QuickSpec: Formal Specifications for Free!

Generating equational specifications from functional programs

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 modules consisting of pure functions. The tool is based on testing, rather than static analysis or theorem proving. The main application of QuickSpec is improving one's understanding of a module by exploring the laws that are generated. We demonstrate this with two case studies: a heap library for Haskell and a fixed-point arithmetic library for Erlang.

1 Introduction

Give us a module, any module, written in Haskell or Erlang, with a purely functional API. Tell us the names and types of the exported functions. In just a few seconds, we will give you in return an *algebraic specification* of your code, in the form of equations that the functions in your API satisfy. What's more, we will do so with no static analysis—you need not even give us the source code.

The purpose of this paper is to explain the trick we use, and to show how surprisingly well it works. Even though our method is based on testing, which inherently means that it is *unsound* (some equations that are generated might not actually hold in general), we are able to retain *completeness* in a certain precise sense, which we will come back to later.

Let us begin with some examples to show what QuickSpec, our tool, can do. We shall derive equations from well-known modules taken from the Haskell and Erlang standard libraries.

Lists As a first example, we consider some of the standard list processing functions in Haskell. When we use QuickSpec, we 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]
(:)  :: Elem -> [Elem] -> [Elem]
[]   :: [Elem]
```

using variables `x, y, z :: Elem` and `xs, ys, zs :: [Elem]`, then QuickSpec outputs the following 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), and indeed these equations 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 derive 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.

If we add a `merge` function for ordered lists, then we obtain an equation relating `merge` and `sort`:

```
merge (sort xs) (sort ys) == sort (xs++ys)
```

We also obtain other equations about `merge`, such as the somewhat surprising `merge (xs++ys) xs == merge xs xs++ys`. Note that this holds even for *unordered* `xs` and `ys`, but is an artefact of the precise definition of `merge`.

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)
```

Arrays QuickSpec has two implementations, one for Haskell and one for Erlang. Let us use our tool to investigate subsets of the API of the standard Erlang library for purely functional, flexible arrays, indexed from zero [2]. Using the following signature[3],

[3] Although Erlang does not have a static type system, the notation used here is commonly used to specify "types" of Erlang functions in documentation

```
new() -> array()
get(index(),array()) -> elem()
set(index(),elem(),array()) -> array()
default_element() -> elem()
```

with variables `X, Y, Z :: elem()`, `I, J, K :: index()`, and `A, B, C :: array()`, we obtained these laws:

```
get(I,new()) == default_element()
get(I,set(I,X,A)) == X
get(I,set(J,default_element(),new())) == default_element()
get(J,set(I,X,new())) == get(I,set(J,X,new()))
set(I,X,set(I,Y,A)) == set(I,X,A)
set(J,X,set(I,X,A)) == set(I,X,set(J,X,A))
```

The `default_element()` is not part of the arrays library: we introduced it and added it to the signature after QuickSpec generated the equation `get(I,new()) == get(J,new())`. Since the result of reading an element from an empty array is constant, we might as well give it a name for use in other equations. When we do so, then the equation just above is replaced by the first one generated.

Some of the equations above are very natural: the second says that writing an element, then reading it, returns the value written; the fifth says that writing to the same index twice is equivalent to just writing the second value. The sixth says that writing the *same* value `X` to two indices can be done in either order—but why can't we swap *any* two writes, as in `set(J,Y,set(I,X,A)) =?= set(I,X,set(J,Y,A))`? The reason is that this equation holds only if `I /= J` (or if `X == Y`, of course)! It would be nice to generate *conditional equations* such as `I /= J ==> set(J,Y,set(I,X,A)) == set(I,X,set(J,Y,A))` but at present QuickSpec cannot do this.

The fourth equation, `get(J,set(I,X,new())) == get(I,set(J,X,new()))`, is a little surprising at first, but it does hold—either both sides are the default element, if `I` and `J` are different, or both sides are `X`, if they are the same.

Finally, the third equation is quite revealing about the implementation:

```
get(I,set(J,default_element(),new())) == default_element()
```

A new array contains the default element at every index; evidently, setting an index explicitly to the default element will not change this, so it is no surprise that the `get` returns this element. The surprise is that the second argument of `get` appears in this complex form. Why is it `set(J,default_element(),new())`, rather than simply `new()`, when both arrays have precisely the same elements? The answer is that *these two arrays have different representations*, even though their elements are the same[4]. That the equation appears in this form tells us, indirectly, that `set(J,default_element(),new()) /= new()` because

[4] Only the Erlang version of QuickSpec behaves like this; the Haskell version uses the ordinary `(==)` operator for equality testing.

if they were equal, then QuickSpec would have simplified the equation. In fact, there is another operation in the API, `reset(I,A)`, which is equivalent to setting index `I` to the default element, and we discover in the same way that `reset(J,new()) /= new()`. `set` and `reset` could have been defined to leave an array unchanged if the element already has the right value—and this could have been a useful optimization, since returning a different representation forces `set` and `unset` to copy part of the array data-structure. Thus this *missing equation* reveals a potentially questionable design decision in the library itself. This is exactly the kind of insight we would like QuickSpec to provide!

The arrays library includes an operation to *fix* the size of an array, after which it can no longer be extended just by referring to a larger index. When we add `fix` to the signature, we discover

```
fix(fix(A)) == fix(A)
get(I,fix(new())) == undefined()
set(I,X,fix(new())) == undefined()
```

Fixing a fixed array does not change it, and if we fix a new array (with a size of zero), then any attempt to get or set an element raises an exception[5].

2 How QuickSpec Works

We hope the examples above have convinced the reader that useful insights can be obtained by studying the equations that QuickSpec generates. Now we shall explain the method we use to derive them. Surprisingly, no sophisticated theorem proving methods are used—we discover our equations simply by *testing* the code under analysis.

2.1 Equivalence classes of terms

A fundamental choice we made in our approach is that we reformulate our goal from generating equations such as `xs++[] == xs` and `[]++xs == xs` into generating *equivalence classes* such as `{xs, xs++[], []++xs}`. There is an immediate advantage to this choice; we abstract away over arbitrary choices in the generated equations. For example, logically, it does not really matter if, instead of the above two equations, we had generated `xs++[] == []++xs` and `xs++[] == xs` because these two equations contain the same information.

Once we have the equivalence classes, it is a trivial process to generate equations. For each equivalence class, we pick one representative term r. Then, for each other member t of that equivalence class, we generate the equation $t = r$. So, an equivalence class with n terms generates $n - 1$ equations. Exactly which representative r we choose does not really matter, but since the term r might appear repeatedly in multiple equations, it is more appealing to the eye if r is

[5] We consider all terms which raise an exception to be equal—and `undefined()` always does so.

the *smallest* term in the equivalence class, according to some total ordering $<$ on terms.

The next question we have to answer is, what should the domain of the computed equivalence relation be? The choice we made was that we simply enumerate all terms up to a given depth, where variables and constants count as having depth 1, and every function application adds a depth of 1. For most applications, choosing depth 3 or 4 generates a set of terms of manageable size to work with.

As a running example, let us consider a tiny list signature consisting of the empty list `[]`, the append operator `++`, and two variables `xs` and `ys`, and a term depth of 2. The following initial set of terms is produced:

```
{xs, ys, [], xs++xs, ys++ys, xs++ys, ys++xs,
  xs++[], ys++[], []++xs, []++ys, []++[]}
```

The equation generator's first job is to produce equivalence classes over these terms.

Rather than generating equations that we know for certain do hold, which would require static analysis or theorem proving, QuickSpec removes equations that for certain do not hold, which only requires testing. The result is a set of equations that *seem to hold*; the validity of the equations is *approximated* by means of random testing.

Thus, QuickSpec applies a simple method using testing to partition the set of terms up into equivalence classes. The method starts with only one big equivalence class of which all terms are a member. Then, it enters a repeating process, where at each step the equivalence relation is *refined*; each equivalence class may be split into several new equivalence classes. Such a step is implemented by generating a *random test case*, in which all variables are assigned a random value of the right type, and subsequently evaluating all terms using this assignment. We refine an existing equivalence class into several ones if it turns out that some of its terms evaluate to different values than others. We keep repeating this refinement process until the calculated equivalence relation appears to be "stable", when no more changes have occurred for an (arbitrarily chosen) number of iterations.

For example, computing the set of equivalence classes for the tiny list API above, we start with one big equivalence class, which consists of all terms. After generating one random test, e.g. `xs=[]`, `ys=[1]`, this class is immediately split up into several new ones:

```
{xs, [], xs++xs, xs++[], []++xs, []++[]}  ([])
{ys, xs++ys, ys++xs, ys++[], []++ys}      ([1])
{ys++ys}                                  ([1,1])
```

We show the values that each new equivalence class evaluates to in parentheses. One more random test case might choose `xs=[1]`, `ys=[2]`, and the result is:

```
{xs, xs++[], []++xs}                      ([1])
{[], []++[]}                              ([])
{xs++xs}                                  ([1,1])
```

```
{ys, ys++[], []++ys}                    ([2])
{xs++ys}                                ([1,2])
{ys++xs}                                ([2,1])
{ys++ys}                                ([2,2])
```

Any further attempt to refine the above equivalence classes fails, and thus the above is the final result. Generating equations from this equivalence relation produces the following (note that equivalence classes with only one element do not generate any equations):

```
xs++[] == xs      []++xs == xs      []++[] == []
ys++[] == ys      []++ys == ys
```

We notice that the expected equations are there (the first two). We also see that we have laws that are instances of other laws, and that we would rather not like to see in the list of equations.

It turns out that generating the equivalence classes was actually the easy part; the hardest part of generating equations is *pruning away* the unwanted equations! This is what the next subsection describes how to do.

Before we look at how to prune the equations, let us take a step back to see what we have done. We have computed an *over-approximation* of the actual equivalence relation on terms. This means that our equivalence relation might equate *more* terms than it should. This is because the random testing may not have found a particular test case that would have separated an equivalence class. As a result, our method is *unsound*; sometimes equations are generated that do not hold. However, the method also discovers all equations between terms of a particular depth, and never separates two equal terms into different equivalence classes. In this sense, our method should be considered *complete* w.r.t. a chosen term depth!

2.2 Pruning

The problem at this point is that we generate too many equations, and thus we need a *pruning principle* that tells us which equations are safe to remove. It is far from obvious what this pruning principle should be, as illustrated by the following four guiding principles we used to come up with our pruning principle.

Lower bound We should not remove too many equations from the set. What does "too many" mean here? At least, the final set of equations should still logically imply all equations in the original set. Thus, a reasonable first choice might be to find "the least subset of equations that logically imply all other equations". However, such a least set is not uniquely defined, and might not even be computable.

Upper bound We should not remove too few equations. It is clear that any equation that is directly implied by another equation (for example, because the former is an instance of the latter) should be removed. However, should any equation that is implied by a subset of the other equations be removed? It

happens quite often that a simple, appealing equation can be proven from a number of more general, larger equations, by means of a complex proof, possibly requiring induction. In that case, we might not want to remove the simpler equation.

Implementability Checking whether or not an equation is logically implied by a set of given equations is at best semi-decidable (depending on if and what kind of induction principles are to be used in the proof). We are aiming for a fully automatic method. Moreover, we want to create a practical tool to be used by programmers; the answer should be computable in a matter of seconds (perhaps minutes), not hours or days. So, whatever pruning principle we choose, it needs to be efficiently implementable.

Predictability The design space for the pruning principle seems to be huge with no obvious optimum, which suggests the use of heuristics. We want our method to be predictable, so that we are able to draw conclusions from the absence or presence of equations, and so that the user of the tool can influence and control which equations are generated in an intelligible way.

The final algorithm we ended up with only removes a law if it can be derived from *simpler* laws. This strategy makes it easy to avoid circular reasoning and lends itself to efficient implementation. Ordering equations according to simplicity is determined by a total order $<$ on equations, which can be specified as a parameter to the algorithm.

Our algorithm is based on 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[6] 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).

In more detail, our algorithm is as follows:

[6] 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.

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 algorithm uses the concept of the *universe*, which is the set of all terms generated in the testing phase. 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 all substitutions σ such that $\sigma(s)$ *or* $\sigma(t)$ are in the universe. 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.

3 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:

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

We decided to test an implementation of leftist heaps with QuickSpec, given the operations above and variables `h, h1, h2 :: Heap` and `x, y, z :: Elem`.

The result QuickSpec generated a rather incomplete specification. The specification describes the behaviour of `findMin` and `deleteMin` on empty and singleton heaps:

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

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

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

and otherwise only contains the following equation:

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

Our aim for the rest of this section is to coax a better set of laws out of QuickSpec[7]. We start with the last equation above. From it we can prove a related equation, `isEmpty (insert x h) == isEmpty (insert y h)`[8]. The two equations say that in the expression `isEmpty (insert x h)`, you can *replace* `x` and `h` by some other values `y` and `h1` without affecting the result. So that expression should have a *constant* value. That value is of course `False`, and if we just add `False` to our signature, then we get the real form of the previous law, `isEmpty (insert x h) == False`[9].

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. We might as well add that to our signature:

```
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:

---

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

[8] Both sides of this equation can be proved equal to `isEmpty (insert x (insert y h))`.

[9] In fact, QuickSpec prints a warning in this case advising the user to represent the value `isEmpty (insert x h)` by a constant.

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

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

```
(++) :: [Elem] -> [Elem] -> [Elem]
tail :: [Elem] -> [Elem]
(:) :: Elem -> [Elem] -> [Elem]
[] :: [Elem]
sort :: [Elem] -> [Elem]
```

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! We also get several laws that 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. 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`. What this means is that all we need to know about a heap is the sorted list of elements, and then we can predict the result of any operation on that heap. The heap itself is just a fancy 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[10]!

So these three equations are a complete specification of `empty`, `findMin` and `deleteMin`! We should extend them to a full specification of heaps. To do so, we add operators to insert an element into a sorted list, to merge two sorted lists, and to test if a sorted list is empty...

```
insertL :: Elem -> [Elem] -> [Elem]
mergeL :: [Elem] -> [Elem] -> [Elem]
null :: [Elem] -> Bool
```

...and our reward is three laws asserting that `insert`, `merge` and `isEmpty` are correct:

```
toList (insert x h) == insertL x (toList h)
mergeL (toList h) (toList h1) == toList (merge h h1)
null (toList h) == isEmpty h
```

We also get another law about `fromList` to go with our earlier collection: `fromList (mergeL xs ys) == fromList (xs++ys)`.

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:

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

```
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.

3.1 A trick

We finish with a "party trick": getting QuickSpec to discover how to implement `insert` and `deleteMin`. We hope to run QuickSpec and see it print equations of the form `insert x h = ?` and `deleteMin h = ?`.

The trick needs some preparation; if we just run QuickSpec straight away, we won't get either equation. There are two reasons, each of which explains the disappearance of one equation.

First, it's impossible to implement `deleteMin` using only the leftist heap API, so there's no equation for QuickSpec to print. To give QuickSpec a chance, we need to reveal the representation of leftist heaps; they're really binary trees. So we add the functions

```
leftBranch :: Heap -> Heap
rightBranch :: Heap -> Heap
```

to the signature. Of course, no implementation of leftist heaps would export these functions, this is only for the trick.

Secondly, QuickSpec won't bother to print out the definition of `insert`—it's easily derivable from the other laws, so QuickSpec considers it boring. Actually, in most ways, it *is* pretty boring; the one thing that makes it interesting is that it defines `insert`, but QuickSpec takes no notice of that.

Fortunately, we have a card up our sleeve: QuickSpec prints a list of *definitions*, equations that appear to define an operator in terms of other operators. The real purpose of this is to suggest redundant operators, but we will use it to see the definition of `insert` instead.

Everything in place, we run QuickSpec. And—hey presto!—out come the equations

```
insert x h = merge h (insert x empty)
deleteMin h = merge (leftBranch h) (rightBranch h)
```

That is, you can insert an element by merging with a unit heap that just contains that element, or delete the minimum element—which happens to be stored at the root of the tree—by merging the root's branches.

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

While working on QuickSpec, we were sent an Erlang library for fixed point arithmetic developed by a South African company. Since we were unfamiliar with the code, and did not immediately understand the API, we decided to experiment with QuickSpec as a program understanding tool. 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 of the API instead. In this section, we give a fairly detailed account of our experiments and developing understanding.

Before we could begin to use QuickSpec, we needed a QuickCheck generator for fixed point data. After a little experimentation, we settled on a generator using one of the API functions to ensure a valid result, choosing one which seemed able to generate an arbitrary result:

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

We suspected that `D` is the number of decimal places in 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. We then added `zero()` to our signature and derived 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 a simpler 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)`.

This equation was generated by QuickSpec, and we then tested it extensively using *QuickCheck*. Once confident that it holds, then we can 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 `recip` 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),recip(A)) == recip(recip(A))
```

The equation is clearly true, but why is the right hand side `recip(recip(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 `recip(recip(A))` does.

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

```
mult(B,zero_like(A)) == zero_like(mult(A,B))
mult(from_minor_int(I,N),from_minor_int(J,M)) ==
           mult(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_mult_distributes_over_add() ->
    ?FORALL({A,B,C},{fp(),fp(),fp(),
        mult(A,add(B,C)) == add(mult(A,B),mult(A,C))).
```

and used it to find a counterexample:

```
{{fp,1,0,4},{fp,1,0,2},{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 `mult` 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(mult(A,B)) == precision(add(A,B))
precision(recip(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.

It is worth noting that QuickSpec initially generated a set of equations including `precision(divide(A,B)) == precision(add(A,B))` for this signature. This is patently untrue, since the left hand side can raise an exception, and the right hand side cannot. When false equations appear, it indicates that the test data we are using is not sufficiently good to falsify them. Seeing this equation revealed to us that zero was not generated often enough by our test data generator, and we eliminated it by tweaking the generator to increase the probability of doing so.

Adjusting Precision The library contained an operation whose meaning we could not really guess from their names, `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)
```

We generalised this to `N =< M ==> adjust(adjust(A,M),N) == adjust(A,N)` which QuickSpec might well have generated if it could produce conditional equations. We tested the generalised equation with QuickCheck, and discovered it to be false. The counterexample QuickCheck found shows that the problem is caused by rounding: adjusting 0.1045 to three decimal places yields 0.105, and adjusting this to two decimals produces 0.11. Adjusting the original number to two decimals in one step produces 0.10, however, which is different. In fact, the original equation that QuickSpec found above is also false—but several hundred tests are usually required to find a counterexample. This shows the importance of testing the most interesting equations that QuickSpec finds more extensively—occasionally, it does report falsehoods.

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.

Notwithstanding the title of our paper, the specification we derived was not entirely "free". We needed to use our ingenuity to extend the signature with useful auxiliaries, such as `negate`, `precision`, `+` and `max`, to get the best out of QuickSpec. Moreover, as the size of the signature grew, then QuickSpec needed to construct several thousand terms, and run hundreds of thousands of tests—which began to take minutes, rather than seconds. To keep the running time within reason, we needed to select subsets of the full signature to investigate.

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.

5 Discussion

Pruning The "right" set of equations that has just the right amount of equations in it cannot really be formally defined. In discussions among the authors of the paper this became very clear; whether or not an equation is considered redundant as part of a larger set seems to be more a matter of taste than science.

Nevertheless, we decided to formally define what we mean by a redundant equation using our pruning principle. The design space of such pruning principles is rather large, and the choices are non-obvious. There are two main points in choosing a pruning principle: (1) It's not decidable in general whether an equation can be safely removed, so we have to use an approximation. There is therefore no single best pruning algorithm; we have to compromise between efficiency and power. Our pruning algorithm is powerful yet quite efficient. We restrict the set of terms that can appear in a proof, but we are able to find *any* proof that respects this restriction. (2) A strictly *minimal* set of laws can be hard to understand; we would prefer a bigger set of laws if it yields more insight. This means keeping certain redundant laws, but exactly which seems to be a matter of taste. We choose to prune away only laws that can be proved from simpler laws, which seems to keep a reasonable number of equations. However, our definition of "simpler" is ad-hoc; a more thoughtful definition would surely improve QuickSpec.

Summarizing, we realize that any choice of pruning principle is going to be arbitrary, but we want to argue that there is no non-arbitrary choice. We believe we have found an interesting point in the design space, which seems to be cheap and practical, although it's surely still possible to improve the pruning principle.

Equality The Erlang version of QuickSpec uses structural equality in the generated equations, which means that terms that may evaluate to different representations of the same abstract value are considered to be different, for example

causing some of the unexpected results in section 4. The Haskell version uses the `(==)` operator, defined in the appropriate `Eq` instance. However, this is unsafe unless `(==)` is a congruence relation with respect to the operations in the API under test! QuickSpec can be extended to *test* for these properties while classifying terms, although space issues do not allow us to go into this here.

6 Related Work

The existing work that is most similar to ours is [4]. They describe a tool for discovering algebraic specifications from Java classes, and use a similar overall approach: they generate terms and evaluate them, dynamically identify terms which are equal, then generate equations and filter away redundant ones.

The most important difference in the two approaches 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 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 test terms for operational equivalence, which is quite expensive; we use fast structural equivalence or a user-specified equality test. They use a heuristic term-rewriting method for pruning equations which will not handle structural properties well (we note that their case studies do not include commutative and associative operators, which we initially found to be extremely problematic); we use a predictable congruence closure algorithm. We are able to generate equations relating higher-order functions; working in Java, this was presumably not possible. They observe—as we do—that conditional equations would be useful, but neither tool generates them. They generate equations of several particular, fairly general, forms; working with equivalence classes, we have no restriction on the shape of equations, only on term size.

Our approach is simpler than theirs but seems to be more effective: our tool improves the range of equations and the pruning of equations, and appears to be faster (our examples take seconds to run, while comparable examples in their setting take hours). However, their task is more difficult in that they have to do all this in an imperative setting. 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 be-

tween 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. Second, theorem proving techniques are used to select theorems and patch non-theorems. Finally, 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 and the implementation are 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.

7 Conclusions and Future Work

We have presented a new tool, QuickSpec, which can automatically generate algebraic specifications for Haskell and Erlang 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. Moreover, it is great fun to use!

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. The difficulty here is to find an efficient way to select appropriate preconditions during term classification. We expect to need to restrict preconditions to be small terms; they must also be true sufficiently often to engender confidence that the equation they guard is really true. Another class of equations we are looking at are algebraic imperative specifications. One possible way of dealing with these is to model them using monads or continuations. The problem is that monadic combinators are polymorphic; so far, we can only deal with monomorphic signatures.

A minor extension is to be able to ask QuickSpec why an equation *isn't* printed. QuickSpec should respond with either a proof of the equation or a counterexample, either of which can be easily extracted from QuickSpec's data structures.

Acknowledgments The title and introduction of this paper were directly inspired by Phil Wadler's paper "Theorems For Free!" [10].

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. Carlsson, R., Gudmundsson, D.: The new array module. In: Däcker, B. (ed.) 13th International Erlang/OTP User Conference. Stockholm (2007), available from `http://www.erlang.se/euc/07/`
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: Proceedings of the 16th International Conference on Rewriting Techniques and Applications. pp. 453–468. Springer LNCS, Nara, Japan (2005)
9. Okasaki, C.: Purely Functional Data Structures. Cambridge University Press (1998)
10. Wadler, P.: Theorems for free! In: FPCA '89: Proceedings of the fourth international conference on Functional programming languages and computer architecture. pp. 347–359. ACM, New York, NY, USA (1989)