Template-based Theory Exploration: Discovering Properties of Functional Programs by Testing

Sólrún Halla Einarsdóttir
Nicholas Smallbone
Moa Johansson
slrn@chalmers.se
nicsma@chalmers.se
moa.johansson@chalmers.se
Chalmers University of Technology
Gothenburg, Sweden

ABSTRACT

We present RoughSpec, a template-based extension of the theory exploration tool QuickSpec. QuickSpec uses testing to automatically discover equational properties about functions in a Haskell program. These properties can help the user understand the program or be used as a source of possible lemmas in proofs of the program's correctness. In RoughSpec, the user supplies templates, which describe families of laws such as associativity and distributivity, and we only consider properties that match the templates. This restriction limits the search space and ensures that only relevant properties are discovered. In this way, we sacrifice broad search for more direction towards desirable property patterns, which makes theory exploration tractable and scalable. We also combine RoughSpec with QuickSpec, using QuickSpec to perform a complete search for smaller term sizes, while using templates for larger, more complex properties, in order to leverage the strengths of both systems.

CCS CONCEPTS

• **Software and its engineering → Functional languages**.

KEYWORDS

Theory exploration, QuickSpec, Functional programming, Algebraic properties, Program understanding, Property-based testing

ACM Reference Format:
Sólrún Halla Einarsdóttir, Nicholas Smallbone, and Moa Johansson. 2020. Template-based Theory Exploration: Discovering Properties of Functional Programs by Testing. In *IFL 2020: Proceedings of the 32nd Symposium on Implementation and Application of Functional Languages (IFL '20), September 2–4, 2020, Canterbury, United Kingdom.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3462172.3462192

1 INTRODUCTION

One strength of functional programming is that programs are easy to reason about. Pure functions often obey simple formal specifications which, as long as the programmer writes them down, are a great help in programming. A formal specification can be proved correct, automatically tested with a tool such as QuickCheck [4] or SmallCheck [17], or simply read in order to understand a codebase.

Many functional programmers already specify their code, by writing e.g. QuickCheck properties, but many do not. Can those who do *not* specify their code also reap the benefits of formal specification? The answer is *yes*: given a piece of code, we can automatically infer properties about it.

A tool that infers properties from code is called a *theory exploration system.* Two theory exploration systems for Haskell are QuickSpec [18] and Speculate [1]. These tools take as input a collection of Haskell functions and, through testing, discover formal properties which can be expressed using those functions. For example, given the list functions `++`, `reverse`, and `map`, QuickSpec discovers a total of five laws, all of them well-known and useful:

```
reverse (reverse xs) = xs
map f (reverse xs) = reverse (map f xs)
(xs ++ ys) ++ zs = xs ++ (ys ++ zs)
reverse xs ++ reverse ys = reverse (ys ++ xs)
map f xs ++ map f ys = map f (xs ++ ys)
```

Both tools work in a similar way. They take as input a *signature*, which describes the set of functions we would like to explore. Very roughly, they potentially consider *all* possible properties, up to some size limit, which can be built from the given functions (and some variables). They (1) build a set of terms from the given functions, (2) use automatic testing to check which terms appear to be equal and build equations from the equal terms,[1] resulting in a set of equational properties which are likely to be true, (3) remove any redundant properties (a property is redundant if it can be derived from other discovered properties), and (4) report all the non-redundant discovered properties. Because they explore all possible properties, the generated specification is *complete* (up to the size limit).

This approach works well on small sets of functions. Completeness means that we get an expressive specification, and discarding redundant properties keeps the specification short. When given

[1] Speculate also discovers non-equational properties.

only a few functions, QuickSpec and Speculate typically produce clear, crisp and useful specifications, like the one above. We have found that studying the output of QuickSpec is a great help in understanding an unfamiliar API.

Unfortunately, this approach breaks down when exploring large APIs: a *complete* theory exploration system simply finds too many laws. In a benchmark running QuickSpec on about 30 list functions [18], over 500 laws were found! The QuickSpec user is unlikely to bother reading all 500. Many are unenlightening, such as this one:

```
map (f x) (take (succ 0) xs) = zipWith f (scanl g x []) xs
```

This law is found, not because it was interesting, but because it was true and because QuickSpec did not consider it to be redundant. When we explore large APIs, we often get huge numbers of uninteresting laws. Furthermore, the search space is huge so the tools often take a while to run: exploring the 30 list functions took about two hours. These problems arise precisely because QuickSpec and Speculate are complete—unless a law is redundant, it will get discovered and printed out, interesting or not.

1.1 RoughSpec

We have developed a new theory exploration system, RoughSpec. Like QuickSpec and Speculate, it takes as input a set of Haskell functions (which we call the *signature*), and uses testing to find properties that seem to hold. The difference is that RoughSpec is *incomplete*: it does not try to find all true properties.

Instead, the user gives a set of *templates*, expressions which describe a family of laws such as associativity or distributivity. RoughSpec searches only for instances of these templates. In this way, the user can specify what kind of properties they would find interesting, and RoughSpec searches only for these properties.

A template is a Haskell equation containing functions, variables and *metavariables*. For example, here is a template which represents commutativity (note that in our syntax, variables are written in uppercase, and a metavariable is written as a variable with a leading question mark):

```
?F X Y = ?F Y X
```

When a template contains metavariables, RoughSpec *instantiates* them with functions drawn from the signature, tests the resulting equations, and reports any that appear to hold. In this case, RoughSpec will search for functions `?F` such that `?F X Y = ?F Y X` for all `X` and `Y` — that is, for commutative functions.

Here are some more examples of templates. They describe: (1) associativity; (2) an invertible function; (3) distributivity; (4) and (5) a function having an identity element; and (6) a list homomorphism (at least on non-empty lists):

```
(1) ?F (?F X Y) Z = ?F X (?F Y Z)
(2) ?F (?G X) = X
(3) ?F (?G X) (?G Y) = ?G (?F X Y)
(4) ?F X ?E = X
(5) ?F ?E X = X
(6) ?F (X ++ Y) = ?G (?F X) (?F Y)
```

In (4) and (5), `?E` is a metavariable, and will be replaced by *constants* drawn from the signature, while `X` is a universally-quantified variable. Note too that apart from metavariables and variables, templates may also mention specific functions, such as `++` in (6).

When we run RoughSpec on a signature of five list functions `++`, `reverse`, `map`, `sort` and `nub`, using the templates (1)–(3) above as well as commutativity, we get the following output:

```
Searching for commutativity properties...
  1. sort (xs ++ ys) = sort (ys ++ xs)
Searching for associativity properties...
  2. (xs ++ ys) ++ zs = xs ++ (ys ++ zs)
  3. sort (sort (xs ++ ys) ++ zs) =
     sort (xs ++ sort (ys ++ zs))
  4. nub (nub (xs ++ ys) ++ zs) =
     nub (xs ++ nub (ys ++ zs))
Searching for inverse function properties...
  5. reverse (reverse xs) = xs
Searching for distributivity properties...
  6. map f xs ++ map f ys = map f (xs ++ ys)
  7. sort (sort xs ++ sort ys) = sort (xs ++ ys)
  8. nub (nub xs ++ nub ys) = nub (xs ++ ys)
```

Each property is tagged with the name of the template that generated it. For example, the first law is an instance of commutativity, `?F X Y = ?F Y X`, with `?F = \xs ys -> sort (xs ++ ys)`. (Section 2 describes the strategy RoughSpec uses to instantiate metavariables.) We see that `++` is associative (2), that `reverse` is its own inverse (5), and that `map` distributes over `++` (6). We also see that composing `++` with `sort` or `nub` produces a binary operation which satisfies many properties in its own right.

By adding more templates, we can find more laws. For example, adding the template `?F (?G X) = ?G (?F X)` produces the law `map f (reverse xs) = reverse (map f xs)`. We have not found all of the important list laws (for example, the law `reverse (xs++ys) = reverse ys ++ reverse xs`), but have produced a useful and short subset.

The templates we have used so far represent well-known properties and apply to a wide range of APIs. The goal of RoughSpec is that the user can start with a "standard" set of templates, and find an incomplete, but useful set of properties for their program. Then they can find more detailed properties by adding templates that are tailored to their domain. By putting the user in charge of choosing templates, we aim to keep the output small and easy to understand.

Our current implementation of RoughSpec is built on top of QuickSpec [18], but the ideas are not QuickSpec-specific, and would work equally well on top of a different theory exploration system such as Speculate [1]. The only difference is that as Speculate also discovers inequalities and conditional laws, a RoughSpec built on top of Speculate would naturally support inequalities and conditions in templates.

Next, we describe how RoughSpec works, and evaluate it on some larger examples.

2 HOW IT WORKS

To use RoughSpec, the user inputs the templates they are interested in, along with the functions they want to explore, in a *signature* [18]. The user must also supply QuickCheck [4] test data generators for any non-standard types they want to test. Figure 1 shows an example of a simple signature, consisting of the functions `reverse`, `++` and `length`, and the template `?F (?G X Y) = ?F (?G Y X)`. Note that in our current implementation, functions are written uncurried, but for readability we write them curried in the main text of this article.

```
simpleSig = [
  con "reverse" (reverse :: [A] -> [A]),
  con "++" ((++) :: [A] -> [A] -> [A]),
  con "length" (length :: [A] -> Int),
  template "nest-commute" "?F(?G(X,Y))=?F(?G(Y,X))"
  ]
```

Figure 1: A signature containing some list functions and a template for properties about nested composition of functions being commutative in two variables.

As described in Section 1, the templates are expressed in a simple term language consisting of:

- metavariables, which represent *holes to be filled* with a function or constant symbol, and are written as a question mark followed by a name;
- variables, which are universally quantified, and are written as a name starting with a capital letter; and
- functions drawn from the signature.

Candidate properties are generated by attempting to fill the holes in a template using function symbols from the signature, making sure the generated equations are well typed. For our example template above, filling in the holes using the functions `length`, `reverse`, and ++ gives two candidate properties:

`length (xs ++ ys) = length (ys ++ xs)` (*cp*1)
`reverse (xs ++ ys) = reverse (ys ++ xs)` (*cp*2)

The generated candidate properties are then tested using QuickCheck [4]. If no counterexamples are found the property is presented to the user as a law. In our example, *cp*1 passes this phase and is presented to the user, while *cp*2 fails and is discarded.

2.1 Expanding templates

In the algorithm described above, each hole in a template can be filled only with precisely one of the function symbols in scope. This means that each template matches a rather narrow class of properties. As we shall see, many properties that we would intuitively consider to be in the same category are not instances of the same template, so the user is forced to write many similar templates to capture a category of properties in full generality.

To help the user avoid the tedious work of typing up a set of nearly-identical templates, and to improve the expressiveness of the discovered properties, we have implemented some automated "expansion" of user input templates, which augments the user-provided templates by automatically adding variants of them.

2.1.1 Nested functions. Consider the property

`length (xs ++ ys) = length (ys ++ xs)` (*cp*1)

discovered in our example above. In that example, we discovered the property using a template that specifically described the composition of two function symbols being commutative. Suppose we had a more general template for commutativity, i.e. `?F X Y = ?F Y X`. What if we want such a template to cover properties like *cp*1, rather than having to type up a specialised "composition of two functions" commutativity template?

In order to do this we have implemented an extension allowing a hole to be filled by the composition of two function symbols. We replace a given hole in our template with two holes representing an outer function applied to an inner function applied to the original hole's arguments. That is, a hole of the form `?F e1...en` turns into `?G (?F e1...en)`. This allows us to discover the property *cp*1 using the commutativity template `?F X Y = ?F Y X`. It also allows us to use a general template for identity functions, `?F X = X`, to discover the property `reverse (reverse xs) = xs`, that is, that `reverse . reverse` is the identity function.

2.1.2 Partial application. Suppose we extend our example signature from Figure 1 by adding the function `map` and a distributivity template

`?F (?G X Y) = ?G (?F X) (?F Y),` (*d*1)

describing a function `?F` distributing over a two-argument function `?G`.

We would like to discover the property

`map f (xs ++ ys) = map f xs ++ map f ys,` (*dmap*)

describing how `map` distributes over ++. However, since our template holes can only be filled using precisely one function symbol (or, following Section 2.1.1, two nested function symbols), this template does not cover the desired property. Instead we would need a more complex template like

`?F X (?G Y Z) = ?G (?F X Y) (?F X Z),`

with an extra variable `X` for `map`'s first argument.

In order to avoid needing a variety of complicated templates when our signatures contain functions with varying numbers of arguments, we allow a template hole to be filled with a partially applied function. We replace a given hole in our template with a hole applied to a number of fresh variables, limited by the maximum arity of the functions in scope. By doing so our desired property *dmap* is now covered by the template *d*1.

In combination with our nested function expansion described above, this also allows us to discover properties such as

```
map f (concat (xss ++ yss)) =
  map f (concat xss) ++ map f (concat yss),
```

by adding the `concat` function to our signature, using the same template *d*1.

This method considers all possible partially-applied functions when filling a hole. In practice we found this to give rise to some rather confusing properties when binary operators were involved. For instance, suppose we extend our example signature with a template `?F (?G X) = ?F X`, which describes pairs of functions `?F` and `?G` where the result of `?F` is preserved when we apply `?G` to its argument. This gives rise to nice properties such as

```
length (reverse xs) = length xs
length (map f xs) = length xs,
```

but also such properties as

`length (xs ++ reverse ys) = length (xs ++ ys),`

where the hole `?F` has been filled by the function `length . (xs ++)`.

We find properties about partially applied functions such as `xs ++` rather confusing and uninteresting, and therefore decided to restrict this expansion. Our restriction is that if a function is a binary operator (specifically, it has two arguments, both of which have the same type) we do not allow a partial application of it to fill a hole.

2.1.3 Restricting expansion. Expanding templates automatically is a delicate balance. In moderation, it produces interesting properties that users want to see, and that intuitively match the given template. If we expand templates too much, we may generate irrelevant properties, overwhelm the user with output or increase the running time of our tool. As can be seen from the special treatment of binary operators in Section 2.1.2, we have implemented some ad hoc restrictions to our expansion heuristics to prevent them from producing properties we found less interesting. In general, which expansions are appropriate and how they should be used seems to depend on the context, the kinds of functions being explored and the user's priorities. In the future, we plan to give the user more control over the expansion process by making the language for describing templates more expressive. For example, the user should be able to specify which functions they are comfortable seeing partially applied.

2.2 Pruning

There is one last ingredient in RoughSpec's algorithm. Suppose we run the algorithm so far on a signature consisting of the functions `reverse`, `++`, `length` and `map`, and two templates from earlier:

- `?F X = X`, expressing that `?F` is the identity function.
- `?F (?G X) = ?F X`, expressing that the result of `?F` is unchanged when we apply `?G` to its argument.

We are presented with the following output:

```
Searching for identity properties...
 1. reverse (reverse xs) = xs
Searching for preserve properties...
 2. length (reverse xs) = length xs
 3. length (map f xs) = length xs
 4. length (reverse (reverse xs)) = length (reverse xs)
 5. length (reverse (map f xs)) = length (reverse xs)
 6. length (map f (reverse xs)) = length (map f xs)
 7. length (map f (map g xs)) = length (map f xs)
 8. reverse (reverse (reverse xs)) = reverse xs
 9. (++) (reverse (reverse xs)) = (++) xs
10. length (reverse (reverse xs)) = length xs
11. length (reverse (map f xs)) = length xs
12. length (map f (reverse xs)) = length xs
13. length (map f (map g xs)) = length xs
14. map f (reverse (reverse xs)) = map f xs
```

Some of these properties appear to be redundant. For instance, property 4 is an instance of property 2, in which `xs` has been replaced by `reverse xs`. Surely our user isn't interested in seeing a property that's just a more specific instance of a previously discovered property?

To avoid cluttering the output with redundant properties, RoughSpec includes a *pruning* phase, which aims to remove discovered properties that are trivial consequences of earlier properties. The pruning phase removes a property if:

- It is an *instance* of an earlier property. In this case, properties 4 and 8 will be pruned away, as they are instances of properties 2 and 1, respectively.
- It can be obtained applying the same function to both sides of an earlier property. For example, property 10 can be obtained by applying the `length` function to both sides of property 1.

More formally, a property is removed if it is of the form $t\sigma = u\sigma$ or $C[t] = C[u]$, where $t = u$ is a previously-discovered property, σ is a substitution and C is a context. In our example, we will discard properties 4, 8, 9, 10, and 14, so that a total of 9 properties remain.

This still leaves us with some redundant properties. For example, property 5 follows by equational reasoning from properties 2 and 3. If we were also to prune away properties that follow by equational reasoning from earlier properties, we would be left with only the three properties 1–3 above. Unfortunately, it is not a good idea to remove all properties that follow by equational reasoning from earlier properties, because the removed properties are often *non-trivial* consequences of the discovered properties, and may well be interesting to the user. The problem is how to add some form of equational reasoning, to remove more redundant properties, but without removing redundant properties too aggressively.

Here is our solution. Because the templates given to RoughSpec describe the exact shapes of properties that the user is interested in, we should be careful not to go too far in pruning away properties matching those desired shapes. We therefore distinguish between two kinds of properties, depending on whether *template expansion* (Section 2.1) was needed to discover the property:

- If the property was found without using expansion, we use the simple pruning algorithm described above, which removes a property if it is a trivial consequence of a single existing property.
- If the property was found using expansion, we use the more powerful pruning algorithm from QuickSpec [18], which removes the property if it follows by equational reasoning from the earlier properties.

For instance, property 11 is pruned away as it follows by equational reasoning from properties 2 and 3, and was generated from an expanded template. However, if we added the template `?F (?G (?H F X)) = ?F X` to our signature, then property 11 would precisely match an input template, so pruning by equational reasoning would be disabled for that property, and it would no longer be pruned away. On the other hand, property 4 will always be pruned away, even if we add a template exactly matching it, as it is a direct instance of property 1.

With this last improvement added, RoughSpec produces the following three properties, and all others are pruned away:

```
Searching for identity properties...
  1. reverse (reverse xs) = xs
Searching for preserve properties...
  2. length (reverse xs) = length xs
  3. length (map f xs) = length xs
```

As properties discovered earlier are used to prune away ones that are discovered later, the order in which the templates are input makes a difference to which properties we output. It is usually a good idea to start with smaller and/or more general templates and move on to larger and/or more specific ones, as smaller properties are more likely to be useful in pruning larger ones, but the user may also want to put the templates they find most relevant first.

2.3 Libraries of default templates

We have compiled a small set of default templates capturing very common properties which we found useful in our case studies:

```
identity: ?F X = X
fixpoint: ?F ?X = ?X
left-id-elem: ?F ?Y X = X
right-id-elem: ?F X ?Y = X
cancel: ?F (?G X) = ?F X
commutative: ?F X Y = ?F Y X
commuting-functions: ?F (?G X) = ?G (?F X)
distributivity: ?F (?G X Y) = ?G (?F X) (?F Y)
homomorphism: ?F (?G X) (?G Y) = ?G (?H X Y)
associativity: ?F (?F X Y) Z = ?F X (?F Y Z)
```

These capture standard algebraic properties, and can be imported into a signature as a means to quickly run a first pass of exploration on a new theory without having to define any templates yourself. The user can then, if need be, simply extend this set with more specific templates.

3 CASE STUDIES

The following examples demonstrate theory exploration using our template-based approach and discuss what kinds of templates we have found to be useful. We compare our results to theory exploration with QuickSpec on the same sets of functions. The code is available at `https://github.com/solrun/quickspec`, in the `template-examples/ifl2020` directory, along with detailed experiment output. All experiments described in this paper were performed on a ThinkPad X260 laptop with a 2.5GHz Intel i7-6500U processor and 16GB of RAM running 64-bit Linux.

Runtimes and memory use. Exploring the large library of list functions in Section 3.3 took just under 8 minutes and reached a maximum heap residency of 700 MB. All other examples ran in < 20s with a maximum heap residency of < 100 MB.

3.1 Pretty Printing

This case study shows how RoughSpec can be useful in understanding an unfamiliar library. Suppose we are using Hughes's pretty-printing library [9] for the first time. We are presented with an intimidating array of combinators:

```
empty :: Doc
text :: String -> Doc
nest :: Int -> Doc -> Doc
(<>) :: Doc -> Doc -> Doc
(<+>) :: Doc -> Doc -> Doc
($$) :: Doc -> Doc -> Doc
hcat :: [Doc] -> Doc
hsep :: [Doc] -> Doc
vcat :: [Doc] -> Doc
sep :: [Doc] -> Doc
fsep :: [Doc] -> Doc
```

The library documentation explains that `Doc` represents a pretty-printed document, `empty` is an empty document, `text` prints a string verbatim, and `nest` indents an entire document by a given number of spaces. The remaining functions combine multiple documents into one:

- <>, <+> and $$ typeset two documents beside one another, beside one another with a space in between, or one above the other, respectively.
- `hcat`, `hsep` and `vcat` are variants of <>, <+> and $$ that take a *list* of documents.
- `sep` and `fsep` choose whichever of <+> and $$ gives the prettiest output.

We may now feel happy going off and writing some pretty printers. But there are still questions unanswered:

- What is the difference between `empty` and `text ""`?
- If I am indenting a multi-line document, should I apply `nest` to each line individually or to the whole document?
- Does it matter if I use <> or `hcat`, <+> or `hsep`, $$ or `vcat`?
- Why is there no analogue of <> for `sep` and `fsep`?

These are the kinds of questions a formal specification of the pretty-printing library would answer. Let us see if RoughSpec can help us.

We start with the list of default templates from 2.3. We reproduce RoughSpec's output verbatim. It finds the following 46 laws:

```
Searching for identity properties...
  1. hcat (unit x) = x
  2. hsep (unit x) = x
  3. vcat (unit x) = x
  4. sep (unit x) = x
  5. fsep (unit x) = x
Searching for fixpoint properties...
 6. nest x empty = empty
 7. nest x (hcat []) = hcat []
 8. nest x (hsep []) = hsep []
 9. nest x (vcat []) = vcat []
 10. nest x (sep []) = sep []
 11. nest x (fsep []) = fsep []
Searching for left-id-elem properties...
 12. nest 0 x = x
 13. empty <> x = x
 14. empty <+> x = x
 15. empty $$ x = x
 16. hcat [] <> x = x
 17. hsep [] <> x = x
 18. vcat [] <> x = x
 19. sep [] <> x = x
 20. fsep [] <> x = x
 21. hcat [] <+> x = x
 22. hsep [] <+> x = x
 23. vcat [] <+> x = x
 24. sep [] <+> x = x
 25. fsep [] <+> x = x
 26. hcat [] $$ x = x
 27. hsep [] $$ x = x
 28. vcat [] $$ x = x
 29. sep [] $$ x = x
 30. fsep [] $$ x = x
Searching for right-id-elem properties...
```

```
 31. x <> empty = x
 32. x $$ empty = x
 33. x <+> empty = x
 34. x <> text [] = x
Searching for cancel properties...
 35. length (unit (nest x y)) = length (unit y)
Searching for commutative properties...
Searching for commuting-functions properties...
 36. nest x (nest y z) = nest y (nest x z)
Searching for distributivity properties...
 37. nest x (y <> z) = nest x y <> nest x z
 38. nest x (y $$ z) = nest x y $$ nest x z
 39. nest x (y <+> z) = nest x y <+> nest x z
Searching for homomorphism properties...
 40. text xs <> text ys = text (xs ++ ys)
 41. hcat xs <> hcat ys = hcat (xs ++ ys)
 42. vcat xs $$ vcat ys = vcat (xs ++ ys)
 43. hsep xs <+> hsep ys = hsep (xs ++ ys)
Searching for associative properties...
 44. (x <> y) <> z = x <> (y <> z)
 45. (x $$ y) $$ z = x $$ (y $$ z)
 46. (x <+> y) <+> z = x <+> (y <+> z)
```

Laws 7–11 are curious. They are all rather similar, and do not look very interesting. In fact, each of these laws contains a term (such as `hsep []` or `vcat []`) which is actually equal to `empty`. Once we know that, we see that these laws are trivial restatements of law 6. The same issue occurs with laws 16–30, which are trivial restatements of 13–15. The problem is that there was no template which allowed RoughSpec to discover laws such as `hsep [] = empty`.

We shall see an automatic fix for this problem in Section 4.1. For now, we fix it by adding the template `?F ?X = ?Y`. This template finds 12 laws, including `hsep [] = empty` and its companions, and now laws 7–11 as well as laws 16–30 are pruned away as they follow from 6 and 13–15 respectively. Law 35 is also pruned away as our new template finds the simpler and subsuming property `length (unit x) = length (unit y)` We are left with a total of 25 laws: 1–6, 12–15, 31–34, and 36–46 above.

Together, these laws answer most of the questions we posed above. The difference between `empty` and `text ""` is that `empty` acts as an identity for the other operators:

```
empty <>  x = x     x <>  empty = x
empty <+> x = x     x <+> empty = x
empty $$  x = x     x $$  empty = x
```

On the other hand, `text ""` mostly does not, only satisfying one identity law:

```
x <> text "" = x
```

If we want to find out why `text ""` is not an identity element, we can now use QuickCheck to find a revealing counterexample (or indeed read Hughes [9] for an explanation).

As for whether one should indent each line separately or the whole document at once, it doesn't matter, because `nest` distributes over `$$`:

```
nest x (y $$ z) = nest x y $$ nest x z
```

Another distributivity law tells us that we can freely choose to typeset a long string in one go, or split it up into smaller pieces:

```
text xs <> text ys = text (xs ++ ys)
```

The `<>`, `<+>` and `$$` operators are associative:

```
(x <> y) <> z   = x <> (y <> z)
(x <+> y) <+> z = x <+> (y <+> z)
(x $$ y) $$ z   = x $$ (y $$ z)
```

and `hcat`, `vcat` and `hsep` appear to be those operators folded over a list:

```
hcat xs <> hcat ys = hcat (xs ++ ys)
vcat xs $$ vcat ys = vcat (xs ++ ys)
hsep xs <+> hsep ys = hsep (xs ++ ys)
```

Therefore, it doesn't matter whether one uses e.g. `<>` or `hcat`—they are equivalent.

Associativity of course means that we can write e.g. `x <> y <> z` without worrying about bracketing. We might wonder whether the same applies to sequences of mixed operators, e.g. `x <> y <+> z`. To find out we can add another template:

```
mixed-associativity: ?G (?F X Y) Z = ?F X (?G Y Z)
  -- as infix: (X `?F` Y) `?G` Z = X `?F` (Y `?G` Z)
```

This reveals that, indeed, a whole host of expressions can be freely rebracketed:

```
nest x y <> z = nest x (y <> z)
(x $$ y) <> z = x $$ (y <> z)
(x <+> y) <> z = x <+> (y <> z)
nest x y <+> z = nest x (y <+> z)
(x <> y) <+> z = x <> (y <+> z)
(x $$ y) <+> z = x $$ (y <+> z)
```

Finally, we come to the question of why there is no two-argument version of `sep` and `fsep`. Given what we learnt above, we might suspect that these operators are not associative. To test this, we can add two new functions to the signature:

```
sep2, fsep2 :: Doc -> Doc -> Doc
sep2 x y = sep [x, y]
fsep2 x y = fsep [x, y]
```

Indeed, no new associativity law appears.[2] Nor is it the case that e.g. `fsep2 (fsep xs) (fsep ys) = fsep (xs ++ ys)`. In fact, no interesting laws of any kind appear.

The laws that `hsep` and family satisfy are very useful when programming. When we want to typeset a list of documents horizontally, we can either use `hsep`, `<+>` or a mixture (e.g. we may write `hsep xs <+> hsep ys` instead of `hsep (xs++ys)`). By contrast, when using `sep` or `fsep`, we must carefully collect all documents into a list and only then apply the combinator. In this case, the *lack of a nice specification* is itself useful information: it warns us that we should take care when using these combinators!

Summary. RoughSpec performed well on the pretty-printing library. It produced a manageable number of equations, all of them simple and easily understood. Despite their simplicity, they answered important questions about how to use the library—the questions listed at the top of this section. We believe that even simple properties, such as associativity and distributivity laws, are a great help in understanding how to use a new library. Finally, we got

[2]Exercise for the reader: reading the documentation of the `pretty` library, it seems reasonable that `fsep2` could be associative. Why is it not?

good results from a "standard" set of templates and were able to improve the output by adding our own.

The one hiccup in RoughSpec's performance was laws 7–11 and 16–30. We were forced to add a template specifically to prune away these laws. In fact, another instance of the same problem occurred: `sep` and `fsep` only differ on lists of at least three elements, which means that `sep2 = fsep2`. QuickSpec discovers this law instantly, but RoughSpec failed to find it as there was no template of the form `?X = ?Y`. Instead, laws about this function appear twice—once with `sep2` and once with `fsep2`.

In both cases, we have two laws containing syntactically different terms *that are actually equal*—for example, `hcat []` and `hsep []`. RoughSpec ought to detect that the terms are equal, and avoid generating duplicate laws. One option is to gather all the terms used to instantiate metavariables, divide them into equivalence classes by testing, and keep only the representative of each equivalence class. Section 4 describes an alternative solution to this problem.

Comparison with QuickSpec. As reported in [18], QuickSpec does well given the combinators `text`, `nest`, `<>`, `<+>` and `$$`, finding a complete specification that matches the one given by Hughes [9]. Unfortunately, when we add `hcat` and friends, QuickSpec finds many complicated, unimportant-looking laws, for example:

```
40. fsep (xs ++ [empty] ++ ys) = fsep (xs ++ ys)
41. hcat (xs ++ [empty] ++ ys) = hcat (xs ++ ys)
42. hsep (xs ++ [empty] ++ ys) = hsep (xs ++ ys)
43. hcat (xs ++ [hcat ys] ++ zs) =
      hcat (xs ++ ys ++ zs)
44. hsep (xs ++ [hsep ys] ++ zs) =
      hsep (xs ++ ys ++ zs)
45. fsep (xs ++ [x $$ (y $$ z)]) =
      fsep xs $$ (x $$ (y $$ z))
46. fsep (xs ++ [x $$ x] ++ ys) =
      fsep xs $$ ((x $$ x) $$ fsep ys)
```

What's more, QuickSpec now takes several minutes to run, while RoughSpec takes 15 seconds. In short, RoughSpec copes much better than QuickSpec once the combinators no longer have a simple specification.

3.2 Model-based properties

In [10], Hughes compared five different methods of defining properties for QuickCheck testing, and found the most effective to be *model-based* testing, which revealed all the bugs in his test programs and required only a small number of properties to be written.

Model-based testing is based on the approach to proving the correctness of data representations introduced by Hoare in [8]. The data representation is related to an appropriate abstract representation using an *abstraction function*. For each operation $op : X \to X$, an abstract implementation $op_{abstract} : X_{abstract} \to X_{abstract}$ is defined, and the following diagram is proven to commute:

$$\begin{tikzcd}
X \arrow[r, "op"'] \arrow[d, "\text{abstraction}"'] & X \arrow[d, "\text{abstraction}"] \\
X_{abstract} \arrow[r, "op_{abstract}"'] & X_{abstract}
\end{tikzcd}$$

The correctness of the data representation and operations in question then follows from the (presumably simpler) correctness proofs for the abstract data and operations.

In model-based testing, we define the same kind of abstract model of the data structure being tested, then *test* that the above diagram commutes. In [10], bugs in the implementation of concrete operations are found to cause counterexamples to such properties.

Since we can include specific function symbols from the exploration scope in our templates, we can use RoughSpec to search only for properties that relate two operations via a given abstraction function, with a template along the lines of:

```
?F (abstraction X) = abstraction (?G X).
```

3.2.1 *Binary search trees.*

In [10], Hughes uses binary search trees as an example and defines five model-based properties relating the tree operations to operations on a list of key-value pairs with *toList* as an abstraction function.

```
1. find x t = findList x (toList t)
2. insertList x (toList t) = toList (insert x t)
3. deleteKeyList x (toList t) = toList (delete x t)
4. toList nil = []
5. toList (union t t1) =
   sort (unionList (toList t) (toList t1))
```

Running RoughSpec on a signature containing the relevant functions and three templates describing model-based properties, we discover precisely these five properties and two additional properties, shown below, in just over 0.3 seconds.

```
6. toList (delete x nil) = []
7. toList nil = sort []
```

Note that properties 6 and 7 are both equivalent to property 4 above. However, since RoughSpec only searches for properties matching the given templates and nothing else, it won't discover that `sort [] = []` or that `delete x nil = nil`, even though these are properties a human might find rather trivial and obvious and would want the pruner to know about. This can be improved by running QuickSpec up to a small term size to provide background information, as discussed in Section 4.

Due to the different shapes of the desired properties we need three different templates to discover them all.

```
?F Y (toList X) = ?G Y X
toList ?X = ?Y
toList (?G X Y) = ?F (toList X) (toList Y)
```

With a more expressive template language, as discussed in Section 2.1, we could manage with fewer, more general templates. For example, all three templates are instances of the general shape `toList (?F X1...Xn) = ?G (toList X1) ... (toList Xn)`, which captures the properties used in model-based testing.

Comparison with QuickSpec. QuickSpec discovers 28 properties about the functions in our signature, among them the five model-based properties. This takes between 10 and 11 seconds, significantly longer than RoughSpec's 0.3 seconds.

3.3 A large library of list functions

Section 4.2 of [18] describes a stress-test where QuickSpec was used to find properties about a set of 33 Haskell functions on lists. This took standard QuickSpec 42 minutes and resulted in 398 properties when limited to terms of size 7 or less, and hit a time limit of 2 hours when the size was increased to 8. As described in the Introduction, many of the laws found by QuickSpec were not interesting. This illustrates how running QuickSpec on larger theories scales poorly with regard to run-time and may produce an overwhelming amount of output. When we ran the most recent version of QuickSpec on this set of functions it ran out of memory and did not manage to produce any properties.

```
length    :: [A] -> Int
sort      :: [Int] -> [Int]
scanr     :: (A -> B -> B) -> B -> [A] -> [B]
(>>=)     :: [A] -> (A -> [B]) -> [B]
reverse   :: [A] -> [A]
(>=>)     :: (A -> [B]) -> (B -> [C]) -> A -> [C]
(:)       :: A -> [A] -> [A]
break     :: (A -> Bool) -> [A] -> ([A], [A])
filter    :: (A -> Bool) -> [A] -> [A]
scanl     :: (B -> A -> B) -> B -> [A] -> [B]
zipWith   :: (A -> B -> C) -> [A] -> [B] -> [C]
concat    :: [[A]] -> [A]
zip       :: [A] -> [B] -> [(A, B)]
usort     :: [Int] -> [Int]
sum       :: [Int] -> Int
(++)      :: [A] -> [A] -> [A]
map       :: (A -> A) -> [A] -> [A]
foldl     :: (A -> A -> A) -> A -> [A] -> A
takeWhile :: (A -> Bool) -> [A] -> [A]
foldr     :: (A -> A -> A) -> A -> [A] -> A
drop      :: Int -> [A] -> [A]
dropWhile :: (A -> Bool) -> [A] -> [A]
span      :: (A -> Bool) -> [A] -> ([A], [A])
unzip     :: [(A, B)] -> ([A], [B])
[]        :: [A]
partition :: (A -> Bool) -> [A] -> ([A], [A])
take      :: Int -> [A] -> [A]

Background functions:
(,)  :: A -> B -> (A, B)
fst  :: (A, B) -> A
snd  :: (A, B) -> B
(+)  :: Int -> Int -> Int
0    :: Int
succ :: Int -> Int
```

Figure 2: A library of list functions.

In contrast, running RoughSpec on this set of functions we can tailor the templates we use to properties we are interested in discovering and produce a more manageable amount of output in a much shorter time. The list of functions is shown in Figure 2. The last six functions are declared as *background* functions. Background functions may appear in properties, but a discovered property must contain at least one non-background function.

Running RoughSpec on this set of functions with the library of 10 default templates presented in Section 2.3, we discover 184 properties in just under 8 minutes. The properties include many useful laws, such as distributivity-like properties:

```
length xs + length ys = length (xs ++ ys)
concat xss ++ concat yss = concat (xss ++ yss)
sum xs + sum ys = sum (xs ++ ys)
```

Template expansion results in more complex properties. The second property below has size 11, much larger than QuickSpec was able to discover:

```
take x (takeWhile p (zip xs ys)) =
  takeWhile p (zip (take x xs) (take x ys))
take x (zipWith f xs (zipWith g ys zs)) =
  zipWith f xs (zipWith g (take x ys) (take x zs))
```

These two properties are given as examples of distributivity (`take` is distributed over the rest of the expression). The user may not consider these laws interesting, which suggests that having a more expressive template language is important. Nonetheless, the laws discovered are better than those found by QuickSpec, and we are able to discover them in a fraction of the time. This demonstrates that RoughSpec is much better suited than QuickSpec to exploring large libraries of functions, and that it makes theory exploration tractable on such libraries that were previously infeasible to explore.

3.4 A window manager

The xmonad window manager [20] is a tiling window manager for X, written in Haskell. We take an implementation of a simple window manager with multiple virtual workspaces containing stacks of screens, in the style of xmonad, as shown in [19], and demonstrate how RoughSpec is useful for finding properties about this data structure.

The data structure `StackSet` represents a set of workspaces each containing a stack of screens, and there are a number of operators to construct and manipulate `StackSet`s, shown below:

```
empty   :: Natural -> StackSet a
current :: StackSet a -> Natural
view    :: Natural -> StackSet a -> StackSet a
peek    :: StackSet a -> a
rotate  :: Ordering -> StackSet a -> StackSet a
push    :: a -> StackSet a -> StackSet a
shift   :: Natural -> StackSet a -> StackSet a
insert  :: a -> Natural -> StackSet a -> StackSet a
delete  :: a -> StackSet a -> StackSet a
index   :: Natural -> StackSet a -> [a]
```

The operator `empty` creates a `StackSet` containing a given number of empty workspaces, `current` returns the current workspace, `view` sets the current workspace, `peek` extracts the screen on top of the current workspace's stack, `rotate` cycles the current screen stack up or down, `push` inserts an screen on top of the current stack, `shift` moves the screen on top of the current stack to the top of stack *n*, `insert` inserts a screen on top of stack *n*, `delete` deletes a given screen, and `index` extracts the workspace at a given index.

Running RoughSpec with the set of default templates described in Section 2.3 we discover the following 23 properties:

```
Searching for identity properties...
Searching for fix-point properties...
  1. current (empty 0) = 0
  2. view x (empty 1) = empty 1
  3. rotate o (empty 1) = empty 1
  4. shift x (empty 1) = empty 1
  5. delete x (empty 1) = empty 1
Searching for left-id-elem properties...
  6. rotate EQ s = s
  7. current (empty x) + y = y
Searching for right-id-elem properties...
  8. peek (push x s) = x
Searching for cancel properties...
  9. current (rotate o s) = current s
 10. current (push x s) = current s
 11. current (shift x s) = current s
 12. current (delete x s) = current s
 13. current (insert x y s) = current s
 14. current (view x (rotate o s)) =
       current (view x s)
 15. current (view x (push y s)) =
       current (view x s)
 16. current (view x (shift y s)) =
       current (view x s)
 17. current (view x (delete y s)) =
       current (view x s)
 18. current (view x (insert y z s)) =
       current (view x s)
Searching for commutative properties...
Searching for op-commute properties...
 19. view x (delete y s) = delete y (view x s)
 20. rotate o (rotate o' s) = rotate o' (rotate o s)
 21. delete x (delete y s) = delete y (delete x s)
 22. view x (insert y z s) = insert y z (view x s)
 23. shift x (push y (view z s)) =
       view z (shift x (push y s))
Searching for 2-distributive properties...
Searching for homomorphism properties...
Searching for associative-3 properties...
```

We may note that our fix-point properties, 1–5, look rather specific, referring to `empty 0` and `empty 1`, and wonder whether they are perhaps valid for `empty n` for more values of n. Property 7, `current (empty x) + y = y`, implies that `current (empty x) = 0`, subsuming property 1, but in a verbose and unclear manner. However, the more interesting property `current (empty x) = 0` does not fit any of our templates. Testing with QuickCheck reveals that properties 3–5 hold for `empty n` for all values of $n > 0$, not just `empty 1`, while property 2 is specific to $n = 1$ (if and only if we have just one workspace in our set, the result of setting a given one of the workspaces as the current one must always be the same). This demonstrates that even when RoughSpec does not generate the most general valid properties for our program, it can help us to gain a better understanding of the functions in our program and come up with more general properties to test and create a more complete specification. However, adding support for conditions such as $n > 0$ could enable us to generate more elegant and useful properties.

The properties shown above, discovered by RoughSpec using our list of default templates, provide us with some insight into how this window manager works. However, it clearly does not provide a complete specification and some operations hardly appear in the properties discovered. If we add a template `?F X = ?F (?F X)` describing idempotency we discover 15 further properties. The first 4, 24–27 shown below, tell us that `view`, `push`, `delete`, and `insert` are idempotent, which is interesting and useful information.

```
24. view x s = view x (view x s)
25. push x s = push x (push x s)
26. delete x s = delete x (delete x s)
27. insert x y s = insert x y (insert x y s)
```

Properties 28–39, however, describe the idempotency of various combinations of the previously mentioned functions (along with the function `shift` in the case of 31 and 33), for example:

```
33. shift x (push y s) =
      shift x (push y (shift x (push y s)))
38. insert x y (delete z s) =
      insert x y (delete z (insert x y (delete z s)))
```

These properties look rather complicated and uninteresting. They are generated due to our template expansions, and once again indicate that in some cases we should limit this expansion. As before, it may be useful to have a more descriptive template language where the user can indicate whether or not a given template should be expanded.

Comparison with hand-written properties and QuickSpec. On this example, QuickSpec takes 12 seconds, while RoughSpec takes 7 seconds.

In [19] the author defines a set of 18 QuickCheck test properties forming a specification for this window manager. Of these 18, five are conditional and therefore cannot be found by our current implementation of RoughSpec.

The author defines an invariant for `StackSets` stating that the current workspace index is within bounds (between 0 and the size of the set), and that the set does not contain any duplicate screens. They test that this invariant holds for all `StackSets` generated by the QuickCheck generator (`invariant s`) and also that it still holds after applying various operations.

```
1. invariant x
2. invariant $ view n x
3. invariant $ rotate n x
4. invariant $ push n x
5. invariant $ delete n x
6. invariant $ shift n x
7. invariant $ insert n i x
```

Note that having discovered property 1 above, both RoughSpec and QuickSpec would prune away properties 2–6, deeming them redundant. However they can be useful as test properties when we don't trust the test data generator to provide good coverage of the range of potential values.

If we add the `invariant` function to our signature, QuickSpec will discover the property `invariant s`. Of the 13 equational properties defined in [19], QuickSpec discovers 4.

RoughSpec, on the other hand, discovers the equivalent but more clunky property `invariant s && x = x`, although it will discover `invariant s` if we add an invariant template `?F ?X = True`, which is perhaps a reasonable template to include when we have boolean-valued functions in our exploration scope. Three of the equational properties defined in [19] are found by RoughSpec with the idempotency template and one of the remaining ones is `current (empty n) = 0`, where RoughSpec discovered the equivalent but less elegant `current (empty x) + y = y`. The problem is similar to the one we encountered in the pretty-printing and binary search examples: RoughSpec instantiates the schema `?E + X = X` with both `?E = current (empty x)` and `?E = 0`, but does not notice that these two terms are equal.

4 A HYBRID APPROACH

RoughSpec and QuickSpec's approaches seem to be complementary. For large APIs, QuickSpec is slow, and often produces an overwhelming amount of output. By contrast, RoughSpec runs quickly, and produces a moderate number of laws. The laws it finds are easy to understand, because they follow standard patterns, and can be targeted to the user's interests.

On the other hand, RoughSpec does not usually find a complete specification. Even when testing lists, RoughSpec failed to find the law `reverse (xs ++ ys) = reverse ys ++ reverse xs`, as it did not match any of the provided templates. This is by design but is nonetheless a weakness.

There is another problem. If RoughSpec is given very general templates, our premise of limiting the search space may no longer hold. For example, consider a template `?F X = ?G X` searching for equivalent functions. This template could produce interesting and useful properties, for instance stating that different sorting functions produce the same output for a given input. However, if our signature contains many functions that have the same type we will produce a large number of candidate properties and testing them will take a long time (and probably most will be falsified). Meanwhile, QuickSpec will discover relevant properties of this shape much more quickly.

4.1 Hybrid tool

To solve these problems, this section introduces a *hybrid* approach. The idea is to run QuickSpec to find all *small* laws, running with a small size limit, and then run RoughSpec to find interesting laws beyond that size limit.

We have implemented a tool that combines QuickSpec and RoughSpec. The user can call `roughSpecWithQuickSpec k s` on their signature `s`, where `k` is an integer parameter specifying up to which term size QuickSpec should go. This prompts QuickSpec to run on the signature up to the specified term size and print out the properties discovered. Then RoughSpec is run, using the templates supplied, but with knowledge of the properties discovered by QuickSpec. The pruner removes any properties subsequently discovered by RoughSpec that follow from those discovered by QuickSpec. This allows us to discover elegant and useful smaller properties using QuickSpec that would be cumbersome and time-consuming to find with RoughSpec as they would require a too-general template. It also helps us prune away redundant and uninteresting properties discovered by RoughSpec. Below we examine the output of the new hybrid tool on the examples we introduced in Section 3.

4.2 Pretty Printing

Our pretty-printing library example from Section 3.1 provides a good illustration of how combining QuickSpec and RoughSpec helps us find better properties.

There, RoughSpec found the undesired properties 7–11 and 16–30, which described semantically equivalent properties of various terms that were all equal to `empty`. In order to remove these properties, we were forced to add a template `?F ?X = ?Y`. This template has a very general shape and can hardly be said to describe a family of interesting properties. Very many true properties as well as falsifiable candidate properties are likely to match this template, so including it is likely to harm RoughSpec's performance. In the case of our pretty-printing signature, adding this template nearly doubles the runtime, taking it from 16 seconds with just the default templates to 28 seconds. However, running our hybrid tool on the pretty-printing signature with QuickSpec up to size 2, we get the same benefits but with a runtime of 14.5 seconds. QuickSpec discovers precisely the 5 laws stating that `hcat []`, `hsep []`, `vcat []`, `sep []` and `fsep []` are equal to `empty`, and properties 7–11 and 16–30 are pruned away. As a result, the number of properties discovered by RoughSpec goes down from 46 to 26.

4.3 Binary Search Trees

In the binary search tree example of Section 3.2, RoughSpec generated two redundant properties:

```
6. toList (delete x nil) = []
7. toList nil = sort []
```

These properties followed from the other discovered properties and the laws `delete x nil = nil` and `sort [] = []`, but these last two laws were not an instance of any of our templates so were not discovered by RoughSpec. This is the same problem that we encountered with the pretty-printing example of Section 3.1.

If we run our hybrid tool with QuickSpec up to term size 3, QuickSpec discovers, among other laws, that `sort [] = []` and `delete x nil = nil`, allowing RoughSpec to prune away its redundant properties. RoughSpec then discovers four properties: the first five from Section 3.2 minus `toList nil = []`, which is now discovered by QuickSpec. (It would be useful to report that this last property matches one of the user-supplied templates, but this is future work.) RoughSpec's runtime goes up from 0.3 to 0.8 seconds, which still compares favourably to the 11 seconds needed by QuickSpec. Here we pay a small price in runtime to obtain better quality output.

4.4 List library

We performed two experiments using the combined tool on the large list library from Section 3.3, allowing QuickSpec to explore terms of size up to 2 and 3 respectively.

When we ran QuickSpec up to term size 2, QuickSpec discovered 7 properties, some of which would otherwise have been discovered by RoughSpec but a few of which RoughSpec had previously missed, for instance `length [] = 0`, `sum [] = 0` (as those don't fit any of our templates), and `concat [] = []` (which RoughSpec's fix-point template does not discover due to the empty list having a different type on the left hand side of the equation than the right hand side). These properties are useful for pruning away some of the redundant properties discovered by RoughSpec, such as `length [] + x = x`. The hybrid tool now discovers 177 properties instead of the previous 184, with 4 additional properties being discovered by QuickSpec instead, and 3 being pruned away. However, even running QuickSpec up to a small term size takes a significant amount of time on this set of functions due to the amount of terms of different types QuickSpec generates and enumerates, and our runtime went from 7.3 minutes to 8.6 minutes.

When we ran QuickSpec up to term size 3, QuickSpec discovered 43 properties and RoughSpec 146 (compared to the original 184) and the runtime was 8.75 minutes. Since adding QuickSpec to the mix increases the runtime quite a bit and (while QuickSpec also discovers a number of properties RoughSpec would have found anyway), our combination of tools does not provide the improvements we hoped for in this case. It seems that the benefits of using QuickSpec decrease as the number of different functions in the signature increase.

Alternatively, running QuickSpec only on the background functions or a limited subset of the library functions could be more helpful to make our pruner smarter without taking too much time.

4.5 Window manager

When we run our hybrid tool with QuickSpec set up to term size 3, QuickSpec discovers two properties:

```
1. rotate EQ s = s
2. current (empty x) = 0
```

This allows RoughSpec to prune away the less elegant properties `current (empty 0) = 0` and `current (empty x) + y = y` (property 1 would have been discovered by RoughSpec anyway), otherwise producing the same output as previously and taking 7 seconds instead of the previous 6.5. When we add the `invariant` function as described in Section 3.4, QuickSpec also discovers the property `invariant s`, allowing RoughSpec to prune away the equivalent but less nice `invariant s && x = x`.

4.6 Summary

Combining QuickSpec and RoughSpec reduces the number of templates needed to discover small, basic properties. These properties are often useful for pruning away other, larger (and to a human, less elegant) properties. What's more, the combination solves the problem we repeatedly encountered in Section 3, where RoughSpec produced redundant laws as a result of failing to notice that two terms are equal. There is often a small overhead in runtime compared to using RoughSpec on its own, which is to be expected, as the search space is bigger when we allow QuickSpec to explore small terms. The larger the signature is, the bigger this overhead is. Still, the combination of QuickSpec and RoughSpec often produces higher-quality laws than either tool manages on its own.

5 RELATED WORK

Apart from QuickSpec [18] and Speculate [1], which we described in the Introduction, there are also theory exploration tools for mathematics [13–15]. Below we describe several which support templates or schemas:

- Buchberger [2] introduced the idea of schema-based theory exploration and his team implemented it in the Theorema [3] system. Theorema provides tools to assist the user in their theory exploration but does not automate the process. The user must provide the schemas (but can store them in a schema library for easier reuse), manually perform substitutions to instantiate the schemas with terms, and conduct proofs interactively.
- IsaScheme [15] is a schema-based theory exploration system for Isabelle/HOL. Users provide the schemas as well as a set of terms to instantiate the schemas with, but the instantiation is performed automatically. The conjectures generated by instantiation are then automatically refuted using Isabelle/HOL's counter-example finders, or proved using the IsaPlanner [5, 6] prover.
- MATHsAiD [14] is an automated theorem-discovery tool which has mainly been applied in the context of abstract algebra. It uses a combination of several exploration techniques, one of them being schema instantiation, which is used for a limited set of lemmas/theorems. The schemas used by MATHsAiD are predefined and built-in to the system and include, for example, reflexivity and transitivity.

None of the above systems has the same flexibility in combining scheme-based and free-form equational exploration as our system. We allow the user to customise the templates used, as well as tune exploration to decide which part of the search space should be explored thoroughly by QuickSpec, and which parts should be explored by a template-based strategy.

Very recently, there has been interest in applying neural networks and techniques developed for natural language processing also for tasks in automated theorem proving. Rabe et al. [16], present an experiment using a language model for various mathematical reasoning tasks, including the generation of new conjectures. They evaluate their tool on tasks such as generating a missing precondition for a given conditional statement, generating the opposite side of an equation, given one side, as well as "free-form" conjecturing. While the model does produce some true statements, the majority these are not new, but for instance exact copies or alpha renamings of statements from the training set. This is of course a very different approach compared to our work, as our tool is designed to produce only statements that are apparently true (by means of testing), and only statements that are *different* from what has been previously seen, ensured by including simple equational reasoning in the generation process.

6 FUTURE WORK

There are many avenues of future work we would like to explore.

RoughSpec currently supports only equations as templates. We would like to allow templates to be arbitrary formulas, such as a template for monotonicity, `X <= Y ==> ?F X <= ?F Y`. There is no deep reason why RoughSpec only supports equations—only that it re-uses code from QuickSpec [18], which itself only supports equations. We plan to lift this restriction, which is not hard but requires a bit of engineering.

We are currently extending RoughSpec to discover conditional equations. In our approach, the user specifies a set of equational templates and a set of condition templates, and the tool discovers which combinations of conditions make each equation true. That is, the user does not have to know the exact condition for each template, but only which conditions are interesting. The main idea is that given a candidate equation and a large set of candidate conditions, we can test if *all conditions together* imply the equation; if they do, we can remove conditions one by one and thus obtain a minimal set of conditions. An important question would be whether we can supply a useful set of "standard" condition templates, as we do for equations.

In the experiments described in this paper we have used handwritten templates provided by the user or by a library of default templates. We would like to further explore using data-driven methods to learn good templates for a given context. We will investigate using machine learning to extract common patterns from proof libraries, learning common lemma shapes given properties of the theorem we want to prove (c.f. [7]), as well as exploiting type-class laws and other algebraic properties.

QuickSpec has been used to discover lemmas in a theorem proving context [12], and we believe our extension could also be useful here, using templates relevant for the theorem we would like to prove. We will also investigate extracting templates from failed proof attempts, similar to critics in proof planning [11].

We currently use a set of heuristics to expand templates. Template expansion is important in order to captures a wide variety of laws, but it sometimes goes too far. For example, given the template `?F (?G X) = ?G (?F X)`, both `?F` and `?G` can be replaced by a nested function, resulting in laws of the form `f (g (h (i x))) = h (i (f (g x)))`. To reduce the use of heuristics, we would like to define an *expressive* template language, in which the user can say precisely what sort of laws they want, for example, to forbid the use of nested functions in the template above, or alternatively allow for expansion as iterative deepening, up to a specified limit. As another example, it should be possible to define a template that captures a general distributivity law `f (g x1) (g x2)...(g xn) = g (f x1...xn)` for n-ary functions, without specialising it to a particular n. Doing so requires designing a small set of combinators for building templates.

Our tool could be made more user-friendly by not requiring the user to explicitly type up a signature. A default signature for a given set of functions could be automatically generated using Template Haskell.

7 CONCLUSION

We have presented RoughSpec, a theory exploration tool in which the user specifies which kinds of properties are interesting. It generates specifications which are short, and easy to understand, but not necessarily complete. It can be used both to produce a rough specification of how a set of functions behaves, and to target specific families of laws that the user is interested in. It also scales well to large APIs. It is complementary to QuickSpec, which uses a complete strategy for conjecture generation.

By combining the two tools, we get the benefits of both: QuickSpec deals with smaller conjectures (of which there are fewer), while RoughSpec handles larger conjectures, resulting in a system that is both fast and outputs a manageable number of largely interesting properties.

ACKNOWLEDGMENTS

This work was partially supported by the Wallenberg Artificial Intelligence, Autonomous Systems and Software Program (WASP), funded by the Knut and Alice Wallenberg Foundation, and by the Swedish Research Council (VR) grants 2016-06204, *Systematic testing of cyber-physical systems (SyTeC)* and 2014-04849, *Learning and exploration in automated reasoning*.

REFERENCES

[1] Rudy Braquehais and Colin Runciman. 2017. Speculate: discovering conditional equations and inequalities about black-box functions by reasoning from test results. In *Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell*. 40–51.
[2] Bruno Buchberger. 2000. Theory exploration with Theorema. *Analele Universitatii Din Timisoara, ser. Matematica-Informatica* 38, 2 (2000), 9–32.
[3] Bruno Buchberger, Adrian Craciun, Tudor Jebelean, Laura Kovács, Temur Kutsia, Koji Nakagawa, Florina Piroi, Nikolaj Popov, Judit Robu, Markus Rosenkranz, and Wolfgang Windsteiger. 2006. Theorema: Towards computer-aided mathematical theory exploration. *Journal of Applied Logic* 4 (12 2006), 470–504. https://doi.org/10.1016/j.jal.2005.10.006
[4] Koen Claessen and John Hughes. 2000. QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proceedings of ICFP*. 268–279.
[5] Lucas Dixon and Jacques D. Fleuriot. 2003. IsaPlanner: A Prototype Proof Planner in Isabelle. *LNCS (LNAI)* 2741, 279–283. https://doi.org/10.1007/978-3-540-45085-6_22
[6] Lucas Dixon and Moa Johansson. 2007. IsaPlanner 2: A Proof Planner for Isabelle.
[7] Jonathan Heras, Ekaterina Komendantskaya, Moa Johansson, and Ewen Maclean. 2013. Proof-Pattern Recognition and Lemma Discovery in ACL2. In *Proceedings of LPAR*. https://doi.org/10.1007/978-3-642-45221-5_27
[8] C. A. R. Hoare. 1976. Proof of correctness of data representations. In *Language Hierarchies and Interfaces*, Friedrich L. Bauer, E. W. Dijkstra, A. Ershov, M. Griffiths, C. A. R. Hoare, W. A. Wulf, and Klaus Samelson (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 183–193.
[9] John Hughes. 1995. The Design of a Pretty-printing Library. In *Advanced Functional Programming*, J. Jeuring and E. Meijer (Eds.). Springer Verlag, LNCS 925, 53–96.
[10] John Hughes. 2020. How to Specify It. In *Trends in Functional Programming*, William J. Bowman and Ronald Garcia (Eds.). Springer International Publishing, Cham, 58–83.
[11] Andrew Ireland and Alan Bundy. 1996. Productive Use of Failure in Inductive Proof. *Journal of Automated Reasoning* 16 (1996), 79–111.
[12] Moa Johansson. 2017. Automated theory exploration for interactive theorem proving: An introduction to the Hipster system. In *Proceedings of ITP (LNCS, Vol. 10499)*. Springer, 1–11.
[13] Moa Johansson, Lucas Dixon, and Alan Bundy. 2011. Conjecture Synthesis for Inductive Theories. *Journal of Automated Reasoning* 47, 3 (01 Oct 2011), 251–289. https://doi.org/10.1007/s10817-010-9193-y
[14] R. L. McCasland, A. Bundy, and P. F. Smith. 2017. MATHsAiD: Automated mathematical theory exploration. *Applied Intelligence* (23 Jun 2017). https://doi.org/10.1007/s10489-017-0954-8
[15] Omar Montano-Rivas, Roy McCasland, Lucas Dixon, and Alan Bundy. 2012. Scheme-based theorem discovery and concept invention. *Expert systems with applications* 39, 2 (2012), 1637–1646.
[16] Markus N. Rabe, Dennis Lee, Kshitij Bansal, and Christian Szegedy. 2020. Mathematical Reasoning via Self-supervised Skip-tree Training. arXiv:2006.04757 [cs.LG]
[17] Colin Runciman, Matthew Naylor, and Fredrik Lindblad. 2008. SmallCheck and Lazy SmallCheck: automatic exhaustive testing for small values. In *Proceedings of the first ACM SIGPLAN symposium on Haskell*. 37–48.
[18] Nicholas Smallbone, Moa Johansson, Koen Claessen, and Maximilian Algehed. 2017. Quick specifications for the busy programmer. *Journal of Functional Programming* 27 (2017). https://doi.org/10.1017/S0956796817000090
[19] Don Stewart. 2007. Roll Your Own Window Manager: Part 1: Defining and Testing a Model. https://donsbot.wordpress.com/2007/05/01/roll-your-own-window-manager-part-1-defining-and-testing-a-model/
[20] Don Stewart and Spencer Sjanssen. 2007. Xmonad. In *Proceedings of the ACM SIGPLAN Workshop on Haskell Workshop* (Freiburg, Germany) *(Haskell '07)*. Association for Computing Machinery, New York, NY, USA, 119. https://doi.org/10.1145/1291201.1291218