholas Smallbone[1], and John Hughes[2]

[1] Chalmers University of Technology {koen,nicsma}@chalmers.se
[2] Chalmers and Quviq AB rjmh@chalmers.se

Abstract. We present QuickSpec, a tool that automatically generates algebraic specifications for sets of pure functions. The tool is based on testing, rather than static analysis or theorem proving. The main challenge QuickSpec faces is to keep the number of generated equations to a minimum while maintaining completeness. We demonstrate how QuickSpec can improve one's understanding of a program module by exploring the laws that are generated using two case studies: a heap library for Haskell and a fixed-point arithmetic library for Erlang.

1 Introduction

Understanding code is hard. But it is vital to understand what code does in order to determine its correctness.

One way to understand code better is to write down one's expectations of the code as formal specifications, which can be tested for compliance, by using a property-based testing tool. Our earlier work on the random testing tool QuickCheck [2] follows this direction. However, coming up with formal specifications is difficult, especially for untrained programmers. Moreover, it is easy to forget to specify certain properties.

In this paper, we aim to aid programmers with this problem. We propose an automatic method that, given a list of function names and their object code, uses testing to come up with a set of *algebraic equations* that seem to hold for those functions. Such a list can be useful in several ways. Firstly, it can serve as a basis for documentation of the code. Secondly, the programmer might gain new insights by discovering new laws about the code. Thirdly, some laws that one expects might be missing (or some laws might be more specific than expected), which points to to a possible miss in the design or implementation of the code.

Since we use testing, our method is potentially unsound, meaning some equations in the list might not hold; the quality of the generated equations is only as good as the quality of the used test data, and care has to be taken. Nonetheless, we still think our method is useful. However, our method is still complete in a precise sense; although there is a limit on the complexity of the expressions that occur in the equations, any syntactically valid equation that actually holds for the code can be derived from the set of equations that QuickSpec generates.

Our method has been implemented for the functional languages Haskell and Erlang in a tool called QuickSpec. At the moment, QuickSpec only works for

purely functional code, i.e. no side effects. (Adapting it to imperative and other side-effecting code is ongoing work.)

Examples Let us now show some examples of what QuickSpec can do, by running it on different subsets of the Haskell standard list functions. When we use QuickSpec, we have to specify the functions and variable names which may appear in equations, together with their types. For example, if we generate equations over the list operators

```
(++) :: [Elem] -> [Elem] -> [Elem]     -- list append
(:)  :: Elem -> [Elem] -> [Elem]       -- list cons
[]   :: [Elem]                         -- empty list
```

using variables `x, y, z :: Elem` and `xs, ys, zs :: [Elem]`, then QuickSpec outputs the following list of equations:

```
xs++[] == xs
[]++xs == xs
(xs++ys)++zs == xs++(ys++zs)
(x:xs)++ys == x:(xs++ys)
```

We automatically discover the associativity and unit laws for append (which require induction to prove). These equations happen to comprise a complete characterization of the `++` operator. If we add the list reverse function to the mix, we discover the additional familiar equations

```
reverse [] == []
reverse (reverse xs) == xs
reverse xs++reverse ys == reverse (ys++xs)
reverse (x:[]) == x:[]
```

Again, these laws completely characterize the `reverse` operator. Adding the `sort` function from the standard `List` library, we compute the equations

```
sort [] == []
sort (reverse xs) == sort xs
sort (sort xs) == sort xs
sort (ys++xs) == sort (xs++ys)
sort (x:[]) == x:[]
```

The third equation tells us that `sort` is idempotent, while the second and fourth strongly suggest (but do not imply) that the result of `sort` is independent of the order of its input.

Higher-order functions can be dealt with as well. Adding the function `map` together with a variable `f :: Elem -> Elem`, we obtain:

```
map f [] == []
map f (reverse xs) == reverse (map f xs)
map f xs++map f ys == map f (xs++ys)
f x:map f xs == map f (x:xs)
```

All of the above uses of QuickSpec only took a fraction of a second to run, and the equations shown here is the verbatim output of the tool.

Main related work The existing work that is most similar to ours is [4]. They describe a tool for discovering algebraic specifications from Java classes using testing, using a similar overall approach as ours (there are however important technical differences discussed in the related work section later in the paper). However, the main difference between our work and theirs is that we generate equations between *nested expressions* consisting of functions and variables whereas they generate equations between Java program fragments that are *sequences of method calls*. The main problem we faced when designing the algorithms behind QuickSpec was taming the explosion of equations generated by operators with structural properties, such as associativity and commutativity, equations that are not even expressible as equations between sequences of method calls (results of previous calls cannot be used as arguments to later ones).

Contributions We present a efficient method, based on testing, that automatically computes algebraic equations that seem to hold for a list of specified pure functions. Moreover, using two larger case studies, we show the usefulness of the method, and present concrete techniques of how to use the method effective in order to understand programs better.

2 How QuickSpec Works

The input taken by QuickSpec consists of three parts: (a) the compiled program, (b) a list of functions and variables, together with their types, and (c) test data generators for each of the types of which there exists at least one variable. As indicated in the introduction, the quality of the test data determines the quality of the generated equations. As such, it is important to provide good test data generators, that fit the program at hand. In our property-based random testing tool QuickCheck [2], we have a range of test data generators for standard types, and a library of functions for building custom generators.

The method The method used by QuickSpec follows four distinct steps: (1) We first generate a (finite) set of terms, called the *universe*, that includes any term that might occur on either side of an equation. (2) We use testing to partition the universe into *equivalence classes*; any two terms in the same equivalence class are considered equal after the testing phase. (3) We generate a list of *equations* from the equivalence classes. (4) We use *pruning* to filter out equations that follow from other equations by equational reasoning.

In the following, we discuss each of these steps in more detail, plus an optimization that greatly increases the efficiency of the method.

The universe First, we need to pin down what kind of equations QuickSpec should generate. To keep things simple and predictable, we only generate one finite set of terms, the *universe*, and any pair of terms from the universe can form an equation.

How the universe is generated is really up to the user, all we require is that the universe is subterm-closed. The most useful way of generating the universe

is letting the user specify a *term depth* (usually 3 or 4), and then simply produce all terms that are not too deep. The terms here consists of the function symbols and variables from the specified API.

The size of the universe is typically 1000 to 50.000 terms, depending on the application.

Equivalence classes The next step is to gather information about the terms in the universe. This is the only step in the algorithm that uses testing, and in fact makes use of the program. Here, we need to determine which terms seem to behave the same, and which terms seem to behave differently. In other words, we are computing an equivalence relation over the terms.

Concretely, in order to compute this equivalence relation, we use a refinement process. We represent the equivalence relation as a set of equivalence classes, partitions of the universe. We start by assuming that all terms are equal, and put all terms in the universe into one giant equivalence class. Then, we repeat the following process: We use the test data generators to generate test data for each of the variables occurring in the terms. We then refine each equivalence class into possibly smaller ones, by evaluating all the terms in a class and grouping together the ones that are still equal, splitting the terms that are different. The process is repeated until the equivalence relation seems "stable"; if no split has happened for the last 200 tests. Note that equivalence classes of size 1 are trivial, and can be discarded; these contain a term that is only equal to itself.

The typical number of non-trivial equivalence classes we get after this process lies between 500 and 5.000, again depending on the size of the original universe and the application.

Once these equivalence classes are generated, the testing phase is over, and the equivalence relation is trusted to be correct for the remainder of the method.

Equations From the equivalence classes, we can generate a list of equations between terms. We do this by picking one representative term r from the class, and producing the equation $t = r$ for all other terms from the class. So, an equivalence class of size k produces $k - 1$ equations. The equations seem to look nicest if r is the *simplest* element of the equivalence class (according to some simplicity measure based on for example depth and/or size), but what r is chosen has no effect on the completeness of the algorithm.

The typical number of equations that are generated in this step lies between 5000 and 50.000. A list of even 5.000 equations is however not something that we want to present to the user; the number of equations should be in the tens, not hundreds or thousands.

There is a great potential for improvement here. Among the equations generated for the boolean operator `&&`, the constant `false`, and the variables `x` and `y` we might find for example:

```
1. x&&false == false  3. y&&false == false  5. false&&false == false
2. false&&x == false  4. false&&y == false  6. x&&y==y&&x
```

It is obvious that law 5 is an instance of laws 1-4, and that laws 3 and 4 are just renamings of laws 1 and 2. Moreover, law 6 and 1 together imply all the

other laws. So, the above laws could be replaced by just `x&&false == false` and `x&&y==y&&x`. This is the objective of the pruning step.

Pruning Pruning filters out unnecessary laws. Which laws are kept and which are discarded by our current implementation of QuickSpec is basically an arbitrary choice. What we would like to argue is that it must be an arbitrary choice, but that it should be governed by the following four principles.

(1) *Completeness:* we can only remove a law if it can be derived from the remaining laws. We cannot force the method to remove all such laws, firstly because there is no unique minimal set, and secondly because derivability is not decidable. (2) *Conciseness:* we should however remove all obvious redundancy, for a suitably chosen definition of redundant. (3) *Implementability:* the method should be implementable and reasonably efficient. (4) *Predictability:* the user should be able to draw conclusions from the absence or presence of a particular law, which means that no ad-hoc decisions should be made in the algorithm.

That the choice of what is redundant or not is not obvious became clear to us after long discussions between the authors of the paper on particular example sets of equations. Also, what "derivable" means is not clear either: Do we allow simple equational reasoning, something weaker (because of decidability issues), or something stronger (by adding induction principles)?

Eventually, we settled on the following. First, all equations are ordered according to a simplicity measure. The simplicity measure can again be specified by the user; the default one we provide is based on term size. Then, we go through the list of equations in order, removing any equation that is "derivable" from the previous ones.

For "derivable", we decided to define a decidable and predictable *approximation* of logical implication for equations. The approximation uses a *congruence closure* data-structure, a generalization of a union/find data-structure that maintains a congruence relation[3] over a finite subterm-closed set of terms. Congruence closure is one of the key ingredients in modern SMT-solvers, and we simply reimplemented an efficient modern congruence closure algorithm [8].

Congruence closure enjoys the following property: suppose we have a set of equations E, and for each equation $s = t$ we record in the congruence closure data-structure $\equiv$ the fact $s \equiv t$. Then for any terms a and b, $a \equiv b$ will be true exactly if $a = b$ can be proved from E using *only* the rules of reflexivity, symmetry, transitivity and congruence of $=$.

This almost gives us a way of checking whether $a = b$ follows from E. However, we want to know whether $a = b$ can be proved at all from E, not whether it can be proved using some restricted set of rules. There's one more rule we could use in a proof, that's not covered by congruence closure: any instance of a valid equation is valid. To approximate this rule, for each equation $s = t$ in E, we should record not just $s \equiv t$ but many *instances* $\sigma(s) \equiv \sigma(t)$ (where σ is a substitution).

[3] A congruence relation is an equivalence relation that is also a congruence: if $x \equiv y$ then $C[x] \equiv C[y]$ for all contexts C.

In more detail, our algorithm is as follows:

1. We maintain a congruence relation $\equiv$ over terms, which initially is the identity relation. The relation $\equiv$ is going to represent all knowledge implied by accepted equations so far, so that if $s \equiv t$ then $s = t$ is derivable from the accepted equations.
2. We order all equations according to the equation ordering $<$, simplest equations first.
3. We loop through all equations, starting at the simplest. For each equation $s = t$, we check if $s \equiv t$ according to the maintained congruence relation $\equiv$. If so, the equation $s = t$ is implied by previous equations, and we discard it.
4. If $s \not\equiv t$, we produce $s = t$ as an equation. We then update the congruence relation $\equiv$ to represent the fact that we have produced the equation $s = t$. We do this by picking a finite set of instances of $s = t$. That is, we choose a finite set Σ of substitutions; then, for each $\sigma \in \Sigma$, we add the fact $\sigma(s) \equiv \sigma(t)$ to the congruence closure data-structure $\equiv$.
5. Once all equations have been taken care of, we are done.

We didn't specify above which instances of each equation $s = t$ to generate. Our original choice was to generate all substitutions σ such that $\sigma(s)$ and $\sigma(t)$ were in the universe. This allowed the algorithm to find any proof that only uses terms from the universe.

Now, instead, we generate substitutions separately for s and t. In the case of s, we generate all substitutions σ such that $\sigma(s)$ is in the universe (where σ ranges over the variables of s) and record $\sigma(s) = \sigma(t)$, and similarly in the case of t. This is a relaxation of our earlier choice.

By doing this, we allow the algorithm to reason also about terms t that lie outside the universe, but only if that term t is equated by an equation to a term s that lies inside the universe. This modification does not noticeably influence performance, but allowing this was vital to prune away equations involving operators with structural properties, such as commutativity and associativity. For example, generating properties about the arithmetic operator `+`, only allowing reasoning within the universe, we end up with:

```
1. x+y = y+x
2. y+(x+z) = (z+y)+x
3. (x+y)+(x+z) = (z+y)+(x+x)
```

The third equation can be derived from the first two, but we need to use a term `x+(y+(x+z))` that lies outside of the universe. Adding the modification we just described to the algorithm, this last equation is also pruned away.

The depth optimisation QuickSpec includes one optimisation to reduce the number of terms generated. We will first motivate the optimisation and then explain it in more detail.

Suppose we have run QuickSpec on an API of boolean operators with a depth limit of 2, giving (among others) the law `x&&x==x`. But now, suppose we want to increase the depth limit on terms from 2 to 3. Using the algorithm

described above, we would first generate all terms of depth 3, including such ones as `x&&y` and `(x&&x)&&y`. But these two terms are obviously equivalent (since we know that `x&&x==x`), we won't get any more laws by generating both of them, and we ought to generate only `x&&y` and not `(x&&x)&&y`.

The observation we make is that, if two terms are equal (like `x&&x` and `x` above), we ought to pick one of them as the "canonical form" of that expression; we avoid generating any term that has a non-canonical form as a subterm. In this example, we don't generate `(x&&x)&&y`, because it has `x&&x` as a subterm. (We do still generate `x&&x` on its own, otherwise we wouldn't get the law `x&&x==x`.)

The depth optimisation applies this observation, and works quite straightforwardly. If we want to generate all terms up to depth 3, say, we first generate all terms up to depth 2 (recursively applying the same optimisation) and sort them into equivalence classes by testing. Then we only generate those terms for which all the direct subterms are the representative of their equivalence class.

We can justify why this optimisation is sound. If we choose not to generate a term `t` with canonical form `t'`, and there would have been an equation `t==u`, there will also be an equation `t'==u`.[4] Furthermore, we will record in the congruence closure data structure all of the equations necessary to prove `t==t'` (all of these terms in these equations have smaller depth than `t`) so our pruning algorithm would have been able to prove `t==u` anyway.)

This optimisation makes a very noticeable difference to the number of terms generated. For a large list signature, the number of terms goes down from 21266 to 7079. For booleans there is a much bigger difference, since so many terms are equal: without the depth optimisation we generate 7395 terms, and with it 449 terms. Time-wise, the method becomes an order of a magnitude faster.

3 Case Studies

In this section, we present two case studies using QuickSpec. Our goal is primarily to derive *understanding* of the code we test. In many cases, the specifications generated by QuickSpec are initially disappointing—but by *extending the signature with new operations* we are able to arrive at concise and perspicuous specifications. Arguably, selecting the right operations to specify is a key step in formulating a good specification, and one way to see QuickSpec is as a tool to support exploration of this design space.

3.1 Case Study #1: Leftist Heaps in Haskell

A *leftist heap* [9] is a data structure that implements a priority queue. A leftist heap provides the usual heap operations:

[4] This relies on `t'` not having greater depth than `t`, which requires the term ordering to always pick the representative of an equivalence class as a term with the smallest depth.

```
empty :: Heap                        findMin :: Heap -> Elem
isEmpty :: Heap -> Bool              deleteMin :: Heap -> Heap
insert :: Elem -> Heap -> Heap
```

When we tested this signature with the variables `h, h1, h2 :: Heap` and `x, y, z :: Elem`. then QuickSpec generated a rather incomplete specification. The specification describes the behaviour of `findMin` and `deleteMin` on empty and singleton heaps:[5]

```
findMin empty == undefined           deleteMin empty == undefined
findMin (insert x empty) == x        deleteMin (insert x empty) == empty,
```

It shows that the order of insertion into a heap is irrelevant:

```
insert y (insert x h) == insert x (insert y h),
```

Apart from that, it only contains the following equation:

```
isEmpty (insert x h1) == isEmpty (insert x h)
```

This last equation is quite revealing—obviously, we would expect both sides to be `False`, which explains why they are equal. But why doesn't QuickSpec simply print the equation `isEmpty (insert x h) == False`? The reason is that `False` is not in our signature! When we add it to the signature, then we do indeed obtain the simpler form instead of the original equation above.[6]

In general, when a term is found to be equal to a renaming of itself with different variables, then this is an indication that a constant should be added to the signature, and in fact QuickSpec prints a suggestion to do so.

Generalising a bit, since `isEmpty` returns a `Bool`, it's certainly sensible to give QuickSpec operations that manipulate booleans. We added the remaining boolean connectives `True`, `&&`, `||` and `not`, and one newly-expressible law appeared, `isEmpty empty == True`.

Merge Leftist heaps actually provide one more operation than those we encountered so far: merging two heaps.

```
merge :: Heap -> Heap -> Heap
```

If we run QuickSpec on the new signature, we get the fact that `merge` is commutative and associative and has `empty` as a unit element:

```
merge h1 h == merge h h1
merge h1 (merge h h2) == merge h (merge h1 h2)
merge h empty == h
```

We get nice laws about `merge`'s relationship with the other operators:

```
merge h (insert x h1) == insert x (merge h h1)
isEmpty h && isEmpty h1 == isEmpty (merge h h1)
```

[5] QuickSpec generates an exceptional value `undefined` at each type.

[6] For completeness, we will list all of the new laws that QuickSpec produces every time we change the signature.

We also get some curious laws about merging a heap with itself:

```
findMin (merge h h) == findMin h
merge h (deleteMin h) == deleteMin (merge h h)
```

These are *all* the equations that are printed. Note that there are no redundant laws here. As mentioned earlier, our testing method guarantees that this set of laws is *complete*, in the sense that any valid equation over our signature, which is not excluded by the depth limit, follows from these laws.

With Lists We can get useful laws about heaps by relating them to a more common data structure, *lists*. First, we need to extend the signature with operations that convert between heaps and lists:

```
fromList :: [Elem] -> Heap
toList :: Heap -> [Elem]
```

`fromList` turns a list into a heap by folding over it with the `insert` function; `toList` does the reverse, deconstructing a heap using `findMin` and `deleteMin`. We should also add a few list operations mentioned earlier: `++`, `tail`, `:`, `[]`, and `sort`; and variables `xs, ys, zs :: [Elem]`. Now, QuickSpec discovers many new laws. The most striking one is

```
toList (fromList xs) == sort xs.
```

This is the definition of heapsort! The other laws indicate that our definitions of `toList` and `fromList` are sensible:

```
sort (toList h) == toList h
fromList (toList h) == h
fromList (sort xs) == fromList xs
fromList (ys++xs) == fromList (xs++ys)
```

The first law says that `toList` produces a sorted list, and the second that `fromList . toList` is the identity (up to `==` on heaps, which actually applies `toList` to each operand and compares them!). The other two laws suggest that the order of `fromList`'s input doesn't matter.

We get a definition by pattern-matching of `fromList`:

```
fromList [] == empty
insert x (fromList xs) == fromList (x:xs)
merge (fromList xs) (fromList ys) == fromList (xs++ys)
```

We also get a family of laws relating heap operations to list operations:

```
toList empty == []
head (toList h) == findMin h
toList (deleteMin h) == tail (toList h)
```

We can think of `toList h` as an abstract model of `h`—all we need to know about a heap is the sorted list of elements, in order to predict the result of any operation

on that heap. The heap itself is just a clever representation of that sorted list of elements.

The three laws above define `empty`, `findMin` and `deleteMin` by how they act on the sorted list of elements—the model of the heap. For example, the third law says that applying `deleteMin` to a heap corresponds to taking the `tail` in the abstract model (a sorted list). Since `tail` is obviously the correct way to remove the minimum element from a sorted list, this equation says exactly that `deleteMin` is correct![7]

So these three equations are a complete specification of `empty`, `findMin` and `deleteMin`!

This section highlights the importance of choosing a rich set of operators when using QuickSpec. There are often useful laws about a library that mention functions from unrelated libraries; the more such functions we include, the more laws QuickSpec can find. In the end, we got a complete specification of heaps (and heapsort, as a bonus!) by including list functions in our testing.

It's not always obvious *which* functions to add to get better laws. In this case, there are several reasons for choosing lists: they're well-understood, there are operators that convert heaps to and from lists, and sorted lists form a model of priority queues.

Buggy Code What happens when the code under test has a bug? To find out, we introduced a fault into `toList`. The buggy version of `toList` doesn't produce a sorted list, but rather the elements of the heap in an arbitrary order.

We were hoping that some laws would fail, and that QuickSpec would produce *specific instances* of some of those laws instead. This happened: whereas before, we had many useful laws about `toList`, afterwards, we had only two:

```
toList empty == []
toList (insert x empty) == x:[]
```

Two things stand out about this set of laws: first, the law `sort (toList h) == toList h` does not appear, so we know that the buggy `toList` doesn't produce a sorted result. Second, we *only get equations about empty and singleton heaps*, not about heaps of arbitrary size. QuickSpec is unable to find *any* specification of `toList` on nontrivial heaps, which suggests that the buggy `toList` *has* no simple specification.

3.2 Case Study #2: Understanding a Fixed Point Arithmetic Library in Erlang

We used QuickSpec to try to understand a library for fixed point arithmetic, developed by a South African company, which we were previously unfamiliar with. The library exports 16 functions, which is rather overwhelming to analyze in one go, so we decided to generate equations for a number of different subsets

[7] This style of specification is not new and goes back to Hoare [5].

of the API instead. In this section, we give a detailed account of our experiments and developing understanding.

Before we could begin to use QuickSpec, we needed a QuickCheck generator for fixed point data. We chose to use one of the library functions to ensure a valid result, choosing one which seemed able to return arbitrary fixed point values:

```
fp() -> ?LET({N,D},{largeint(),nat()},from_minor_int(N,D)).
```

That is, we call `from_minor_int` with random arguments. We suspected that `D` is the precision of the result—a suspicion that proved to be correct.

Addition and Subtraction We began by testing the `add` operation, deriving commutativity and associativity laws as expected. Expecting laws involving zero, we defined

```
zero() -> from_int(0)
```

and added it to the signature, obtaining as our reward a unit law, `add(A,zero()) == A`.

The next step was to add subtraction to the signature. However, this led to several very similar laws being generated—for example,

```
add(B,add(A,C)) == add(A,add(B,C))
add(B,sub(A,C)) == add(A,sub(B,C))
sub(A,sub(B,C)) == add(A,sub(C,B))
sub(sub(A,B),C) == sub(A,add(B,C))
```

To relieve the problem, we added another derived operator to the signature instead:

```
negate(A) -> sub(zero(),A).
```

and observed that the earlier family of similar laws was no longer generated, replaced by a single one, `add(A,negate(B)) == sub(A,B)`. Thus by *adding* a new auxiliary function to the signature, `negate`, we were able to *reduce* the complexity of the specification considerably.

After this new equation was generated by QuickSpec, we tested it extensively using *QuickCheck*. Once confident that it held, we could safely *replace* `sub` in our signature by `add` and `negate`, without losing any other equations. Once we did this, we obtained a more useful set of new equations:

```
add(negate(A),add(A,A)) == A
add(negate(A),negate(B)) == negate(add(A,B))
negate(negate(A)) == A
negate(zero()) == zero()
```

These are all very plausible—what is striking is the *absence* of the following equation:

```
add(A,negate(A)) == zero()
```

When an expected equation like this is missing, it is easy to formulate it as a QuickCheck property and find a counterexample, in this case `{fp,1,0,0}`. We discovered by experiment that `negate({fp,1,0,0})` is actually the same value! This strongly suggests that this is an alternative representation of zero (`zero()` evaluates to `{fp,0,0,0}` instead).

$0 \neq 0$ It is reasonable that a fixed point arithmetic library should have different representations for zero of different precisions, but we had not anticipated this. Moreover, since we want to derive equations involving zero, the question arises of *which zero* we would like our equations to contain! Taking our cue from the missing equation, we introduced a new operator `zero_like(A) -> sub(A,A)` and then derived not only `add(A,negate(A)) == zero_like(A)` but a variety of other interesting laws. These two equations suggest that the result of `zero_like` depends only on the number of decimals in its argument,

```
zero_like(from_int(I)) == zero()
zero_like(from_minor_int(J,M)) == zero_like(from_minor_int(I,M))
```

this equation suggests that the result has the *same* number of decimals as the argument,

```
zero_like(zero_like(A)) == zero_like(A)
```

while these two suggest that the number of decimals is preserved by arithmetic.

```
zero_like(add(A,A)) == zero_like(A)
zero_like(negate(A)) == zero_like(A)
```

It is not in general true that `add(A,zero_like(B)) == A` which is not so surprising—the precision of `B` affects the precision of the result. QuickSpec does find the more restricted property, `add(A,zero_like(A)) == A`.

Multiplication and Division When we added multiplication and division operators to the signature, then we followed a similar path, and were led to introduce `reciprocal` and `one_like` functions, for similar reasons to `negate` and `zero_like` above. One interesting equation we discovered was this one:

```
divide(one_like(A),reciprocal(A)) == reciprocal(reciprocal(A))
```

The equation is clearly true, but why is the right hand side `reciprocal(reciprocal(A))`, instead of just `A`? The reason is that the left hand side raises an exception if `A` is zero, and so the right hand side must do so also—which `reciprocal(reciprocal(A))` does.

We obtain many equations that express things about the precision of results, such as

```
multiply(B,zero_like(A)) == zero_like(multiply(A,B))
multiply(from_minor_int(I,N),from_minor_int(J,M)) ==
          multiply(from_minor_int(I,M),from_minor_int(J,N))
```

where the former expresses the fact that the precision of the zero produced depends both on `A` and `B`, and the latter expresses

$$i \times 10^{-m} \times j \times 10^{-n} = i \times 10^{-n} \times j \times 10^{-m}$$

That is, it is in a sense the commutativity of multiplication in disguise.

One equation we expected, but did *not* see, was the distributivity of multiplication over addition. Alerted by its absence, we formulated a corresponding QuickCheck property,

```
prop_multiply_distributes_over_add() ->
    ?FORALL({A,B,C},{fp(),fp(),fp()},
        multiply(A,add(B,C)) == add(multiply(A,B),multiply(A,C))).
```

and used it to find a counterexample:

```
A = {fp,1,0,4}, B = {fp,1,0,2}, C = {fp,1,1,4}
```

We used the library's `format` function to convert these to strings, and found thus that $A = 0.4, B = 0.2, C = 1.4$. Working through the example, we found that multiplying `A` and `B` returns a representation of 0.1, and so we were alerted to the fact that `multiply` rounds its result to the precision of its arguments.

Understanding Precision At this point, we decided that we needed to understand how the precision of results was determined, so we defined a function `precision` to extract the first component of an `{fp,...}` structure, where we suspected the precision was stored. We introduced a `max` function on naturals, guessing that it might be relevant, and (after observing the term `precision(zero())` in generated equations) the constant natural zero. QuickSpec then generated equations that tell us rather precisely how the precision is determined, including the following:

```
max(precision(A),precision(B)) == precision(add(A,B))
precision(divide(zero(),A)) == precision(one_like(A))
precision(from_int(I)) == 0
precision(from_minor_int(I,M)) == M
precision(multiply(A,B)) == precision(add(A,B))
precision(reciprocal(A)) == precision(one_like(A))
```

The first equation tells us the addition uses the precision of whichever argument has the most precision, and the fifth equation tells us that multiplication does the same. The second and third equations confirm that we have understood the representation of precision correctly. The second and sixth equations reveal that our definition of `one_like(A)` raises an exception when `A` is zero—this is why we do not see `precision(one_like(A)) == precision(A)`.

The second equation is more specific than we might expect, and in fact it is true that

```
precision(divide(A,B)) == max(precision(A),precision(one_like(B)))
```

but the right hand side exceeds our depth limit, so QuickSpec cannot discover it.

Adjusting Precision The library contained an operation whose meaning we could not really guess from its name, `adjust`. Adding `adjust` to the signature generated a set of equations including the following:

```
adjust(A,precision(A)) == A
precision(adjust(A,M)) == M
zero_like(adjust(A,M)) == adjust(zero(),M)
adjust(zero_like(A),M) == adjust(zero(),M)
```

These equations make it fairly clear that `adjust` sets the precision of its argument. We also generated an equation relating double to single adjustment:

```
adjust(adjust(A,M),0) == adjust(A,0)
```

Summing Up Overall, we found QuickSpec to be a very useful aid in developing an understanding of the fixed point library. Of course, we could simply have formulated the expected equations as QuickCheck properties, and tested them without the aid of QuickSpec. However, this would have taken very much longer, and because the work is fairly tedious, there is a risk that we might have forgotten to include some important properties. QuickSpec automates the tedious part, and allowed us to spot missing equations quickly.

Of course, QuickSpec also generates *unexpected* equations, and these would be much harder to find using QuickCheck. In particular, when investigating functions such as `adjust`, where we initially had little idea of what they were intended to do, then it would have been very difficult to formulate candidate QuickCheck properties in advance.

We occasionally encountered false equations resulting from the unsoundness of the method. In some cases these showed us that we needed to improve the distribution of our test data, in others (such as the difference between rounding in two stages and one stage) then the counterexamples are simply hard to find. QuickSpec runs relatively few tests of each equation (a few hundred), and so, once the most interesting equations have been selected, then it is valuable to QuickCheck them many more times.

4 Related Work

As mentioned earlier, the existing work that is most similar to ours is [4]; a tool for discovering algebraic specifications from Java classes. They generate terms and evaluate them, dynamically identify terms which are equal, then generate equations and filter away redundant ones. There are differences in the kind of equations that can be generated, which have been discussed earlier. The most important difference with our method is the fact that they initially generate only *ground* terms when searching for equations, then later generalise the ground equations by introducing variables, and test the equations *using the ground terms as test data*. To get good test data, then, they need to generate a large set of terms to work with, which heavily effects the efficiency of the subsequent generalization and pruning phases. In contrast, in our system, the number of terms does not affect the quality of the test data. So we get away with generating fewer terms—the cost of generating varying test data is only paid during testing, i.e. during the generation of the equivalence relation, and not in the term generation or pruning phase. Furthermore, we don't need a generalisation phase because our terms contain variables from the start.

There are other differences as well. They use a heuristic term-rewriting method for pruning equations; we use a predictable congruence closure algorithm. They observe—as we do—that conditional equations would be useful,

but neither tool generates them. Our tool appears to be faster (our examples take seconds to run, while comparable examples in their setting take hours). It is unfortunately rather difficult to make a fair comparison between the efficacy and performance of the two approaches, because their tool and examples are not available for download.

Daikon is a tool for inferring likely invariants in C, C++, Java or Perl programs [3]. Daikon observes program variables at selected program points during testing, and applies machine learning techniques to discover relationships between them. For example, Daikon can discover linear relationships between integer variables, such as array indices. Agitar's commercial tool based on Daikon generates test cases for the code under analysis automatically [1]. However, Daikon will not discover, for example, that `reverse(reverse(Xs)) == Xs`, unless such a double application of `reverse` appears in the program under analysis. Whereas Daikon discovers invariants that hold at existing program points, QuickSpec discovers equations between arbitrary terms constructed using an API. This is analogous to the difference between *assertions* placed in program code, and the kind of *properties* which QuickCheck tests, that also invoke the API under test in interesting ways. While Daikon's approach is ideal for imperative code, especially code which loops over arrays, QuickSpec is perhaps more appropriate for analysing pure functions.

Inductive logic programming (ILP) [7] aims to infer logic programs from examples—specific instances—of their behaviour. The user provides both a collection of true statements and a collection of false statements, and the ILP tool finds a program consistent with those statements. Our approach only uses *false* statements as input (inequality is established by testing), and is optimized for deriving equalities.

In the area of *Automated Theorem Discovery* (ATD), the aim is to emulate the human theorem discovery process. The idea can be applied to many different fields, such as mathematics, physics, but also formal verification. An example of an ATD system for mathematicians is MathSaid [6]. The system starts by generating a finite set of *hypotheses*, according to some syntactical rules that capture typical mathematical thinking, for example: if we know $A \Rightarrow B$, we should also check if $B \Rightarrow A$, and if not, under what conditions this holds. Theorem proving techniques are used to select theorems and patch non-theorems. Since this leads to many theorems, a filtering phase decides if theorems are interesting or not, according to a number of different predefined "tests". One such test is the simplicity test, which compares theorems for simplicity based on their proofs, and only keeps the simplest theorems. The aim of their filtering is quite different from ours (they want to filter out theorems that mathematicians would have considered trivial), but the motivation is the same; there are too many theorems to consider.

5 Conclusions and Future Work

We have presented a new tool, QuickSpec, which can automatically generate algebraic specifications for functional programs. Although simple, it is remarkably powerful. It can be used to aid program understanding, or to generate a QuickCheck test suite to detect changes in specification as the code under test evolves. We are hopeful that it will enable more users to overcome the barrier that formulating properties can present, and discover the benefits of QuickCheck-style specification and testing.

For future work, we plan to generate conditional equations. In some sense, these can be encoded in what we already have by specifying new custom types with appropriate operators. For example, if we want `x<=y` to occur as a precondition, we might introduce a type `AscPair` of "pairs with ascending elements", and add the functions `smaller,larger :: AscPair -> Int` and the variable `p :: AscPair` to the API. A conditional equation we could then generate is:

```
isSorted (smaller p : larger p : xs) == isSorted (larger p : xs)
```

(Instead of the perhaps more readable `x<=y ==> isSorted (x:y:xs) == isSorted (y:xs)`.) But we are still investigating the limitations and applicability of this approach.

Another class of equations we are looking at are equations between program fragments that can have side effects.

References

1. Boshernitsan, M., Doong, R., Savoia, A.: From Daikon to Agitator: lessons and challenges in building a commercial tool for developer testing. In: ISSTA '06: Proceedings of the 2006 international symposium on Software testing and analysis. pp. 169–180. ACM, New York, NY, USA (2006)
2. Claessen, K., Hughes, J.: QuickCheck: a lightweight tool for random testing of Haskell programs. In: ICFP '00: Proceedings of the fifth ACM SIGPLAN international conference on Functional programming. pp. 268–279. ACM, New York, NY, USA (2000)
3. Ernst, M.D., Perkins, J.H., Guo, P.J., McCamant, S., Pacheco, C., Tschantz, M.S., Xiao, C.: The Daikon system for dynamic detection of likely invariants. Sci. Comput. Program. 69(1-3), 35–45 (2007)
4. Henkel, J., Reichenbach, C., Diwan, A.: Discovering documentation for Java container classes. IEEE Trans. Software Eng. 33(8), 526–543 (2007)
5. Hoare, C.A.R.: Proof of correctness of data representations. Acta Inf. 1, 271–281 (1972)
6. McCasland, R.L., Bundy, A.: Mathsaid: a mathematical theorem discovery tool. In: Proceedings of the Eighth International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC'06) (2006)
7. Muggleton, S., de Raedt, L.: Inductive logic programming: Theory and methods. Journal of Logic Programming 19, 629–679 (1994)
8. Nieuwenhuis, R., Oliveras, A.: Proof-producing congruence closure. In: RTA '05. pp. 453–468. Springer LNCS, Nara, Japan (2005)
9. Okasaki, C.: Purely Functional Data Structures. Cambridge University Press (1998)