Under consideration for publication in J. Functional Programming

Quick Specifications for the Busy Programmer

NICHOLAS SMALLBONE, MOA JOHANSSON, KOEN CLAESSEN and MAXIMILIAN ALGEHED
Chalmers University of Technology
(*e-mail:* {nicsma,moa.johansson,koen,algehed}@chalmers.se)

Abstract

QuickSpec is a theory exploration system which tests a Haskell program to find equational properties of it, automatically. The equations can be used to help understand the program, or as lemmas to help prove the program correct. QuickSpec is largely automatic: the user just supplies the functions to be tested and QuickCheck data generators.

Previous theory exploration systems, including earlier versions of QuickSpec itself, scaled poorly. This paper describes a new architecture for theory exploration with which we can find vastly more complex laws than before, and much faster. We demonstrate theory exploration in QuickSpec on problems both from functional programming and mathematics.

1 Introduction

Formal specifications are a powerful tool for understanding programs. For example, if we know that $\mathit{filter}\ p\ (xs +\!\!+ ys) = \mathit{filter}\ p\ xs +\!\!+ \mathit{filter}\ p\ ys$, we know that *filter* acts pointwise on its input: to understand what *filter* does, we need only understand what it does to singleton lists. When a library comes with a specification, the user need not guess how the library behaves: they can work it out for themselves.

A specification also helps the designer of a library. Formal specifications are unforgiving of irregular behaviour: if the code has special cases, the specification will have special cases. Implicit assumptions about how the library is used must be explicitly spelled out in the specification. A clean, clear specification is a good sign that the design of the library is on the right track. And of course, we can use an automated testing tool like QuickCheck (Claessen & Hughes, 2000) to check the library against its specification.

But it is *hard* and *time-consuming* to write a specification. Properties are different from programs, and writing them needs a different mindset; it is easy not to bother, especially if the code has to be delivered to a tight deadline. The result is code that behaves in unexpected ways, and libraries that we have to decipher through trial and error.

How can we lower the entry barrier for formal specification? In this paper, we show how to *discover a specification*, automatically, simply by testing the program. Our tool, QuickSpec, discovers *equations* between the functions in the program. It does so efficiently, taking seconds on smaller inputs or minutes on larger ones; for example, it discovers the above *filter* law in a fraction of a second. You can study the discovered equations to understand the program better, or use them as the basis for a full specification.

Example As a first example, illustrating both the strengths and limits of QuickSpec, let's use it to discover the monad laws. To run QuickSpec, we have to provide it with a set of functions to explore, which we call the *signature*.[1] QuickSpec understands polymorphic functions, but not typeclass polymorphism, so we have to specialise the monad functions to a specific monad, say lists. We give QuickSpec the functions *return* and $\gg\!=$, and it immediately prints:

$$xs \gg\!= return = xs \tag{1}$$

$$return\ x \gg\!= f = f\ x \tag{2}$$

These are two of the monad laws. However, the third monad law, $(xs \gg\!= f) \gg\!= g = xs \gg\!= (\lambda x.\ f x \gg\!= g)$, is missing. Why?

Note that this law contains a λ-term. QuickSpec only builds equations from the functions it is given; it does not try to synthesise new functions, and in particular it does not generate λ-terms. To characterise monads without needing λ-terms, we can use the *Kleisli composition* operator, here specialised to lists: $(\gg\!\!>) :: (a \to [b]) \to (b \to [c]) \to (a \to [c])$.

With $(\gg\!\!>)$ added to the signature, QuickSpec finds five additional laws:

$$return \gg\!\!> f = f \tag{3}$$

$$f \gg\!\!> return = f \tag{4}$$

$$(f \gg\!\!> g)\ x = f\ x \gg\!= g \tag{5}$$

$$(f \gg\!\!> g) \gg\!\!> h = f \gg\!\!> (g \gg\!\!> h) \tag{6}$$

$$(xs \gg\!= f) \gg\!= g = xs \gg\!= (f \gg\!\!> g) \tag{7}$$

Laws 3 and 4 state that $\gg\!\!>$ has unit *return*, law 5 defines $\gg\!\!>$ in terms of $\gg\!=$ and law 6 state that $\gg\!\!>$ is associative. Finally, equation 7 is the third monad law!

The whole process takes about $1\frac{1}{2}$ seconds from start to finish, and the bulk of the time is spent testing these laws with QuickCheck, i.e. there is very little overhead discovering these laws compared to just testing them.

What use is QuickSpec? In this paper, we are interested in QuickSpec as a standalone tool: the programmer runs it, and sees the equations that are printed out. The laws discovered by QuickSpec have several practical uses:

- *Confirming/denying expectations.* A programmer can check the relatively small set of equations discovered by QuickSpec to ensure that the specification is what they really expected and intended. If not, this could indicate a bug.

[1] For the benefit of the curious reader, the example on this page uses the following signature:

```
signature {
  constants = [
    constant "return" (return :: A → [A]),
    constant ">>=" ((>>=) :: [A] → (A → [B]) → [B]),
    constant ">=>" ((>=>) :: [A] → (A → [B]) → [B])] }
```

The signature is an ordinary Haskell expression which gives the names and values of all the functions QuickSpec should explore. In signatures, *A* and *B* represent type variables.
For user-defined datatypes, we also have to supply QuickCheck generators for random data. These are small functions that QuickCheck uses to produce random values of a given type.

- *Regression testing.* Properties which look sensible can be added as regression tests; if the program's behaviour changes in the future then these properties might fail.
- *Refinement.* Similarly, properties can be discovered from a model implementation and then used to test an optimised implementation.
- *Understanding third-party code.* For poorly-documented third-party libraries, where the source code is perhaps not even available, the discovered laws can give the user an insight into how the library behaves.

QuickSpec can also be used as part of a program verification system or proof assistant. One challenge when proving properties of functional programs is finding lemmas which can be proved by induction; this is difficult to do automatically. QuickSpec is used as a backend by the state-of-the-art inductive theorem provers HipSpec and Hipster (Claessen *et al.*, 2013; Johansson *et al.*, 2014), where the conjectures discovered are piped through to the prover. Once proved, they are used as lemmas when the theorem prover tackles harder problems.

Finally, QuickSpec is an example of a *theory exploration system*. In mathematics, after defining the basic operators and concepts of a new structure, the mathematician has to come up with all the appropriate basic conjectures about that structure before they can proceed to more advanced theorems. Theory exploration systems (Buchberger *et al.*, 2006; Johansson *et al.*, 2011; Montano-Rivas *et al.*, 2012) take a mathematical structure and try to discover as many interesting lemmas about it as possible. These are then given to the mathematician as inspiration for more complex theorems.

QuickSpec is therefore not just a tool for programmers, but can also be used as a theory exploration system for mathematics and theorem proving. In this paper, we concentrate on QuickSpec itself and not its other applications.

Contributions. In principle, a theory exploration system such as QuickSpec simply generates and checks a large set of conjectures, built somehow from the functions and constants in the theory. As there is no way of knowing in advance which conjectures might be true, it should consider all possible conjectures up to some resource limit.

The reader might find it hard to believe that this could possibly scale beyond small examples. Indeed, finding interesting properties or lemmas about a program or mathematical structure is a very difficult problem, which suffers inherently from exponential blowup. Existing theory exploration systems, including earlier versions of QuickSpec (Claessen *et al.*, 2010), work only on small examples.

Our paper presents a new, efficient, architecture for theory exploration, which suffers far less from exponential blowup than existing designs. It often does less work by a factor of hundreds or thousands than older theory exploration systems. The new architecture scales well, discovering complex equational laws over large signatures of functions. While earlier theory exploration systems were limited to toy problems, this new incarnation of QuickSpec is a capable tool for finding equational specifications for real programs and libraries. We start by demonstrating its power on a pretty-printing library (Section 2); we then describe QuickSpec's design in detail (Section 3), followed by a variety of case studies (Section 4).

QuickSpec is open source and available from:

`https://github.com/nick8325/quickspec`.

2 Pretty-printing

In his classic paper, Hughes (Hughes, 1995) uses equational reasoning to design an efficient pretty-printing library. In this section, we will take this library as an initial example of how QuickSpec can be used. The library defines a type *Doc* of *documents*, and four combinators which build documents:

type $Doc = ...$	-- the abstract type of documents
$(\diamond) :: Doc \to Doc \to Doc$	-- horizontal composition, $x \diamond y$
$(\$\$) :: Doc \to Doc \to Doc$	-- vertical composition, $x \ \$\$ \ y$
$nest :: Int \to Doc \to Doc$	-- indentation, $nest\ k\ x$
$text :: String \to Doc$	-- plain text, $text\ xs$

The four combinators behave as follows:

- Horizontal composition: $x \diamond y$ sets x next to y on the page.
- Vertical composition: $x \ \$\$ \ y$ sets x above y on the page.
- Indentation: $nest\ k\ x$ indents x by k spaces.
- Plain text: $text\ xs$ converts a string xs into a document.

Hughes starts with a denotational semantics of his combinators, in the form of a model implementation, which he assumes to be correct. He lists laws that hold in his semantics, then uses those laws to derive an efficient version of the combinators.

The laws themselves appear out of nowhere, and Hughes does not explain how he thought of them. One of the laws in particular is rather intimidating:

$$text\ xs \diamond ((text\ \texttt{""} \diamond x) \ \$\$ \ y) = (text\ xs \diamond x) \ \$\$ \ nest\ (length\ xs)\ y$$

In this section we put ourselves in Hughes's shoes: from a nice model implementation of the pretty-printing combinators, work out what laws it satisfies. However, instead of doing it ourselves by hand, we will let QuickSpec help us. In all examples, we give all equations output by QuickSpec in full, without omissions, unless otherwise stated.

As before, we need to define a signature, which will consist of all four pretty-printing combinators. We run QuickSpec, and in a couple of seconds it prints the following equations:

$$(x \diamond y) \diamond z = x \diamond (y \diamond z) \tag{1}$$

$$(x \ \$\$ \ y) \ \$\$ \ z = x \ \$\$ \ (y \ \$\$ \ z) \tag{2}$$

$$(x \ \$\$ \ y) \diamond z = x \ \$\$ \ (y \diamond z) \tag{3}$$

$$x \diamond nest\ i\ y = x \diamond y \tag{4}$$

$$nest\ i\ (x \diamond y) = nest\ i\ x \diamond y \tag{5}$$

$$nest\ i\ x \ \$\$ \ nest\ i\ y = nest\ i\ (x \ \$\$ \ y) \tag{6}$$

$$nest\ i\ (nest\ j\ x) = nest\ j\ (nest\ i\ x) \tag{$*$}$$

These laws reveal a number of properties of the pretty-printing combinators:

- Both horizontal and vertical composition are associative (laws 1 and 2)—in other words, we can lay out a *sequence* of documents horizontally or vertically, without worrying about bracketing.

- In a horizontal composition $x \diamond y$, if x is a multi-line document constructed using $\$\$$, then y is appended to the *last* line of x (law 3).
- In a horizontal composition $x \diamond y$, the indentation of y is ignored (law 4): only the indentation of x matters (law 5).
- Nesting a multi-line document indents every line in it (law 6).

The final starred equation states that two nested *nest*s commute. All very well and good, but is there more we can say than this? Presumably, indenting a document by i and then by j has the total effect of indenting it by $i+j$, but we don't get a law to this effect. This is because QuickSpec only discovers laws about functions in the signature, and $+$ is not in our signature!

We add $+$ to the signature, and 0 for good measure. We can even mark them as *background functions*—QuickSpec will only print a law if it involves at least one non-background function, so we won't get laws about just $+$ and 0. Having done this, QuickSpec finds two more laws:

$$\mathit{nest}\ 0\ x = x \tag{7}$$

$$\mathit{nest}\ (i+j)\ x = \mathit{nest}\ i\ (\mathit{nest}\ j\ x) \tag{8}$$

What's more, it no longer prints the starred law above since it follows from law 8. Inspired by our success, we also add $+\!\!+$ and `""`. We get two laws about the *text* combinator:

$$x \diamond \mathit{text}\ \texttt{""} = x \tag{9}$$

$$\mathit{text}\ xs \diamond \mathit{text}\ ys = \mathit{text}\ (xs +\!\!+ ys) \tag{10}$$

We also get three rather more complicated-looking laws.

$$\mathit{text}\ \texttt{""} \diamond (\mathit{text}\ xs\ \$\$\ x) = \mathit{text}\ xs\ \$\$\ x \tag{a}$$

$$\mathit{text}\ \texttt{""} \diamond ((\mathit{text}\ xs \diamond x)\ \$\$\ y) = (\mathit{text}\ xs \diamond x)\ \$\$\ y \tag{b}$$

$$(\mathit{text}\ \texttt{""} \diamond x)\ \$\$\ (\mathit{text}\ \texttt{""}\ \$\$\ x) = \mathit{text}\ \texttt{""} \diamond (x\ \$\$\ x) \tag{$*$}$$

These laws are curious because they all involve terms of the form $\mathit{text}\ \texttt{""} \diamond x$. Isn't this the same as x? The law $\mathit{text}\ \texttt{""} \diamond x = x$ is conspicuous by its absence from the list above, so we can QuickCheck it, which reveals a counterexample:

$\mathit{nest}\ 2\ (\mathit{text}\ \texttt{"ab"})$ renders to `"  ab"`, but
$\mathit{text}\ \texttt{""} \diamond \mathit{nest}\ 2\ (\mathit{text}\ \texttt{"ab"})$ renders to `"ab"`.

It seems that $\mathit{text}\ \texttt{""} \diamond x$ strips any indentation from x, so $\mathit{text}\ \texttt{""} \diamond x = x$ exactly when x is not indented. We therefore see from the first two laws above that any document whose first line starts with a *text* is not indented. This suggests that the indentation of a document as a whole is determined by how its first line is indented.

The third, starred, law is slightly different in that it has *text* `""` on *both* sides. Could we generalise this law to an arbitrary string, *text xs*? The correct generalisation certainly isn't

$$(\mathit{text}\ xs \diamond x)\ \$\$\ (\mathit{text}\ xs\ \$\$\ x) = \mathit{text}\ xs \diamond (x\ \$\$\ x)$$

because the left-hand side will print the string xs twice. Nor is it

$$(\mathit{text}\ xs \diamond x)\ \$\$\ (\mathit{text}\ \texttt{""}\ \$\$\ x) = \mathit{text}\ xs \diamond (x\ \$\$\ x)$$

because on the left-hand side, the second x won't be indented, while on the right-hand side it will be. We need to indent the second x by the length of xs, and in hindsight it is obvious that we should also have *length* in our signature—clearly, changing the length of a string will affect how a document is typeset (by, for example, moving other documents set next to it). Adding the standard Haskell function $length :: String \to Int$ to the signature, we get a generalisation of the starred law,

$$(text\ xs \diamond x)\ \$\$\ (text\ "" \diamond x) = text\ xs \diamond (nest\ (length\ xs)\ x\ \$\$\ x) \tag{c}$$

in which the second x on the right-hand side is effectively unindented. We also get three more laws, including Hughes's scary law from the start of this section!

$$text\ xs \diamond (text\ ""\ \$\$\ x) = text\ xs\ \$\$\ nest\ (length\ xs)\ x \tag{d}$$

$$text\ (xs +\!\!+ ys)\ \$\$\ nest\ (length\ xs)\ x = text\ xs \diamond (text\ ys\ \$\$\ x) \tag{e}$$

$$text\ xs \diamond ((text\ "" \diamond x)\ \$\$\ y) = (text\ xs \diamond x)\ \$\$\ nest\ (length\ xs)\ y \tag{11}$$

This means that QuickSpec has succeeded in finding all 11 of Hughes's laws (the 11 numbered laws in this section); all we had to do was give it the right auxiliary functions to explore. It also finds five extra ones, which are the ones labelled (a)–(e). The complete set of laws is listed in Figure 1; the five which are not in (Hughes, 1995) are marked with a star.

$$\begin{aligned}
nest\ 0\ x &= x \\
x \diamond text\ "" &= x \\
(x\ \$\$\ y)\ \$\$\ z &= x\ \$\$\ (y\ \$\$\ z) \\
x \diamond nest\ i\ y &= x \diamond y \\
(x\ \$\$\ y) \diamond z &= x\ \$\$\ (y \diamond z) \\
(x \diamond y) \diamond z &= x \diamond (y \diamond z) \\
nest\ i\ (x \diamond y) &= nest\ i\ x \diamond y \\
nest\ (i+j)\ x &= nest\ i\ (nest\ j\ x) \\
text\ xs \diamond text\ ys &= text\ (xs +\!\!+ ys) \\
nest\ i\ x\ \$\$\ nest\ i\ y &= nest\ i\ (x\ \$\$\ y) \\
text\ ""\ \diamond (text\ xs\ \$\$\ x) &= text\ xs\ \$\$\ x && (*) \\
text\ xs \diamond (text\ ""\ \$\$\ x) &= text\ xs\ \$\$\ nest\ (length\ xs)\ x && (*) \\
text\ (xs +\!\!+ ys)\ \$\$\ nest\ (length\ xs)\ x &= text\ xs \diamond (text\ ys\ \$\$\ x) && (*) \\
(text\ xs \diamond x)\ \$\$\ (text\ "" \diamond x) &= text\ xs \diamond (nest\ (length\ xs)\ x\ \$\$\ x) && (*) \\
text\ ""\ \diamond ((text\ xs \diamond x)\ \$\$\ y) &= (text\ xs \diamond x)\ \$\$\ y && (*) \\
text\ xs \diamond ((text\ ""\ \diamond x)\ \$\$\ y) &= (text\ xs \diamond x)\ \$\$\ nest\ (length\ xs)\ y
\end{aligned}$$

Fig. 1: What QuickSpec discovers about the pretty-printing library. Starred equations are ones not found in Hughes.

Of these five extra laws, four are special cases of the final law. QuickSpec prints laws as it discovers them, and explores laws in order of size; as the final law is quite big, and the

special cases smaller, we get those special cases. This is not so bad: small special cases of a large general law are often quite instructive. The fifth law,

$$(\mathit{text}\ \mathit{xs} \diamond x)\ \$\$\ (\mathit{text}\ \texttt{""} \diamond x) = \mathit{text}\ \mathit{xs} \diamond (\mathit{nest}\ (\mathit{length}\ \mathit{xs})\ x\ \$\$\ x)$$

does *not follow* from Hughes's 11: his laws are complete for ground documents, but not documents with variables.

This law is not very useful: it talks about typesetting x above itself, hardly a common operation. We can generalise it by observing that we do not really need to have the *same* document twice, but two documents which are indented by the same amount. We therefore introduce a new concept, the *nesting level* of a document. Using our earlier idea that a document x is unindented if $\mathit{text}\ \texttt{""} \diamond x = x$, we say that the nesting level of x is k if $\mathit{nest}\ (-k)\ x$ is unindented. We define a function *nesting* which takes a document and returns its nesting level; given *nesting*, QuickSpec discovers six new laws. The first four show that the four combinators affect the nesting level as we would expect:

$$\begin{aligned}
\mathit{nesting}\ (\mathit{text}\ \mathit{xs}) &= 0 \\
\mathit{nesting}\ (x\ \$\$\ y) &= \mathit{nesting}\ x \\
\mathit{nesting}\ (x \diamond y) &= \mathit{nesting}\ x \\
\mathit{nesting}\ (\mathit{nest}\ i\ x) &= i + \mathit{nesting}\ x
\end{aligned}$$

while the fifth one shows that our definition of *nesting* makes sense, as unindenting a document and then indenting it by its nesting level recovers the original document:

$$\mathit{nest}\ (\mathit{nesting}\ x)\ (\mathit{text}\ \texttt{""}) \diamond x = x$$

Finally, Hughes's 11th law is gone! Instead, it is replaced by the following reformulation, which we think is simpler as it doesn't rely on using *text* "" to unindent x, instead explicitly adjusting the nesting of y by the correct amount:

$$\mathit{text}\ \mathit{xs} \diamond (x\ \$\$\ \mathit{nest}\ (\mathit{nesting}\ x)\ y) = (\mathit{text}\ \mathit{xs} \diamond x)\ \$\$\ \mathit{nest}\ (\mathit{length}\ \mathit{xs})\ y$$

From these equations about *nesting*, one of the starred laws from Figure 1:
$(\mathit{text}\ \mathit{xs} \diamond x)\ \$\$\ (\mathit{text}\ \texttt{""} \diamond x) = \mathit{text}\ \mathit{xs} \diamond (\mathit{nest}\ (\mathit{length}\ \mathit{xs})\ x\ \$\$\ x)$ also follows, so QuickSpec can now discard it. We conjecture that this final set of laws is a complete equational specification for Hughes's pretty-printing library—Hughes's specification is complete only for ground documents.

Summing up. QuickSpec was able to find all of Hughes's laws, and with a bit more thought on our part was even able to generalise them, letting us use the concept of nesting level to reason about non-ground documents. All we had to do was supply the right signature. The skill in using QuickSpec lies in knowing which functions to include in the signature, including auxiliary functions, and sometimes which to leave out. By looking at unexpectedly specific laws, and thinking how to generalise them, we can often come up with new auxiliary functions which will give us better laws.

For the complete pretty-printing example, including *nesting*, running QuickSpec takes about ten seconds. Previous theory exploration systems cannot handle such a complex set of laws.

3 Implementing a theory exploration system

In the previous section, we got a taste of what QuickSpec can do. Now it is time to look under the hood and see *how* it does it. Before we dive into QuickSpec's algorithms, we will take a look at a very basic model of theory exploration, to properly understand the problems that QuickSpec is designed to solve.

3.1 Inventing equations slowly

How can we discover equations about a program? One idea is to generate *all* equations up to some size limit, test them (e.g. with QuickCheck) and print out the ones which seem to hold. This idea finds *too many* laws: given the function $+\!\!+$ and constant $[\,]$, it will discover $xs +\!\!+ [\,] = xs$ and $[\,] +\!\!+ xs = xs$, but also $xs +\!\!+ [\,] = [\,] +\!\!+ xs$ and $[\,] +\!\!+ [\,] = [\,]$, which follow from the first two laws. We must therefore *prune* the discovered laws, removing the redundant laws, i.e. the ones which follow from other discovered laws. The resulting design, which we call SlowSpec, is shown in Figure 2.

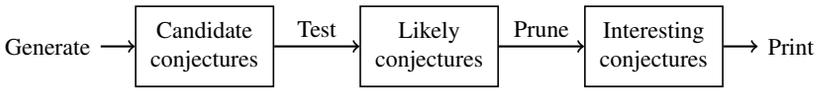


Fig. 2: A simple but slow design for theory exploration.

SlowSpec will find a nice set of laws—eventually. Unfortunately, it is terribly slow, and on non-trivial examples we will run out of time or space before it terminates. To see why, consider the pretty-printing example. The law

$$text\ xs \diamond ((text\ \texttt{""} \diamond x)\ \$\$\ y) = (text\ xs \diamond x)\ \$\$\ nest\ (length\ xs)\ y$$

relates two terms of size 9. There are approximately 10^6 terms of size $\leqslant 9$ over the pretty-printing combinators and three variables. This is too many for SlowSpec, for two reasons:

- *Quadratic blowup.* To find all equations between a set of n terms, SlowSpec will try n^2 conjectures. In other words, the pretty-printing example will need to test 10^{12} conjectures! This is not a reasonable number.
- *Wasted testing.* Suppose that each of the 10^6 terms is on average equal to one other term.[2] Then SlowSpec will generate about $n/2 = 500000$ true conjectures, all of which will be thoroughly tested—but all but perhaps 50 will then be pruned away as redundant. In this particular case, 99.99% of the testing we do is simply wasted time.

In the next section we describe QuickSpec's design, which avoids both these problems:

- The runtime is proportional to the number of explored *terms*, not the number of possible conjectures.
- Most of the testing effort is spent on the final laws that are presented to the user, not conjectures that are falsified or pruned away. The testing effort is in practice proportional to the number of discovered laws.

[2] This number reflects reality quite closely in our experience.

3.2 Inventing equations quickly

QuickSpec uses the same ingredients as SlowSpec—it tests conjectures, prunes out redundant ones, and presents what remains to the user. But it rearranges these ingredients to discover equations much more efficiently. In the following section we describe the three refinements that take us from SlowSpec to QuickSpec:

- We interleave testing and pruning to avoid testing redundant conjectures (section 3.2.1).
- We test *terms* instead of conjectures to avoid quadratic blowup (section 3.2.2).
- We use so-called *schemas* to avoid testing most terms altogether (section 3.2.3).

3.2.1 Interleaving testing and pruning

SlowSpec tests many laws that are later pruned away. For example, given the functions $+\!\!+$, $[\,]$ and *reverse*, it will discover both $xs +\!\!+ [\,] = xs$ and $\mathit{reverse}\ xs +\!\!+ [\,] = \mathit{reverse}\ xs$, and then prune away the second one. In practice, very many pairs of terms are equal, but a small subset of these equalities implies the rest, and the overwhelming majority (well over 99%) are redundant.

Our first improvement is to *not test redundant laws*. To do so, we will test each conjecture before generating the next one. This allows us to prune a conjecture before we test it, if it follows from what we have already discovered.

As before, we enumerate all equations up to some size limit. For each equation, we do the following:

- If the equation follows by equational reasoning from the equations we have already discovered, it is redundant and we discard it.
- Otherwise, we test the equation. If it is false, we discard it. If it seems to be true, we print it out. It can then be used to help prove later equations.

With this design, after discovering the law $xs +\!\!+ [\,] = xs$, the law $\mathit{reverse}\ xs +\!\!+ [\,] = \mathit{reverse}\ xs$ will be discarded without being tested, since it is an instance of the first law. By interleaving testing and pruning, we avoid testing laws that we already know to be true. As false laws are typically falsified after a few cases, but true laws have to be tested fully, this cuts out the vast majority of testing effort.[3]

3.2.2 Enumerating terms instead of equations

The design so far still suffers from the quadratic blowup of section 3.1: if we want to discover all equations between a set of n terms, we need to generate n^2 conjectures. Even though almost all will be false and discarded quickly, generating them still uses time and memory.

To solve this problem, instead of enumerating *conjectures* we will enumerate *terms*. For each term, we will try to discover an equation that relates it to a *previously-explored* term. We call this *considering* the term.

[3] It is hard to quantify exactly how much, but Section 4.5 suggests that a factor of 1000 may be common.

Term	**Status**	**Result**
$[\,]$	Not equal to a previous term	Added to candidate terms
xs	Not equal to a previous term	Added to candidate terms
ys	Not equal to a previous term	Added to candidate terms
$xs +\!\!+ [\,]$	Equal to xs by testing	$xs +\!\!+ [\,] = xs$ discovered
$[\,] +\!\!+ xs$	Equal to xs by testing	$[\,] +\!\!+ xs = xs$ discovered
$ys +\!\!+ [\,]$	Equal to ys by existing laws	Discarded
$[\,] +\!\!+ [\,]$	Equal to $[\,]$ by existing laws	Discarded
$xs +\!\!+ ys$	Not equal to a previous term	Added to candidate terms
$xs +\!\!+ (ys +\!\!+ zs)$	Not equal to a previous term	Added to candidate terms
$xs +\!\!+ (xs +\!\!+ xs)$	Not equal to a previous term	Added to candidate terms
$[\,] +\!\!+ (xs +\!\!+ ys)$	Equal to $xs +\!\!+ ys$ by existing laws	Discarded
$(xs +\!\!+ ys) +\!\!+ zs$	Equal to $xs +\!\!+ (ys +\!\!+ zs)$ by testing	Associativity discovered
$(xs +\!\!+ xs) +\!\!+ xs$	Equal to $xs +\!\!+ (xs +\!\!+ xs)$ by existing laws	Discarded

Table 1: Exploring the functions $[\,]$ and $+\!\!+$.

We can adapt the design of section 3.2.1 to consider terms instead of equations. We maintain a set of *discovered equations*, which are used for pruning, and a set of *candidate terms*, which each term we consider is checked against.

To consider a new term we do the following (the changes from the previous section are marked in *italics*):

- If we can *prove the term equal to a candidate term*, using the equations discovered so far, it is redundant and we discard it.
- Otherwise, we test the term. If it is *different from all candidate terms*, we add it to the candidate term set. If it seems to be *equal to a candidate term*, we add the equation to the discovered set.[4]

Table 1 shows a term-by-term trace of this algorithm exploring the list functions $+\!\!+$ and $[\,]$ and discovering the laws $xs +\!\!+ [\,] = [\,]$, $[\,] +\!\!+ xs = xs$ and $(xs +\!\!+ ys) +\!\!+ zs = xs +\!\!+ (ys +\!\!+ zs)$.

More formally, we can model *consider* as a function of type $\mathit{Term} \to \mathit{State} \to \mathit{State}$, where the state consists of E, the discovered equations, and T, the candidate terms. The specification of *consider* is then as follows:

$$\mathit{consider}\ t\ (E,T) = \begin{cases} (E,T) & \text{if } t = u \text{ follows from } E \text{ for some } u \in T \\ (E \cup \{t = u\},T) & \text{if } t = u \text{ holds by testing for some } u \in T \\ (E,T \cup \{t\}) & \text{otherwise.} \end{cases}$$

[4] Notice that this algorithm maintains the invariant that no two candidate terms are equal, which means each term can only be equal to one candidate term. In theory a term may *test* equal to two candidate terms, but we will see in Section 3.3 that this is not possible.

A term ordering Notice that the laws we get out depend on the order in which we enumerate terms. If, for example, we had generated the term $[] +\!\!+ []$ before $xs +\!\!+ []$, we would have got the law $[] +\!\!+ [] = []$ as well as $xs +\!\!+ [] = xs$.

In QuickSpec, we prune away a law if it is redundant *with respect to previous laws*—but it may be redundant with respect to *later* laws. It's important to enumerate terms in a sensible order—by generating the terms in Table 1 in a different order, we could have "discovered" the unwanted laws $(xs +\!\!+ xs) +\!\!+ xs = xs +\!\!+ (xs +\!\!+ xs)$ or $[] +\!\!+ (xs +\!\!+ ys) = xs +\!\!+ ys$.

But what is a sensible order? We want to discover more general laws before more specific ones. In particular, we want to generate a law $t = u$ before any of its instances $t\sigma = u\sigma$. So we define an order $<$ on terms with the property that, if σ is not a renaming, then $t < t\sigma$, and then we enumerate terms in that order. It seems that the precise order does not matter much as long as it satisfies this property. QuickSpec defines $<$ as follows:

- If t's size is smaller than u's size, then $t < u$.
  For example, $xs +\!\!+ [] < (xs +\!\!+ ys) +\!\!+ []$.
- Otherwise, the term with the most variable occurrences is smaller.
  For example, $xs +\!\!+ xs < xs +\!\!+ [] < [] +\!\!+ []$.
- Otherwise, the term with the most *distinct* variables is smaller.
  For example, $xs +\!\!+ (ys +\!\!+ zs) < xs +\!\!+ (xs +\!\!+ xs)$.
- If all else fails, we simply compare the terms lexicographically.

Efficiency At this point, we do not seem to have gained anything by enumerating terms. When we consider a new term, we must still compare it against all existing candidate terms. Thus, exploring n terms will still take $O(n^2)$ time.

To fix this, we must efficiently compare each new term against all existing terms *simultaneously*. For testing, we will assume we have a data structure for sets of terms that supports the following operation:

$$\mathit{test} :: \mathit{Term} \to \mathit{Set\ Term} \to \mathit{Result}$$
$$\textbf{data}\ \mathit{Result} = \mathit{EqualTo\ Term} \mid \mathit{Distinct}\ (\mathit{Set\ Term})$$

The idea is that *test* takes a term t and a set of terms T and tests t against all terms in T. If it finds that t is equal to some term $u \in T$ then it returns *EqualTo u*. Otherwise, it inserts t into T and returns $\mathit{Distinct}\,(T \cup \{t\})$. We will see how to implement this in section 3.3.

For pruning, instead of trying to prove equations we will compute a *normal form* for each term, so that to prove an equation we simply normalise both sides and see if they turn out equal. We assume a normalisation function

$$\mathit{norm} :: \mathit{Set\ Equation} \to \mathit{Term} \to \mathit{Term}$$

with the property that if $\mathit{norm}\ E\ t = \mathit{norm}\ E\ u$, then $t = u$ follows from E. Conversely, if $t = u$ follows from E, then we hope that $\mathit{norm}\ E\ t = \mathit{norm}\ E\ u$, but because equational theorem proving is undecidable this does not always hold.[5] We will see how to normalise terms effectively in section 3.4.

[5] When this happens, we get redundant equations that our pruner could not prove. In practice, an equation which is redundant but hard to prove is quite likely to be interesting to the user anyway.

We can now refine the specification of *consider* above into the following code. The comments explain how the code relates to the specification. Notice that we keep the candidate terms T normalised with respect to the discovered equations E, which makes it easy to check if the new term can be proved equal to any candidate term:

```
type State = (Set Equation, Set Term)
consider :: Term → State → State
consider t (equations, terms)
    -- Does t = u follow from E for some u ∈ T?
  | norm equations t ∈ terms = (equations, terms)
  | otherwise =
    -- Does t = u hold by testing for some u ∈ T?
    case test t terms of
      EqualTo u →
        let equations' = equations ∪ {t = u} in
        (equations', map (norm equations') terms)
      Distinct terms' → (equations, terms')
```

Generating fewer terms Though *consider* is now efficient, we still feed it a vast number of terms. We would like to avoid generating most terms at all.

Suppose we know that a term t is equal to a candidate term u. Then there is no point generating any term which has t as a subterm; we will generate the same term with t replaced by u anyway. So, when we build terms, we only build them out of candidate subterms. This reduces the number of generated terms substantially; in the pretty-printing example, it goes down 10^6 to 10^4.

Summary The algorithm we have now described is an efficient theory exploration system, though we will refine it further in the next section.

Why is this algorithm fast? One reason is that we enumerate terms instead of conjectures. Because the number of conjectures grows exponentially with size, exploring n^2 conjectures in $O(n)$ time allows us to effectively find laws of double the size.

The other main reason is that *most terms are cheap to explore*. Normalisation is very cheap, as we shall see in section 3.4, so we discard redundant terms quickly. When a term is not equal to any candidate term, a very small number of test cases (typically 10 or so) suffice to distinguish it from the candidate terms, so this case is also quick.

The only time exploring a term is expensive is *when we discover a new law*—in this case, we must test the law thoroughly, which may take some time. We must also update the data structures used by normalisation, which we will see later is also expensive. But in return we get to print out a law—the user sees progress.

Looking back to the promises we made about QuickSpec in Section 3.1, we see that we have kept both. QuickSpec's runtime is proportional to the number of explored terms, and only the discovered laws are tested thoroughly. We will see in Section 4.5 that the testing effort is indeed roughly proportional to the number of discovered laws.

3.2.3 Enumerating schemas instead of terms

We can reduce the number of terms generated still further. The rough idea is to avoid generating lots of terms that are the same up to variable renaming. Instead of directly discovering a law like

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

we will first discover that there is a law of the shape

$$(?+\!\!+?)+\!\!+? = ? +\!\!+ (?+\!\!+?)$$

where each ? stands for a *variable*, and then work out how to fill in the variables in the most general way. Notice that there is a law of this shape if and only if the law

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

holds. Our idea is to generalise from laws with one variable to laws with many variables.

To do so we enumerate *schemas* instead of terms. A schema is a term in which all variables have been replaced by a hole. For example, the term $(xs +\!\!+ ys) +\!\!+ zs$ has the schema $(?+\!\!+?)+\!\!+?$, while the term $(xs +\!\!+ ys) +\!\!+ [\,]$ has the schema $(?+\!\!+?) +\!\!+ [\,]$. A schema represents all the terms we could build by plugging *variables* into the holes in whatever combination we like; we call these terms the *instances* of the schema.

We consider two instances of each schema:

- The *most general instance* of a schema such as $? +\!\!+ (?+\!\!+?)$ is one where all holes are instantiated with different variables, such as $xs +\!\!+ (ys +\!\!+ zs)$.
- The *one-variable instance* is one where all holes of each type are instantiated with the same variable, such as $xs +\!\!+ (xs +\!\!+ xs)$.

We observe that, if the most general instance of a schema can be pruned away by some discovered law, then all of its instances can be pruned away using the same law. Conversely, if there is a law waiting to be discovered about some instance of a schema, then the *one-variable instance* of that law must also hold: $(xs +\!\!+ ys) +\!\!+ zs = xs +\!\!+ (ys +\!\!+ zs)$ implies $(xs +\!\!+ xs) +\!\!+ xs = xs +\!\!+ (xs +\!\!+ xs)$. If testing the one-variable instance of a schema reveals no laws then we need not test the other instances.

We then add a *schema layer on top* of the previous section's *term layer*. The term layer, as before, considers terms and discovers equations, but the schema layer is in charge of giving it terms to consider. The way it does this is to test each schema and possibly *instantiate* it, which means generating all instances of the schema and giving them to the term layer to consider. In more detail, if the schema has n holes, we fill it in with all possible combinations of n variables, and send the resulting terms to the *consider* function of Section 3.2.2. The goal is that most schemas will not need to be instantiated and the number of terms we consider will be drastically reduced: a single schema with n holes stands for n^n terms.

Our algorithm is quite similar to the previous section's. We enumerate schemas in order of size. We maintain a set of *candidate schemas* analogous to the candidate terms from Section 3.2.2. For each schema:

- We discard the schema if we can prove its *most general* instance equal to either
  - a term we have considered, or

— the most general instance of a candidate schema,

using the equations we have discovered so far.

- Otherwise, we add the schema to the candidate schemas, and then test the *one-variable* instance of the schema against the one-variable instances of all other candidate schemas.
  If this reveals an equation $t = u$, we instantiate both schemas.
  Otherwise, we do not instantiate the schema.

Most commonly, the schema is tested and is not equal to any existing schema. In that case, we are finished with that schema, and just add it to the candidate schemas.

Otherwise, we have discovered a candidate equation between two one-variable instances, such as $(xs +\!\!+ xs) +\!\!+ xs = xs +\!\!+ (xs +\!\!+ xs)$, meaning that testing has not found any values for *xs* falsifying this equation. However, a more general version of the equation might exist. To find out if it does, QuickSpec takes all instances (with "?"s instantiated by variables) of both schemas, and feeds them to the term layer. We generate these instances in order of generality, i.e. the instances with the fewest repeated variables first. This makes the term layer first discover the most general formulation of the law, and then use this law to discard the remaining instances.

Example Let us revisit the pretty printing library from Section 2. Suppose QuickSpec is given the functions $\diamond$, 0 and *nest*. On the first iteration QuickSpec generates schemas of size 1: The trivial schema 0 and the one-hole schemas '?' (one for each type). Of course none of the schemas is equal.

Let us fast-forward to size 3. Here QuickSpec generates (among others) the schema *nest* 0 ?. When we test this schema, we find that it is equal to '?'. As these schemas only have one hole each, we generate the terms *nest* 0 *x* and *x*, and discover the law *nest* $0\ x = x$. From now on, we will not construct any larger schemas that contain *nest* 0 ? as a sub-schema, as such a schema will necessarily be equal to an existing one.

Now let's see what happens at size 5. Here the schemas

$$(?\diamond?)\diamond? \text{ and } ?\diamond(?\diamond?)$$

are generated. They each have three holes of the same type and neither is redundant. To test them, we instantiate all those holes with the same variable, say *x*, obtaining the one-variable instances $(x\diamond x)\diamond x$ and $x\diamond(x\diamond x)$. The two schemas will end up equal after testing. We then feed all instances of the schemas to the term layer, in order of generality, i.e. adding instances with larger number of different variables first. As soon as we have fed the term layer the terms $(x\diamond y)\diamond z$ and $x\diamond(y\diamond z)$, it will discover associativity. We will generate more instances such as $(x\diamond x)\diamond y$, which will either be pruned away by associativity or not be equal to any other term.

Singleton schemas The algorithm above has one problem: it does not find laws such as commutativity where both sides share the *same* schema. A schema such as ?+? will not be equal to any other schema and we will not instantiate it. The flaw in the algorithm described above is that it assumes we can generalise any law $t = u$ from the corresponding law about the one-variable instances of *t*'s and *u*'s schema, but in the case of commutativity this is a trivial law which has the same term $x + x$ on both sides.

To fix this problem, we also need to instantiate any schema that is not equal to any existing schema. To improve performance, we sacrifice completeness and only do this if the schema's size is below some threshold, by default 5.[6] We are then not guaranteed to find a law above this size if both sides of that law have the same schema, but will still find all other laws.

Associative-commutative functions There is also a slight aesthetic problem with the algorithm as it stands. Suppose we explore an associative-commutative function such as $+$. After instantiating the schema $?+?$, we will discover commutativity. We will then instantiate $?+(?+?)$, and discover the law $x+(y+z)=y+(x+z)$. In the presence of commutativity, this law implies associativity: coming to the schema $(?+?)+?$, we will discard it, and not print the associativity law.[7] Both laws are equivalent, but we would rather find the conventional formulation of associativity.

At the same time, we do not want an ad hoc heuristic that treats associative-commutative functions specially. Instead we observe that $(x+y)+z=x+(y+z)$ is a terminating rewrite rule, in the sense that applying it repeatedly left-to-right on a term is bound to terminate, while $x+(y+z)=y+(x+z)$ is not, because we can rewrite $t+(u+v)\rightarrow u+(t+v)\rightarrow t+(u+v)$ forever. We would rather discover terminating rewrite rules than arbitrary equations, as these are more likely to represent sensible simplification laws.

Whenever QuickSpec discovers a new law $t=u$, with t being the newly-explored term, it tries to show that $t=u$ is a terminating rewrite rule.[8] If it is, it prints it. Otherwise, it forgets about the law, and does not add t to the set of explored terms. It then continues. After exploring all terms of the same size as t, it then reconsiders t. If t still cannot be pruned, it emits the law $t=u$. In effect QuickSpec behaves as if we had delayed exploring t until a bit later.

3.3 Decision trees for testing terms

In Section 3.2 we assumed that QuickSpec could efficiently test a single term against a set of other terms. To do this, we maintain a *decision tree* alongside the set of terms. Figure 3 illustrates one such tree; the internal nodes are test cases, whose edges are labelled with test results, and the leaves are terms.

To test a new term against the terms in a decision tree, we try inserting it into the decision tree. That is, we take the test case at the root, evaluate the term on that test case and follow the appropriate edge down into the tree.

In Figure 3, we are inserting `xs++xs` into the tree. When we evaluate `xs++xs` on the test case `xs = [1]`, `ys = []`, there is no edge labelled with the result `[1, 1]`. This means that `xs++xs` is not equal to any existing term in the tree and we insert it at that point.

[6] On the pretty-printing example, switching off this threshold increases runtime by a factor of 5, so recovering completeness is possible at a cost.

[7] The term $(x+y)+z$ is equal to $z+(x+y)$ by commutativity, which as an instance of $?+(?+?)$ is a term we have already explored.

[8] In our current implementation, it checks if $t>_{KB}u$ where $>_{KB}$ is the Knuth-Bendix ordering on terms.

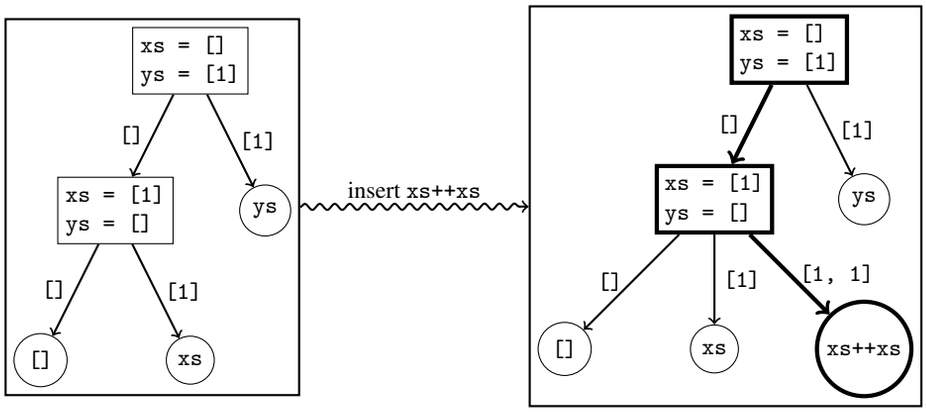


Fig. 3: Inserting a term into a decision tree.

Suppose we instead insert the term `xs++ys`. The decision tree will take us to the leaf `ys`. Does this mean that `xs++ys = ys`? Certainly not! But the decision tree does not contain the right test case to falsify this equation. When insertion reaches a leaf in the decision tree, that gives us a *candidate equation*, which we test using QuickCheck. In this case, testing `xs++ys = ys` will reveal a counterexample such as `xs = [1], ys = [1]`. We then refine the decision tree by adding that test case to it; see Figure 4. As an optimisation, before testing a candidate equation with QuickCheck, we try using all the other test cases in the decision tree; this saves us having to generate new test cases most of the time.

If QuickCheck fails to find a counterexample, then we have discovered a law; our decision tree remains the same.[9]

Notice that we only have to QuickCheck *one* equation, not one for each term in the decision tree; this makes the process efficient. Furthermore, the decision trees are typically very shallow; for the pretty-printing example, the maximum depth is 9. This means that we spend little testing effort on placing terms in the decision tree.[10]

3.4 Pruning

In Section 3.2 we also assumed that we can check if a term can be proved equal to any of the candidate terms using the discovered equations. Recall that we prune terms by *normalising* them somehow with respect to the discovered equations. The pruner consists of a function

$$norm :: Set\ Equation \to Term \to Term$$

which computes the normal form of a term. To check if a term is redundant, we check if its normal form is equal to the normal form of an existing term. By maintaining a set containing all terms' normal forms, we can do this check efficiently; the only cost is that, when we discover a new law, we need to renormalise this set.

[9] The function *consider* will then renormalise the set of terms, but the decision tree is not affected.

[10] On the other hand, the decision trees can be wide: one node can have many children. We therefore store the children of each node not in a list, but in a map (whose keys are test case results), which ensures good performance during insertion.

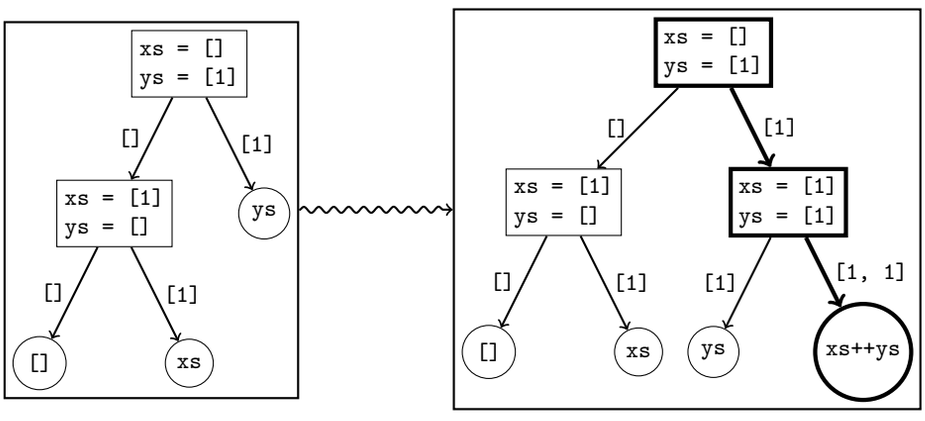


Fig. 4: Adding the test case `xs = [1], ys = [1]` and the term `xs++ys`.

The question is: how to do this normalisation? We would like two terms to have the same normal form if and only if they can be proved equal, but this is impossible because equational theorem proving is not decidable. Instead, we must have an *approximation*: if two terms have the same normal form, they must be equal; but even if they can be proved equal they may have different normal forms. We want this to happen rarely, because every time it happens, we discover an equation that is redundant. Furthermore, normalisation must be very quick, because we generate tens of thousands of terms.

We have implemented a pruner based on term rewriting. The core of the pruner works on first-order, simply-typed terms; a higher layer translates away polymorphic types (section 3.5) and higher-order features (section 3.7). In addition, whenever we normalise a term it is first Skolemised: its variables are replaced by fresh, uninterpreted constants. This allows the normalisation function to only consider ground terms. Recall from section 3.2.2 that we also have a total, well-founded ordering on ground terms, the *term order*.

Naive term rewriting One simple idea is to use the discovered equations as rewrite rules. To normalise a ground term, we look for an equation that we we can apply once to rewrite the term to a lesser term. We repeat this process until all the matching equations lead to a greater term. For example, given the law $xs +\!\!+ [\,] = xs$, we would normalise the term $xs +\!\!+ (ys +\!\!+ [\,])$ to $xs +\!\!+ ys$.

This process is very simple to implement and quite efficient. However, it tends to generate many redundant laws. For example, given the two pretty-printing laws

$$(x \diamond y) \diamond z = x \diamond (y \diamond z)$$
$$x \diamond \mathit{text}\ \texttt{""} = x,$$

it is a fact that

$$x \diamond (\mathit{text}\ \texttt{""} \diamond y) = x \diamond y$$

by reassociating the left-hand side. However, the term $x \diamond (\mathit{text}\ \text{“”} \diamond y)$ is already in normal form, as applying associativity right-to-left gives a *greater* term. As a result, we get the fact above as an extra law.

Knuth-Bendix completion Instead, we use Knuth-Bendix completion (Knuth & Bendix, 1983). Completion takes a set of equations and produces a confluent, terminating set of rewrite rules which are equivalent to the input equations. For example, given the two equations above it will produce the two rules

$$(x \diamond y) \diamond z \to x \diamond (y \diamond z)$$
$$x \diamond \mathit{text}\ \texttt{""} \to x$$

as well as a third one,

$$x \diamond (\mathit{text}\ \texttt{""} \diamond y) \to x \diamond y.$$

This third rule is exactly what we were missing in the previous section: QuickSpec will no longer print the third law, as completion proves $x \diamond (\mathit{text}\ \texttt{""} \diamond y)$ equal to $x \diamond y$. Completion works by finding pairs of equal terms that cannot be proved equal by simplification and adding those equalities as new rewrite rules. When it terminates, it guarantees that two terms are equal if and only if they have the same normal form modulo the rewrite rules. When completion works it is perfect, but there are two ways in which it can fail, and we have to account for both of them in our pruner.

First, completion only works with orientable equations. Given an equation $t = u$, it needs to turn it into a rewrite rule $t \to u$, such that all instances of t are greater than the corresponding instance of u. This is necessary to ensure that the resulting rewrite system is terminating, but it is not always possible. For example, in a commutativity law, $t + u = u + t$, both sides are renamings of one another, so whichever way we try to orient the equation, there will be instances where the right-hand side is greater than the left. We solve this problem using *unfailing completion* (Bachmair *et al.*, 1989) and *ground joinability testing* (Martin & Nipkow, 1990), two standard techniques which allow completion to keep certain rules as unoriented equations.

A more serious problem is that completion is not guaranteed to terminate. It can go on generating more and more rewrite rules forever. This problem is easy for us to solve as the new rules must get bigger and bigger and in practice they rapidly become enormous. We simply throw away any rule whose left-hand or right-hand side is bigger than the maximum term size we are exploring. This means that our pruner can fail to prove an equation whose proof goes through a very large intermediate subterm, but finds all proofs that do not do this. This ensures termination; in practice, it is also fairly efficient and lets few redundant equations through.

3.5 Polymorphism

We have not yet talked about types. It is easy to support *simple* types in QuickSpec: we make sure to enumerate only well-typed terms and schemas, and we send typed equations to the pruner, which must respect those types while doing rewriting.

Polymorphic functions are trickier. We would like to just enumerate polymorphic schemas and feed them to our algorithm, but there are several difficulties:

1. We rely on the property that all schemas have a most specific instance, namely the one-variable instance. Polymorphic schemas do not have this property. For example,

if $f :: a \to Int \to Bool \to b$, then both $f\ x\ x\ y$ and $f\ x\ y\ x$ are well-typed, but not $f\ x\ x\ x$, so the schema $f\ ?\ ?\ ?$ has *two* most specific instances.

2. We may need to discover laws between terms whose types are not equal, but unifiable. For example, given $sort :: Ord\ a \Rightarrow [a] \to [a]$ and a specialised sorting function for lists of integers $intsort :: [Int] \to [Int]$, we would like to discover $sort\ xs = intsort\ xs$, but the schemas *sort* ? and *intsort* ? have different types.
3. Given a polymorphic function such as *map*, we can construct terms of quite strange types such as $map\ (map\ (map\ (map\ map))) :: [[[[a \to b]]]] \to [[[[[a] \to [b]]]]]$. The user is unlikely to be interested in laws about the type $[[[[[a] \to [b]]]]]$: after all, it probably does not appear in the program. Furthermore, QuickCheck's size control interacts badly with deeply nested types so we will generate extremely large test data.
4. Our pruning algorithms are based on term rewriting, which typically assumes a simply-typed setting.

We solve these problems with an incomplete but effective approach. The goal is to leave the basic algorithms alone as far as possible, keep the core of QuickSpec simply-typed, and add a small layer on top which deals with polymorphism.

Monomorphic testing Instead of monomorphic schemas, we now enumerate polymorphic schemas. However, before we consider each schema, we first monomorphise it, setting all its type variables to an abstract type A. In reality, this type is implemented by *Int*, so that we can test monomorphised schemas. The algorithm to consider a schema then works as before, with no changes for polymorphism—except that we will not discover polymorphic laws, but monomorphic laws mentioning the type A. Thus once we have discovered a law $t = u$, containing polymorphic functions specialised to the type A, we do type inference to find the law's most general type.

For example, given the functions *map* and $+\!\!+$, we will discover the law

$$map\ f\ (xs +\!\!+ ys) = map\ f\ xs +\!\!+ map\ f\ xs$$

where the variables *xs* and *ys* have type $[A]$ and f has type $A \to A$. Type inference will then replace the occurrences of A by type variables, generalising *xs* and *ys* to type $[a]$ and f to type $a \to b$. This generalisation does not change how the law is shown to the user, but it is important to give the pruner the polymorphic version so that it can apply this law at any type, not just A.

This trick sacrifices some completeness. For example, given the function $f :: a \to Int \to Bool \to b$ from above, we will generate the schema $f\ ?\ ?\ ?$ but then monomorphise a and b to A. The terms $f\ x\ x\ y$ and $f\ x\ y\ x$ will not be well-typed, because $A \neq Int$ and $A \neq Bool$. Note that we still generate all schemas; we just lose some instances of some of them. This incompleteness has not been a problem in practice.

Type unification Let us call a schema the *representative* if when we considered it, its one-variable instance was not equal to any other term. Whenever we have two representative schemas whose polymorphic types are unifiable, we generate instances of both schemas specialised to the most general unifier (mgu) of their types. This ensures that, for example, the schema '?' (which has the polymorphic type a) gets generated at every type. In the

example with *sort* and *intsort*, we will notice that their types are unifiable to $[Int] \to [Int]$ and generate $sort :: [Int] \to [Int]$, which will let us discover $sort\ xs = intsort\ xs$.

A type universe To avoid terms of absurd types such as $map\ (map\ (map\ (map\ map))) :: [[[[a \to b]]]] \to [[[[[a] \to [b]]]]]$, we restrict the set of allowed types. Before exploring the program, we compute a *universe of types*, which is a guess at the set of types the programmer might be interested in. We are only interested in a law if the types of all its subterms are in the universe.

We therefore only generate a schema if:

- the types of all its proper subterms are in the universe, and
- the schema's type itself is in the universe, or it is a function whose result type is in the universe (because we might later generate the schema applied to an argument).

The type universe is in reality a set of *monomorphic* types, and we consider a polymorphic type to be in the universe if there is some monomorphic instance in the universe. Currently the type universe consists of all argument and result types of the functions in the signature, monomorphised so that all type variables are replaced by the abstract type A from above.

Using a type universe restricts the set of types we consider, ruling out silly types that do not appear in the user's program. It also makes the pruner's job easier. Each function in the signature can only be instantiated to a finite set of monomorphic types. Therefore, whenever we discover a polymorphic law, we generate all monomorphic instances and pass them to the pruner. To normalise a polymorphic term, we monomorphise all type variables to A before passing it to the pruner. The pruner itself does not have to deal with polymorphic laws or terms; it is only given simply-typed laws, as before.

Example Suppose QuickSpec is given the signature below:

$$\begin{array}{ll} 0 & :: Int \\ (+\!+) & :: [a] \to [a] \to [a] \\ sum & :: [Int] \to Int \\ (+) & :: Int \to Int \to Int \\ concat & :: [[a]] \to [a] \end{array}$$

The type universe consists of the types: Int, $[Int]$, A, $[A]$ and $[[A]]$, which are the only types QuickSpec will allow in terms it constructs. This allows QuickSpec to discover, for example, the law $concat\ (xs +\!+ ys) = concat\ xs +\!+ concat\ ys$, where the left-hand side occurrence of $+\!+$ has type $[[a]] \to [[a]] \to [[a]]$ and the right-hand side occurrence has type $[a] \to [a] \to [a]$.

3.6 Observational equivalence and background theories

Sometimes, we want to generate laws about a type which we cannot compare directly, but for which we have a notion of observational equivalence: two values of that type are equivalent if all observations make them equal. To support this, for any type T the user can supply an observation function of type $Obs \to T \to Res$, where Obs can be any type that we can generate random data for, and Res any type we can compare for equality. QuickSpec

will then include a random value of type *Obs* as part of each test case, and will compare values of type T by applying these random observations.

We also normally want to supply background functions, functions that can appear in generated laws but are not interesting in their own right, such as $+$ and $+\!+$ in the pretty-printing example. To do this, we first run QuickSpec using *only* the background functions, and it discovers all laws about them. We then re-run QuickSpec, adding the remaining functions but also giving the pruner all the laws we have already discovered. The background laws will therefore be used for pruning; in particular, they will not be printed again, because the pruner will consider the terms in them to be redundant. As future work, we plan to augment QuickSpec with a library of already discovered properties for commonly occuring background functions, to avoid having to explore them repeatedly.

3.7 Higher-order functions

Higher-order functions bring a few challenges of their own. The first is how to compare terms of function type for testing. We solve this using QuickSpec's support for observational equivalence from Section 3.6. When we want to compare two functions for equality, we generate a random argument, apply both functions to that argument and compare the result.

A more difficult problem is pruning. Our pruning infrastructure is based on first-order term rewriting. In this setting each function has a fixed arity: you cannot write *map f* if *map* is a function of arity 2. Also, you can only apply a function symbol, not a variable, so you can't write $f(x)$ if f is a variable.

A simple and well-known solution is for the pruner to treat Haskell functions as *constants* and introduce a binary function symbol *apply* for function application. The term *map f xs* will be $apply(apply(map,f),xs)$ as far as the pruner is concerned. QuickSpec will transform all the laws it discovers, and all the terms it wants to normalise, into this form before passing them to the pruner. This neatly solves both problems above as we can write $apply(map,f)$ to partially apply a function or $apply(f,x)$ to apply a variable.

A more efficient encoding Unfortunately, this encoding is inefficient and puts stress on the pruner. For example, many equational reasoning algorithms treat commutative functions specially and look for axioms of the form $f(x,y) = f(y,x)$. But we will express this axiom as $apply(apply(f,x),y) = apply(apply(f,y),x)$, and any pruning algorithm which syntactically checks for commutativity will not detect it. Performance will suffer: pruning will take much longer, or will be able to prove fewer equations.[11]

Instead, we have a hybrid encoding which uses *apply* only when necessary. Let us illustrate it with an example. Suppose we have the binary function *map*. We will introduce three function symbols: $map_2(f,xs)$ of arity 2, $map_1(f)$ of arity 1, and map_0 of arity 0. The first represents a fully-applied call to *map* and the other two are partial applications. To

[11] QuickSpec's built-in pruner does not suffer from this problem, but the encoding still hurts performance by cluttering the axioms. Occasionally it can be useful to connect QuickSpec to an external theorem prover to filter redundant equations even more thoroughly; in this case the encoding is really harmful.

relate these functions to one another, we add the axioms $apply(map_0,f) = map_1(f)$ and $apply(map_1(f),xs) = map_2(f,xs)$.

Whenever we have an expression like $f\ x$ where f is a variable, we translate it to $apply(f,x)$. We do the same with over-saturated functions like $foldr\ (\circ)\ id\ fs\ x$, where the fourth argument will be passed using *apply* as *foldr* has arity 3. Apart from these two cases, the translated laws will not use *apply*.

Point-free reasoning and higher-order functions To illustrate how higher-order functions are explored in practice, let us see how QuickSpec explores the functions *map*, *id* and $(\circ)$. We will ignore the process of discovering the properties and concentrate on the pruning, picking a few laws that illustrate features of our encoding.

Recall that for each Haskell function f of arity k, the pruner is given several function symbols $f_0,\ldots,f_k$ which represent partial applications of f. For our example these function symbols are map_0, map_1 and map_2 for the map function; id_0 and id_1 for the identity function; and $compose_0, compose_1, compose_2, compose_3$ for function composition. We also introduce the function symbol *apply* and the following axioms:

$$apply(map_0,f) = map_1(f)$$
$$apply(map_1(f),xs) = map_2(f,xs)$$
$$apply(id_0,x) = id_1(x)$$
$$apply(compose_0,f) = compose_1(f)$$
$$apply(compose_1(f),g) = compose_2(f,g)$$
$$apply(compose_2(f,g),x) = compose_3(f,g,x)$$

The first law QuickSpec will discover is $id\ x = x$. As a result it feeds the pruner the following equation:

$$id_1(x) = x$$

Notice that we do not use *apply* in the discovered law. Next QuickSpec might discover $map\ id = id$ (recall that we test this law by applying both sides to a random argument). QuickSpec feeds the pruner the following equation:

$$map_1(id_0) = id_0$$

Next QuickSpec might generate the terms *xs* and *map id xs*. Of course, the law $map\ id\ xs = xs$ follows from $map\ id = id$, and $id\ xs = xs$, but does our encoding allow the pruner to see this? The answer is yes: the pruner reasons as follows:

$$\begin{aligned}
& map_2(id_0,xs) \\
&= apply(map_1(id_0),xs) && \text{-- axiom for } apply \\
&= apply(id_0,xs) && \text{-- } map\ id = id \\
&= id_1(xs) && \text{-- axiom for } apply \\
&= xs && \text{-- } id\ xs = xs
\end{aligned}$$

Next QuickSpec might discover that $(f \circ g)\ x = f\ (g\ x)$. It feeds the pruner the following equation:

$$compose_3(f,g,x) = apply(f,apply(g,x))$$

Notice that on the right-hand side we apply a variable. QuickSpec might also discover $map\,f \circ map\,g = map\,(f \circ g)$, which becomes:

$$compose_2(map_1(f), map_1(g)) = map_1(compose_2(f,g))$$

Of course, we now want QuickSpec to be able to prune away the law $map\,f\,(map\,g\,xs) = map\,(f \circ g)\,xs$. Indeed it can:

$$\begin{aligned}
& map_2(f, map2\ (g, xs)) \\
&= apply(map_1(f), apply(map_1(g), xs)) && \text{-- axiom for } apply \\
&= compose_3(map_1(f), map_1(g), xs) && \text{-- } (f \circ g)\ x = f\ (g\ x) \\
&= apply(compose_2(map_1(f), map_1(g)), xs) && \text{-- axiom for } apply \\
&= apply(map_1(compose_2(f,g)), xs) && \text{-- } map\,f \circ map\,g = map\,(f \circ g) \\
&= map_2(compose_2(f,g), xs) && \text{-- axiom for } apply
\end{aligned}$$

Once QuickSpec has discovered a law, the axioms for *apply* allows it to eta-expand that law during pruning. It may seem like the pruner has to do a lot of work, expanding and contracting the definition of *apply*, but this sort of reasoning is the bread and butter of Knuth-Bendix completion. Furthermore, the pruner need only reason about *apply* when there is a partially-applied function; first-order reasoning is unchanged.

Although our encoding allows the pruner to eta-expand laws, it does not allow it to eta-reduce them. For example, the pruner will not be able to show that

$$id \circ f = f,$$

even though it can prove the eta-expanded version, $(id \circ f)\ x = f\ x$, by expanding the definition of $(\circ)$. Thus, even though we have discovered the definitions of *id* and $(\circ)$, QuickSpec will also find three point-free laws about them:

$$\begin{aligned}
& id \circ f = f \\
& f \circ id = f \\
& (f \circ g) \circ h = f \circ (g \circ h)
\end{aligned}$$

In general, QuickSpec is able to prune away a point-free law if it can be proved *using point-free reasoning*, that is, without eta-expanding the law. This is a deliberate choice: without it, QuickSpec would prune away the three laws above, and point-free laws that can only be proved by reasoning about points seem noteworthy. If we wanted, we could have QuickSpec eta-expand each term before passing it to the pruner, which would filter out such laws.

3.8 Conditional Equations

As described so far, QuickSpec is only able to find equations that hold unconditionally. However, many interesting equations hold under certain conditions. In this case, QuickSpec may find no law, or find overly-specific laws.

For example, when asked to generate laws about *zip* and $+\!\!+$, QuickSpec produces the following equation:

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

The equation is certainly valid, but too specific: both sides contain two occurrences of *xs*, whereas the law actually holds for different choices of *xs*, as long as they have the same length.

We have implemented a small extension to basic QuickSpec that, given a limited set of interesting *condition predicates*, is able to produce conditional laws using those predicates as conditions.

For the example, we can explicitly specify *length xs* = *length ys* as a condition predicate, in which case QuickSpec produces the law:

$$length\ xs_1 = length\ xs_2 \Rightarrow zip\ xs_1\ (xs_2 +\!\!+ ys) = zip\ xs_1\ xs_2$$

This is indeed the expected, most general version of the law.

The extension formulates the problem of generating conditional laws in terms of generating unconditional laws, by means of introducing a *new type* with selection functions, for every condition predicate that is explicitly specified by the user. For example, for the condition predicate *length xs* ≡ *length ys*, the extra type (called *ListsSameLength* below) that is introduced looks as follows:

```
data ListsSameLength a = Lists [a] [a]
xs1, xs2 :: ListsSameLength → [a]
xs1 (Lists xs _)  = xs
xs2 (Lists _ ys) = ys
```

After this new type is introduced, QuickSpec also needs a way to generate two lists of the same length. The default method of doing this in QuickSpec given a condition predicate is to use the QuickCheck combinator *suchThat* that repeatedly generates a random value until it satisfies a predicate.

```
instance Arbitrary a ⇒ Arbitrary (ListsSameLength a) where
  arbitrary =
    do (xs, ys) ← arbitrary `suchThat` (λ(xs, ys). length xs ≡ length ys))
       return (Lists xs ys)
```

All the above code is automatically generated from the given predicate *length xs* ≡ *length ys*, and enough to produce conditional laws having that predicate as a condition. Basic QuickSpec indeed finds the following (now unconditional) law:

$$zip\ (xs_1\ p)\ (xs_2\ p +\!\!+ ys) = zip\ (xs_1\ p)\ (xs_2\ p)$$

Here, *p* is a variable that quantifies over *ListsSameLength*. Our extension then automatically pretty-prints the above law as the conditional law about *zip* and ++ we saw earlier.

In general, using *suchThat* may not be an effective way to produce a generator for generating arbitrary values that satisfy a given predicate. This is why our condition extension allows the user to specify their own generators if they wish. It is actually possible to automatically derive a generator from a given predicate, which has been described in related work (Claessen *et al.*, 2014; Duregård, 2016).

The reader may wonder why we require user assistance at all when dealing with conditions. Why do we not simply consider all Boolean predicates in the given signature as conditions, or even consider all equations between terms of the same type as conditions? There are two

reasons for this. First, the sheer number of possibilities of general conditional equations is several orders of magnitudes larger than for unconditional equations. Secondly, pruning for conditional equations is much more difficult than for unconditional ones. To prune conditional equations, we would need full first-order logic reasoning, something we do not currently know how to use to *a priori* prune away whole classes of equations to consider.

Solving these problems in a general way is therefore future work. We believe that letting the user specify the interesting conditions to use is a good compromise between simplicity and power. Our approach allows for complex equations but only with simple, well-understood conditions; these sorts of laws are very common in functional programming.

3.9 A summary of QuickSpec's architecture

We have now seen all of QuickSpec's architecture. Figure 5 summarises the entire design.

The main QuickSpec pipeline is drawn in bold. In the leftmost box, QuickSpec enumerates schemas in order of size. The schemas go through the schema layer of Section 3.2.3, which either discards each schema or adds it to the candidate set, and potentially instantiates it, sending instances to the term layer of Section 3.2.2. Both layers maintain a set of candidates (for pruning) as well as a decision tree for those candidates (for testing); the schema layer's decision tree contains the one-variable instances of the candidates.

When a law is discovered, it is added to the pruner, and the candidate terms and schemas are renormalised. The pruner consists of the implementation of Knuth-Bendix completion mentioned in Section 3.2.3, on top of which sit layers that strip away higher-order functions (Section 3.5) and types (Section 3.7).

As described in Section 3.2.2, we do not want to explore every term, but only those whose subterms are in the candidate term set. The schema enumeration does exactly this, but uses candidate schemas instead of terms; this explains the arrow from "Candidate schemas" to "Enumerate schemas".

Towards the end of Section 3.2.3, under *Associative-commutative functions*, we described a feature which reorders the discovered laws in the hope of finding a nicer formulation of them. This component sits at the end of the pipeline, filtering out laws just before they are printed.

The dotted box represents the polymorphism layer: everything inside the dotted box deals only with monomorphic terms, schemas and laws. This layer takes care of introducing and eliminating polymorphism as described in Section 3.5. In particular, schemas are monomorphised before they reach the schema layer, and the discovered laws are made polymorphic as they leave the box on the right. This layer also generates the appropriate type instances of all candidate schemas.

Finally, the diagram does not show QuickSpec's conditional equation support 3.8. Conditional equations are implemented on top of QuickSpec: a preprocessing step transforms the signature to introduce the new types and selection functions, and the final laws are printed by a conditional-aware algorithm.

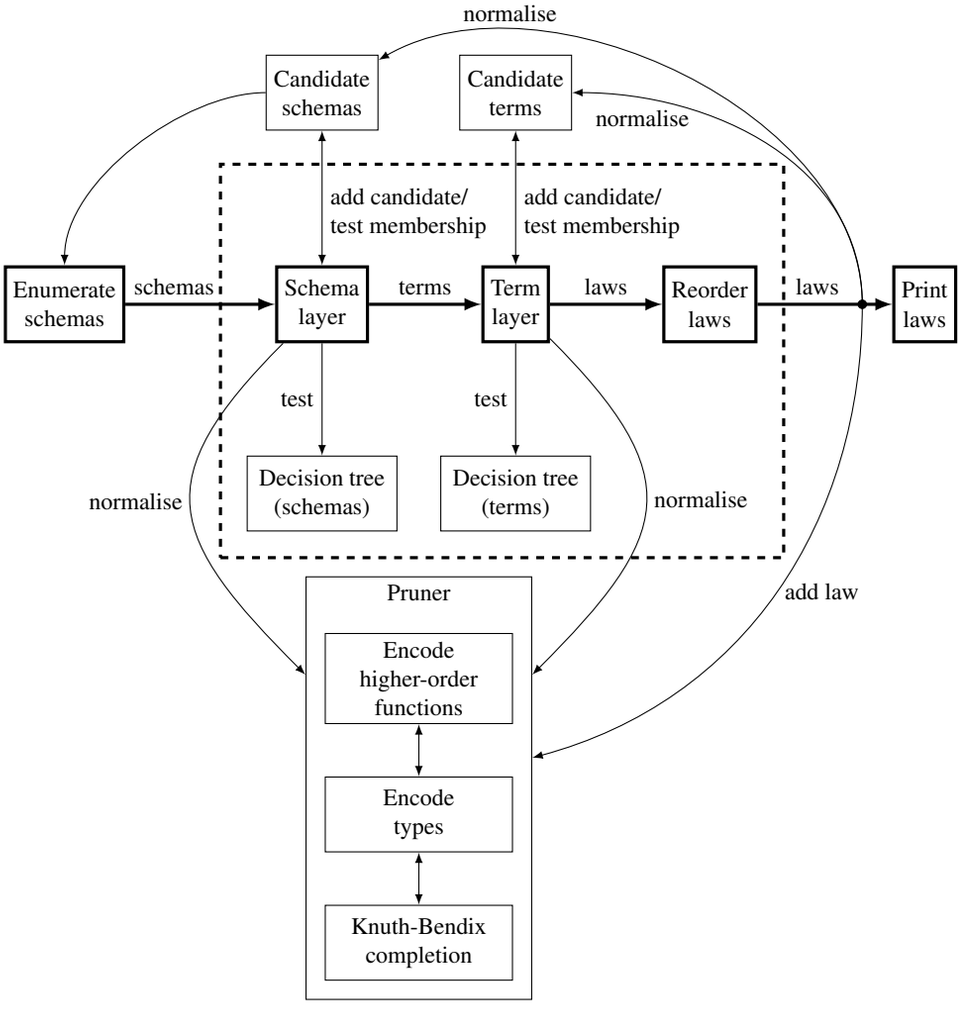


Fig. 5: QuickSpec's architecture. The bolded part is the main pipeline. Everything inside the dotted box only deals with monomorphic laws.

4 Case Studies

In this section we try out QuickSpec on four difficult examples which the previous version, QuickSpec 1, was too slow to handle well (see section 5.1). These examples are: Henderson's *functional geometry* combinators for describing pictures (Section 4.1), a large collection of list functions from the Haskell standard library (Section 4.2), an example from mathematics where we use QuickSpec to explore the theory about *octonions* (Section 4.3), and finally the *Map* datastructure from the Haskell standard library (Section 4.4). We had four goals in trying these examples:

1. To demonstrate how QuickSpec can be used to improve understanding of programs and find bugs.

2. To look at the generated laws critically, and use them to suggest ways to improve QuickSpec.
3. To stress-test QuickSpec and make sure it copes.
4. To have fun exploring interesting signatures that we did not fully understand.

The code for the experiments is available from the QuickSpec GitHub repository:
`github.com/nick8325/quickspec/tree/master/examples`

4.1 Functional geometry

Henderson's *functional geometry* (Henderson, 1982; Henderson, 2002) is a set of combinators for describing pictures. In his papers, he uses these combinators to reconstruct an M. C. Escher woodcut out of simple building blocks. Let's see what laws his combinators satisfy!

The five basic combinators (four of them illustrated in Figure 6) are *above*, which draws one picture above another, *beside*, which draws one picture beside another, *over*, which draws two pictures at the same spot, *rot*, which rotates a picture by 90°, and $flip$, which flips a picture horizontally.

An unusual feature of Henderson's combinators is that pictures do not have a built-in size. Rather, when you draw a picture, you give a size, and the picture fits itself to that size. The *above* and *beside* combinators divide the available space into two, one for each sub-picture. We would therefore not expect *above* and *beside* to be associative as, for example, in the picture $beside\ x\ (beside\ y\ z)$, the picture x gets 50% of the space while y and z get 25% each. We weren't sure what laws *would* hold, though, and what better way to find out than by using QuickSpec!

Henderson defines a denotational semantics for his pictures as a function taking a position and size for the drawing and returning a set of primitive shapes (whose nature is left unspecified). We simply implemented this semantics in Haskell, gave the combinators to QuickSpec, and ran it. Based on our experience we chose to explore terms up to size 7, this is usually a good initial choice covering many interesting properties.

We decided that we would like to explore simpler combinators on their own before adding the more complex ones, so we ran QuickSpec incrementally, first exploring *over*, then adding *beside* and *above*, then adding *rot*, and finally $flip$.

The laws for *over* are not surprising: it is commutative, associative and idempotent. We also get $over\ x\ (over\ x\ y) = over\ x\ y$, which follows from associativity and idempotence, but happens to be discovered before associativity. This is because QuickSpec considers $over\ x\ (over\ x\ y)$ to be simpler than $over\ (over\ x\ y)\ z$; perhaps QuickSpec should wait a bit after discovering a law before printing it, to see if it discovers a generalisation.

$$\begin{aligned} over\ x\ x &= x \\ over\ x\ y &= over\ y\ x \\ over\ x\ (over\ x\ y) &= over\ x\ y \\ over\ (over\ x\ y)\ z &= over\ x\ (over\ y\ z) \end{aligned}$$

Once we add *above* and *beside*, things get more interesting. The following four laws state that, when drawing two pictures in the same place using *over*, any commonalities between

the pictures can be factored out:

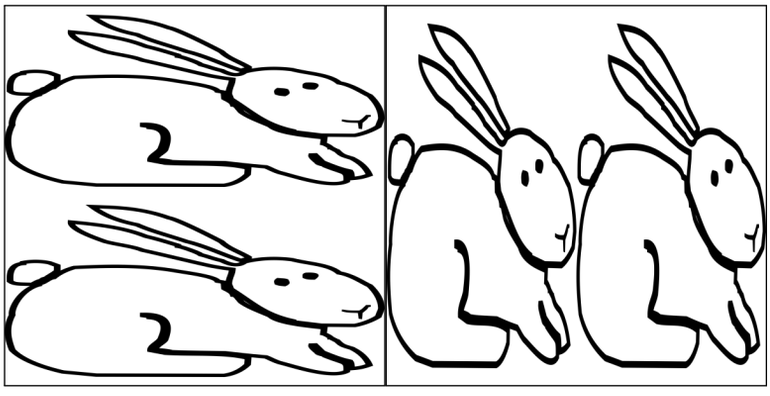
$$over\ (above\ x\ y)\ (above\ x\ z) = above\ x\ (over\ y\ z)$$
$$over\ (above\ x\ z)\ (above\ y\ z) = above\ (over\ x\ y)\ z$$
$$over\ (beside\ x\ y)\ (beside\ x\ z) = beside\ x\ (over\ y\ z)$$
$$over\ (beside\ x\ z)\ (beside\ y\ z) = beside\ (over\ x\ y)\ z$$

We can visualise the first law like so; the others are similar:

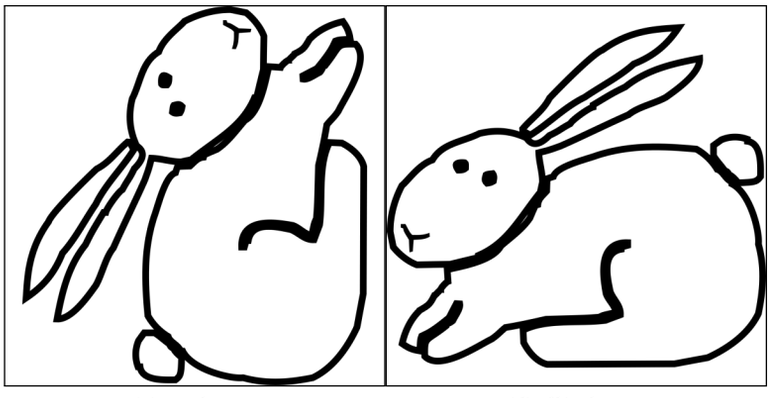
$$\begin{array}{|c|}\hline x \\ \hline y \\ \hline \end{array}\ over\ \begin{array}{|c|}\hline x \\ \hline z \\ \hline \end{array} = \begin{array}{|c|}\hline x \\ \hline y\ over\ z \\ \hline \end{array}$$

The four laws are in fact consequences of the following two laws, which QuickSpec discovers shortly afterwards, which state that *over* and *above* commute with each other, as do *over* and *beside*:

$$above\ (over\ x\ z)\ (over\ y\ w) = over\ (above\ x\ y)\ (above\ z\ w)$$



(a) *above bunny bunny* (b) *beside bunny bunny*



(c) *rot bunny* (d) *flip bunny*

Fig. 6: Picture combinators and bunnies.

$$beside\ (over\ x\ z)\ (over\ y\ w) = over\ (beside\ x\ y)\ (beside\ z\ w)$$

Really, we might wish that QuickSpec had only printed these two laws and not the previous four. This is the same problem we had with the *over* combinator. Anyway, after these laws we get a final, rather pleasing, one:

$$beside\ (above\ x\ z)\ (above\ y\ w) = above\ (beside\ x\ y)\ (beside\ z\ w)$$

Both sides of this equation draw four figures, x, y, z and w, in a square. The left-hand side draws x-z-y-w, top-to-bottom, left-to-right, while the right-hand side draws x-y-z-w, left-to-right, top-to-bottom. The law states that both constructions produce the same figure.

Adding *rot*, we get four new laws:

$$\begin{aligned}
over\ (rot\ x)\ (rot\ y) &= rot\ (over\ x\ y)\\
above\ (rot\ y)\ (rot\ x) &= rot\ (beside\ x\ y)\\
beside\ (rot\ x)\ (rot\ y) &= rot\ (above\ x\ y)\\
rot\ (rot\ (rot\ (rot\ x))) &= x
\end{aligned}$$

The first law states that *rot* distributes over *over* (not surprising, as pretty much everything distributes over *over*). The second and third laws show that rotating an *above* gives a *beside*, and vice versa. Notice that in one of the laws the x and y are swapped on the left-hand side; the geometrically-astute reader can use this fact to deduce that *rot* rotates the picture anti-clockwise:

$$rot\ \boxed{x}\boxed{y} = \begin{array}{|c|}\hline rot\ y \\ \hline rot\ x \\ \hline \end{array}$$

$$rot\ \begin{array}{|c|}\hline x \\ \hline y \\ \hline \end{array} = \boxed{rot\ x}\boxed{rot\ y}$$

Finally, rotating a picture by 360° gets you back where you started.

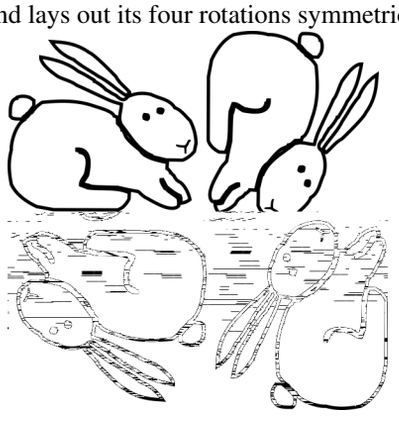
Lastly, we add $flip$ and get the following four laws:

$$\begin{aligned}
flip\ (flip\ x) &= x\\
rot\ (flip\ (rot\ x)) &= flip\ x\\
over\ (flip\ x)\ (flip\ y) &= flip\ (over\ x\ y)\\
above\ (flip\ x)\ (flip\ y) &= flip\ (above\ x\ y)
\end{aligned}$$

We see that flipping a picture twice does nothing, that rotating then flipping then rotating is the same as flipping,[12] that $flip$ distributes over *over*, and that it also distributes over *above*. Note that $flip$ does not distribute over *beside*: this indicates that $flip$ flips its argument horizontally rather than vertically.

[12] Rotating a bunny in your head may help to visualise this.

Henderson also defines two larger combinators, *quartet* and *cycle*, in terms of the primitive ones. The *quartet* function takes four figures and draws them in a square formation, while *cycle* takes a figure and lays out its four rotations symmetrically in a square, like so:



QuickSpec discovers the definition of *quartet*:

$$above\ (beside\ x\ y)\ (beside\ z\ w) = quartet\ x\ y\ z\ w$$

It also discovers three laws about *cycle*:

$$rot\ (cycle\ x) = cycle\ x$$
$$flip\ (cycle\ (rot\ x)) = cycle\ (flip\ x)$$
$$over\ (cycle\ x)\ (cycle\ y) = cycle\ (over\ x\ y)$$

The first law states that *cycle* is rotationally symmetric. The second law says that, very curiously, cycling $flip\ x$ versus $rot\ x$ gives two pictures that are mirror images of one another! Again, the reader may visualise this by drawing bunnies. Finally, *cycle* distributes over *over*.

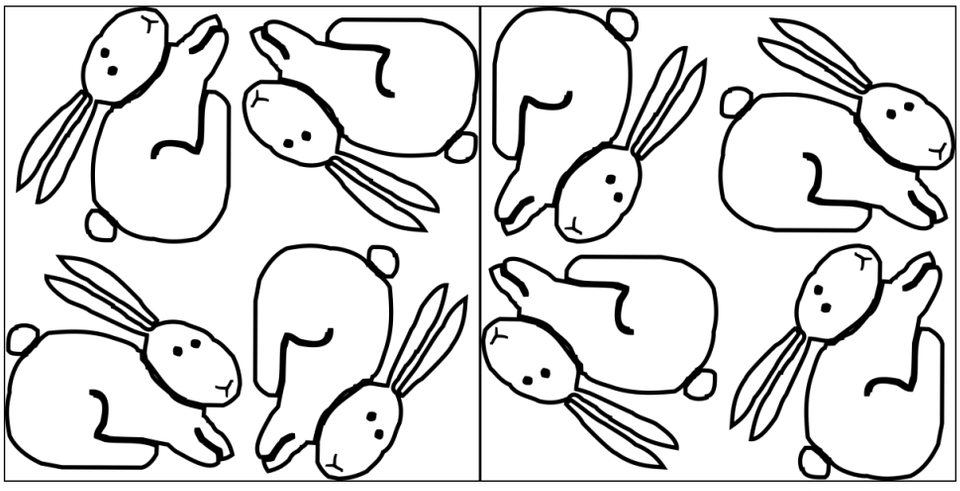
4.1.1 An odd set of laws

These are not the laws we originally found. When we first ran QuickSpec, it claimed that *rot* was not rotationally symmetric, but only 180° rotationally symmetric! This caused lots of head-scratching until we realised that another law was subtly wrong:

$$above\ (rot\ x)\ (rot\ y) = rot\ (beside\ x\ y)$$

We take a picture in which x is to the left of y and we rotate it anticlockwise. Afterwards, x should end up below y, but according to the law, it ends up *above*! It turns out that in Henderson's 2002 paper, the definition of *above* is slightly wrong: it swaps its arguments, and should more rightly be called *below*. Thus, an unexpected property (*cycle* not being as symmetric as expected) has alerted us to a bug in the implementation! Bugs often manifest themselves as missing or warped QuickSpec properties.

It turns out that this bug is quite exciting. The *cycle* combinator uses *above* to lay out the picture, and when *above* is buggy *cycle* produces an entirely different picture, seen in Figure 7(a). When we rotate the picture, we get Figure 7(b): this buggy *cycle* is indeed not rotationally symmetric.



(a) *cycle bunny* (b) *rot* (*cycle bunny*)

Fig. 7: The buggy *cycle* is not rotationally symmetric.

When we run QuickSpec on the buggy *cycle*, we no longer get the law $rot\ (cycle\ x) = cyclex$, but instead we get the following three laws:

$$rot\ (rot\ (cycle\ x)) = cycle\ x$$
$$rot\ (cycle\ (cycle\ x)) = cycle\ (cycle\ x)$$
$$cycle\ (rot\ (rot\ x)) = rot\ (cycle\ x)$$

The first law shows that *cycle* is 180° rotationally symmetric, as Figure 7 demonstrates. The second, though, we found quite staggering: constructing the picture $cycle\ (cycle\ x)$ gives a 4x4 pattern which *is* rotationally symmetric! Trying it with bunnies gives the nice patchwork in Figure 8. Notice that the whole picture is rotationally symmetric, while each quarter of the picture is 180°symmetric, as predicted by QuickSpec. We have tried this construction on a few line drawings and made some quite "arty" pictures.

The third law above explains the previous two more deeply:

$$cycle\ (rot\ (rot\ x)) = rot\ (cycle\ x)$$

In other words, *cycle* isn't rotationally symmetric because rotating *cycle x* gives a figure in which the four *x*s have been rotated by 180°(as Figure 7 confirms). Rotating *cycle x* twice gives a figure where each *x* has been rotated a whole 360°, which explains why *cycle* is still 180° rotationally symmetric. Finally, we can use this law to reason that

$$\begin{aligned} & rot\ (cycle\ (cycle\ x)) \\ = & cycle\ (rot\ (rot\ (cycle\ x))) \\ = & cycle\ (cycle\ (rot\ (rot\ (rot\ (rot\ x))))) \\ = & cycle\ (cycle\ x) \end{aligned}$$

which explains why $cycle\ (cycle\ x)$ is rotationally symmetric.

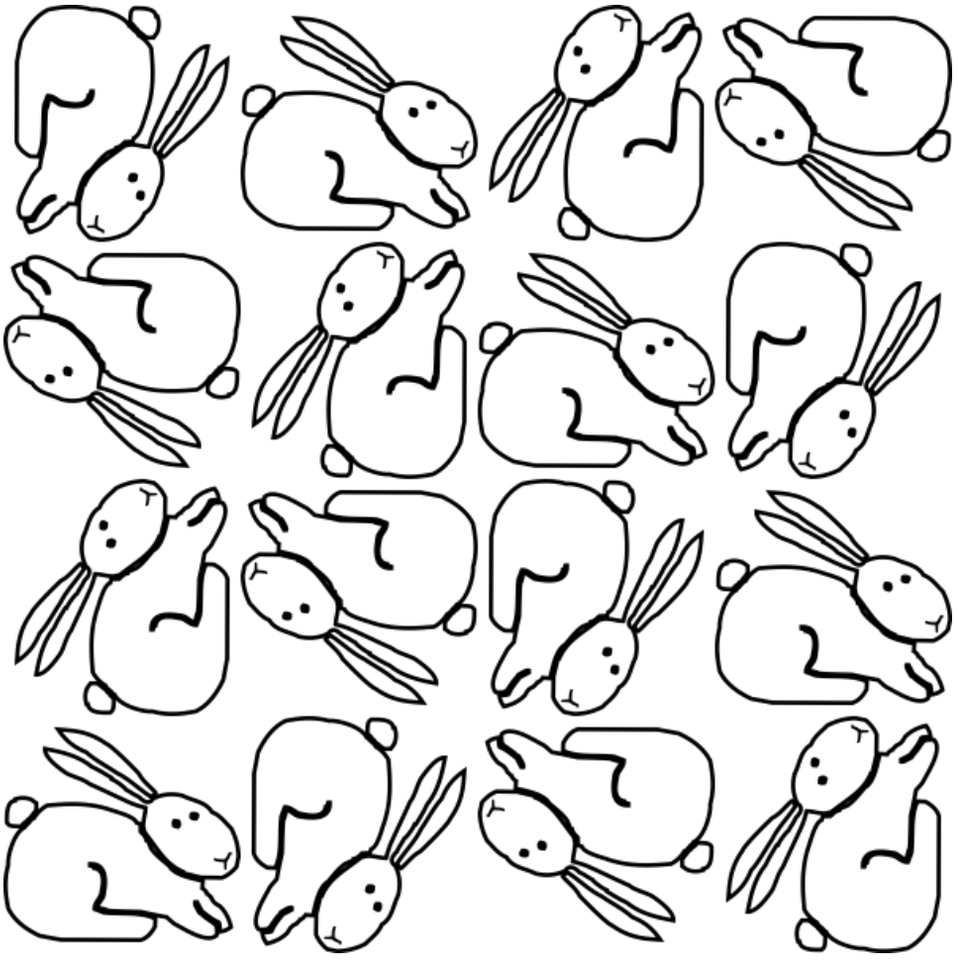


Fig. 8: *cycle* (*cycle bunny*).

4.1.2 The empty picture

Henderson also defines an empty picture *blank*, which represents a blank space. Note that *blank* is not an identity for *above* and *beside*, for the same reason that these operators are not associative.

We originally included *blank*, along with *over*, as the very first functions we put in our signature. Unfortunately, along with the laws we expected like

$$over\ x\ blank = x$$
$$above\ blank\ blank = blank$$
$$beside\ blank\ blank = blank$$

we also got 11 much uglier laws, including:

$$beside\ blank\ (above\ x\ blank) = above\ (beside\ blank\ x)\ blank$$
$$rot\ (beside\ blank\ x) = above\ (rot\ x)\ blank$$

$$flip\ (above\ x\ blank) = above\ (flip\ x)\ blank$$

Notice that these laws are all instances of more general laws that we discovered earlier in this section. For example, the third law is a combination of

$$flip\ (above\ x\ y) = above\ (flip\ x)\ (flip\ y)$$

with the fact that $flip\ blank = blank$. The trouble is that each of these laws is *smaller* than its generalisation, which means that QuickSpec discovers the special case first. One solution might be for QuickSpec to generate terms containing few distinct function symbols before terms containing many; that way, we would find the general case above (which contains $flip$ and *above*) before the special case (which also contains *blank*). As a workaround, we instead added *blank* to the signature *last*, after exploring everything else, so all the general laws were already discovered. By doing so we only got the few laws we expected.

4.1.3 Summary

QuickSpec worked very nicely on the functional geometry combinators. It found revealing laws about all of the combinators, such as observing that $flip$ flips horizontally and *rot* rotates anticlockwise. It enabled us to find a bug in the *above* combinator, and even pointed out a nice construction for making patchwork drawings! Performance-wise, QuickSpec performed well: the basic set of combinators takes 25 seconds to explore.

We also saw that the order in which QuickSpec is asked to explore functions can affect results, with the *blank* function better left until last, when general properties of the other functions have already been discovered. Furthermore, QuickSpec sometimes discovered a law which it later generalised; the initial law was therefore redundant. These problems suggest that simply generating laws in order of size is not optimal; in the future we plan to investigate better ways to order the discovered laws.

4.2 A stress test: lists

Our next example is a pure stress test: we wanted to see how far we could push QuickSpec before it breaks. We did this by giving QuickSpec a very large signature consisting of many functions all at once.

For this experiment, we gave QuickSpec as many list functions as we could think of, namely: `length`, `map`, `concat`, `[]`, `(:)`, `(++)`, `reverse`, `sort`, `usort` (which sorts a list and removes duplicates), `(>>=)`, `(>=>)`, `foldr`, `foldl`, `scanr`, `scanl`, `filter`, `partition`, `break`, `span`, `takeWhile`, `dropWhile`, `take`, `drop`, `zip`, `unzip`, `zipWith`. We also gave it the auxiliary functions `sum`, `0`, `succ`, `(+)`, `(,)`, `fst`, `snd`, and asked it to generate terms up to size 7 on these 33 functions. This took about 42 minutes, during which QuickSpec discovered 398 laws, generated over 50 000 terms and ran around 874 000 tests. It is not sensible to run QuickSpec in such an undirected way; we just wanted to see if it would cope.

Increasing the size to 8, QuickSpec gets most of the way but then starts to run out of memory and slows to a crawl, and eventually hits our time-out limit of two hours. If we limit it to the first 26 functions in our list, it is able to explore all terms of size 8 within two

hours, finding some 586 laws. To complete the whole signature, it appears that QuickSpec would need to test well over 100 000 terms: this should be entirely possible in principle, with an optimised implementation. Indeed, a finely-tuned implementation should be able to go beyond size 8.

So QuickSpec survives its encounter with the huge signature, at least up to size 7. It does, however, generate too many laws. We do discover some nice list laws, such as (when i and j are natural numbers)

$$drop\ j\ (take\ (j+i)\ xs) = take\ i\ (drop\ j\ xs)$$

but we also get a lot which are not so interesting:

$$zip\ (map\ f\ xs +\!\!+ ys)\ xs = zip\ (map\ f\ xs)\ xs$$
$$map\ (f\ x)\ (take\ (succ\ 0)\ xs) = zipWith\ f\ (scanl\ g\ x\ [])\ xs \qquad (*)$$

The first law is really a consequence of the fact that *zip* truncates its arguments to the length of the shortest one, and the fact that

$$length\ (map\ f\ xs +\!\!+ ys) > length\ (map\ f\ xs) = length\ xs$$

so we could prune it away if we had used our extension for finding conditional equations. The second law can be proved by case analysis on whether *xs* is empty or not. Case analysis would be a useful extension to the pruner but might also remove interesting laws; we should add it as a user-configurable option.

The upshot is that running QuickSpec on such a huge signature works, but is not yet user-friendly—laws relating lots of unconnected list functions are bound to be a bit eccentric. However, the user can at least get an initial set of laws this way, and then restrict the signature to focus testing a bit more.

In the future we plan to develop heuristics that eliminate strange laws at the cost of completeness. For example, the user might specify a bound on the number of distinct functions appearing in one law, which would sacrifice law $(*)$ above.

4.3 The Octonions

Can QuickSpec be useful for domains beyond functional programming? In this experiment, we apply QuickSpec to mathematics, and ask it to explore the properties of a somewhat exotic kind of numbers, called the *octonions* (Baez, 2002). The octonions are one of only four *normed division algebras*, the others being the more well known real numbers, the complex numbers and the quaternions. However, unlike the real or complex numbers, multiplication on octonions is neither associative nor commutative. Mathematicians have found weaker axioms that octonions satisfy; we wanted to see if QuickSpec could find them. In particular, we wanted to see if QuickSpec can discover that octonions are instances of an algebraic structure called a *Moufang loop*. Moufang loops are similar to groups, but not necessarily associative, and were invented by Ruth Moufang when studying geometry in planes with octonion coordinates (Moufang, 1935). Formally, a Moufang loop is a loop satisfying any one of four equivalent *Moufang identities*:

$$x*(y*(x*z)) = ((x*y)*x)*z \qquad \text{(M1)}$$

$$y*(x*(z*x)) = ((y*x)*z)*x \tag{M2}$$
$$(x*y)*(z*x) = (x*(y*z))*x \tag{M3}$$
$$(x*y)*(z*x) = x*((y*z)*x) \tag{M4}$$

Representation in Haskell. While complex numbers can be represented as pairs of real numbers and quaternions as quadruples of reals, the octonions can be thought of as 8-tuples of reals, as the name suggests. Perhaps more elegantly, by using the *Cayley-Dickson construction*, octonions can also be expressed as pairs of quaternions, just as quaternions can be represented as pairs of complex numbers and complex numbers as pairs of reals. We use this construction when modelling the octonions in Haskell and include operations for addition, multiplication and inverse.

We gave QuickSpec multiplication ($*$), inverse ($^{-1}$) and the constant 1, and asked it to produce terms up to size 7. In just 3 seconds it came back with the equations below:

$$1^{-1} = 1 \qquad \text{inverse of identity element (1)}$$
$$x*1 = x \qquad \text{right identity axiom (2)}$$
$$1*x = x \qquad \text{left identity axiom (3)}$$
$$(x^{-1})^{-1} = x \qquad \text{involution of inverse (4)}$$
$$x*x^{-1} = 1 \qquad \text{right inverse axiom (5)}$$
$$(x*x)*y = x*(x*y) \qquad \text{left alternative loop property (6)}$$
$$(x*y)*x = x*(y*x) \qquad \text{flexible loop property (7)}$$
$$(x*y)*y = x*(y*y) \qquad \text{right alternative loop property (8)}$$
$$y^{-1}*x^{-1} = (x*y)^{-1} \qquad \text{antiautomorphic inverse property (9)}$$
$$y*(y^{-1}*x) = x \qquad \text{left inverse property (10)}$$
$$(x*y)*(z*x) = x*((y*z)*x) \qquad \text{Moufang identity (11)}$$
$$(x*y)*(y*y) = x*(y*(y*y)) \qquad (12)$$
$$(x*(y*x))*z = x*(y*(x*z)) \qquad \text{left Bol loop property (13)}$$

Are these equations likely to be interesting to a mathematician? As the authors themselves are not mathematicians, we used the simple heuristic of assuming that mathematicians give interesting properties names. As it turns out, of the 13 properties QuickSpec found all except (12) have names.[13] Some of the properties discovered are basic axioms of quasigroups and loops in algebra, while others capture more exotic properties of the octonions. Equation (11) is the Moufang identity (M4), so QuickSpec does indeed discover that the octonions form a Moufang loop. The three equations (6–8) hint at the fact that multiplication of octonions is *diassociative*, meaning that any expression containing only two variables can be freely reassociated, a property which cannot be expressed equationally.

There is a small caveat to this case study: we model octonions as 8-tuples of real numbers, but we cannot use arbitrary real numbers for testing, as most real numbers are

[13] The names and descriptions of the properties can be found in for instance (Smith & Romanowska, 1999), `http://groupprops.subwiki.org/wiki/` or `https://en.wikipedia.org/wiki/Quasigroup`.

uncomputable. Instead, we test the laws on a subset of the octonions, namely 8-tuples of rational numbers. As a consequence, QuickSpec might suggest some false positives: equations which hold for all rational octonions but not for some octonions not in this subset. However, all the laws discovered in this case-study hold for all octonions. Furthermore, since the octonion operations are built only from continuous functions on the real numbers such as multiplication, one can prove that any equation that holds for rational octonions in fact holds for all octonions.

This case study shows that a tool like QuickSpec has the potential to be used as a mathematician's assistant, automatically exploring algebraic structures to suggest properties which could be interesting to prove, suggest areas of further investigation and perhaps even reveal unexpected connections between structures.

4.4 The Map data structure

In this case-study we explore QuickSpec's capabilities of dealing with conditional equations. The standard Haskell *Data.Map* library implements finite key-value maps. We asked QuickSpec to explore the following functions of the map library:

$$\begin{array}{ll} \mathit{empty} & :: \mathit{Ord}\ a \Rightarrow \mathit{Map}\ a\ b \\ \mathit{insert} & :: \mathit{Ord}\ a \Rightarrow a \to b \to \mathit{Map}\ a\ b \to \mathit{Map}\ a\ b \\ (\cup) & :: \mathit{Ord}\ a \Rightarrow \mathit{Map}\ a\ b \to \mathit{Map}\ a\ b \to \mathit{Map}\ a\ b \\ \mathit{lookup} & :: \mathit{Ord}\ a \Rightarrow a \to \mathit{Map}\ a\ b \to \mathit{Maybe}\ b \end{array}$$

Here are the equations that QuickSpec finds for $\cup$:

$$\begin{aligned} m \cup m &= m \\ m \cup \mathit{empty} &= m \\ \mathit{empty} \cup m &= m \\ m \cup (m \cup n) &= m \cup n \\ m \cup (n \cup m) &= m \cup n \\ (m_1 \cup m_2) \cup m_3 &= m_1 \cup (m_2 \cup m_3) \end{aligned}$$

We see that $\cup$ is idempotent and associative and has an identity element, as we expect. What is surprising is that $\cup$ is not commutative. In fact, it cannot be commutative, as when the maps given to $\cup$ have a key in common but with different values, one of the maps must take precedence. The fourth and fifth equations follow from idempotence and associativity, but QuickSpec generates them anyway. This is because they are regarded as being simpler than associativity.

We find the following equations about *insert*:

$$\begin{aligned} \mathit{insert}\ i\ a\ m \cup m &= \mathit{insert}\ i\ a\ m \\ \mathit{insert}\ i\ a\ (m \cup n) &= \mathit{insert}\ i\ a\ m \cup n \\ \mathit{insert}\ i\ a\ (\mathit{insert}\ i\ a\ m) &= \mathit{insert}\ i\ a\ m \\ \mathit{insert}\ i\ a\ (\mathit{insert}\ j\ a\ m) &= \mathit{insert}\ j\ a\ (\mathit{insert}\ i\ a\ m) \end{aligned}$$

The second equation indicates that $\cup$ is left-biased, and the third shows that *insert* is idempotent. The last equation indicates that the order we insert keys makes no difference,

but it is not as general as we would like: the equation says that we must use the same data value a for both insertions. The reader might like to think about why the more general law

$$insert\ i\ a\ (insert\ j\ b\ m) = insert\ j\ b\ (insert\ i\ a\ m)$$

is not found.

Finally, we find the following equations for *lookup*:

$$\begin{aligned} lookup\ i\ empty &= Nothing \\ lookup\ i\ (insert\ i\ a\ m) &= Just\ a \\ lookup\ i\ (insert\ j\ a\ empty) &= lookup\ j\ (insert\ i\ a\ empty) \end{aligned}$$

The first two are what we would expect. The last equation is a bit unsatisfying: it requires the argument to *insert* to be *empty*, and we would have liked to see a more general version of the equation.

Summing up, for both *insert* and *lookup*, QuickSpec found overly specific laws. The reason is that *the generalisations of these laws only hold conditionally*. Perhaps we can find them using QuickSpec's extension for conditional equations (Section 3.8)?

If we specify $i \neq j$ as a condition predicate, QuickSpec generates the following equations:

$$\begin{aligned} &i \neq j \Rightarrow insert\ i\ a\ (insert\ j\ b\ m) = insert\ j\ b\ (insert\ i\ a\ m) \\ &i \neq j \Rightarrow lookup\ i\ (insert\ j\ a\ m) = lookup\ i\ m \end{aligned}$$

The first law generalises our previous *insert* law. This law reveals why the generalisation we suggested earlier was not found: it is incorrect! If $i = j$ then the order of insertion *does* matter, as the second insertion will overwrite the first.

The second law allows us to show our law from above,

$$lookup\ i\ (insert\ j\ a\ empty) = lookup\ j\ (insert\ i\ a\ empty),$$

by case analysis on whether $i = j$.

Coming up with the correct condition predicates is not always easy. In particular, placing several conditions on the same variable is something which is difficult in the current design, because of the way conditions are represented as types. There is clearly more future work to be done here; we would like QuickSpec to combine condition predicates freely, just as it does with functions when building terms.

4.5 Performance Analysis

We have demonstrated how QuickSpec can help the user to discover interesting laws about their programs. In this section we measure how expensive that process is, and use the results to suggest ways to speed up QuickSpec further.

We ran QuickSpec on several examples, and measured the runtime, the memory use, the total number of terms generated, and the total number of test cases evaluated.[14] We then judged these numbers in three different ways.

[14] All tests were run on a laptop with a 2.6GHz Intel i7-5600U processor and 12GB of RAM running 64-bit Linux.

First, we ran the same examples on QuickSpec 1, the previous state of the art, and compared the numbers. The results are found in Table 2. QuickSpec 1 ran out of memory on all of our examples except for the octonions, so we used that as an example. We also took the pretty-printing example, and recorded the numbers at the point where QuickSpec 1 ran out of memory; we estimate that its numbers would have been perhaps 3 times higher had it run to completion.

QuickSpec 2 is indeed much faster than QuickSpec 1, running the two examples in seconds rather than minutes. Furthermore, the memory use does not get out of control. We can see that the number of test cases is reduced much more dramatically than the number of terms; this is because QuickSpec 2 cuts out redundant terms, and as Section 3.2 notes, these are the ones which are most expensive to test.

	Number of terms			**Number of tests**		
	QS1	QS2	Reduction	QS1	QS2	Reduction
Pretty-printing	110 000	2 800	**39** ×	75 000 000	24 000	**3 000** ×
Octonions	925	871	**1.06** ×	780 000	13 900	**56** ×
	Execution time (s)			**Memory use (MB)**		
	QS1	QS2	Reduction	QS1	QS2	Reduction
Pretty-printing	1 200	7	**170** ×	12 000	137	**87** ×
Octonions	1 800	30	**60** ×	110	116	**0.95** ×

Table 2: Performance of QuickSpec 1 versus QuickSpec 2. Pretty-printing numbers for QuickSpec 1 are taken at the time that it ran out of memory.

Next, we took the laws that QuickSpec discovered and tested them using Quick*Check*, again measuring the runtime and the total number of test cases evaluated. By taking the ratio of QuickSpec's runtime to QuickCheck's, we can find the *overhead* of discovering the laws compared to simply testing them. An overhead of 1 represents a perfect theory exploration system which is able to discover all conjectures without making any wrong guesses; we should not expect to get an overhead of 1. The results are found in Table 3.

We measured the overhead both in terms of how many test cases QuickSpec executed and how much time it took. The number of tests tells us how long QuickSpec would take if we implemented its algorithms with perfect efficiency: how much time is spent in actually testing the program?

We will start by looking at the testing overhead. The octonions immediately stand out: QuickSpec requires only 6% more tests to discover the 13 laws compared to testing them with QuickCheck, cheap indeed. One reason is that in this domain it is rather easy to find values which separate terms which are not equal. In fact, a single test case was enough to separate all candidate terms!

For the functional geometry example (here including *over*, *above*, *beside*, *rot*, *flip*, *quartet* and *cycle* and exploring up to size 7) we get the largest overhead. One reason for this is that

	Laws	**Number of tests**			**Execution time (s)**		
		QC	QS	Overhead	QC	QS	Overhead
Pretty-printing	16	16 000	24 000	**1.5** ×	1.2s	7.1s	**5.9** ×
Functional Geometry	18	18 000	68 000	**3.8** ×	1.1s	76s	**69** ×
Large List Library	398	398 000	874 000	**2.2** ×		2200s	
Octonions	13	13 000	13 900	**1.06** ×	25s	33s	**1.32** ×

Table 3: Performance of QuickSpec versus QuickCheck. We configured both to try each law on 1000 test cases (the default for QuickSpec). As we had to manually translate all of the discovered laws into QuickCheck properties, we did not run QuickCheck on the large list library.

QuickSpec's schema instantiation is inefficient for idempotent functions such as *over*: if a schema contains *over* ? ?, its one-variable instance will contain *over x x*; this is equal to *x*, and so we will always instantiate the schema. Indeed, when we remove *over* from the signature, the number of terms falls by a factor of 5.

For the pretty-printing example, QuickSpec needs 50% more tests than QuickCheck, and for the large list library, it needs just over twice as many. In all four examples, the testing overhead for discovering the laws is quite small.

On the other hand, the time overhead is higher than the testing overhead, sometimes considerably. This implies that most of the time is not spent testing the program, but in QuickSpec itself. Time spent pruning is a necessary overhead, but none of the examples spend a substantial amount of time in pruning. Rather, we noticed several possible bottlenecks:

- We save work by throwing away most schemas without generating their instances. However, once we discover a schema equation we generate all of its instances, which is wasteful, and leads to the performance problem we saw above with idempotent functions like *over*.
  It would be better to instantiate each schema gradually so that, after discovering a one-variable equation, we try to generalise it to a two-variable equation, then a three-variable equation, and so on. The number of instances of an n-variable schema is n^n, but the number of two-variable instances is only 2^n, so this idea seems promising. For *over*, we would fail to generate any two-variable laws and then stop.
- In the huge lists example, QuickSpec ends up spending most of its time renormalising the candidate terms after having discovered a new law (i.e. on the penultimate line of the function *consider* on page 12). This is simply because there are a lot of candidate terms. It would be better to normalise the candidate terms lazily: when the decision tree gives us an equation to be QuickChecked, normalise both sides of the equation first in case they become equal, but don't otherwise normalise the candidate terms.
- The implementation itself is not yet highly-optimised. For example, after QuickSpec explores a background theory it throws away the decision tree, only remembering the

discovered laws. This leads to duplicated work when exploring the main signature, which is quite noticeable when running the geometry example.

As a final comparison, we measured how much memory QuickSpec used on our four examples and how many terms it generated. The results are found in Table 4.

All the examples (including the large list library) fit comfortably in the test computer's memory. However, the geometry and large list examples are quite memory-hungry. Most of that memory is consumed by the decision tree.

The decision tree is not large in nodes—the number of nodes is one more than the number of generated terms. The culprit is the test *results* that are stored in the edges of the tree. In particular, the geometry combinators and some of the list combinators return large results. The need to store tens of thousands of test results is what mostly determines memory use.

While analysing these numbers, we hit upon a simple memory-saving idea: instead of storing test results in the decision tree, only store their *hashes* and forget the results themselves. When we implemented this idea, the geometry example used a factor of 10 less memory. We plan to provide this feature in a future release of QuickSpec.

	Number of terms	**Memory use (MB)**
Pretty-printing	2800	137
Functional Geometry	23000	3093
Large List Library	52000	4627
Octonions	871	116

Table 4: Memory use of QuickSpec.

Summary Our performance analysis shows that QuickSpec is far ahead of previous theory exploration systems, and that it is very good at discovering laws using only a small amount of testing, suggesting that its performance could in theory be competitive with QuickCheck—discovering the laws need not be much more expensive than testing them.

On the other hand, it is also clear that the implementation currently lags slightly behind the design, although we were happy with its performance while running the case studies ourselves. We found several potential bottlenecks in the implementation and suggested ways to fix them which, we hope, will make QuickSpec even faster in future.

4.6 Lessons learned

Finally, we would like to present some heuristics for using QuickSpec effectively, which we found while carrying out the case studies. These recommendations are based on our practical experience with the tool so far, and should just be taken as a starting point for users wishing to experiment with QuickSpec on their own.

While QuickSpec can cope with large signatures of 25 functions or more (such as in the list library case study), the user might be the bottleneck. With such large signatures

QuickSpec can produce many laws and it will be quite difficult to understand the output. In this case, it is better to focus in on smaller parts of the signature and run QuickSpec several times on different combinations of functions. As a rule of thumb, a good starting point seems to be a signature consisting of up to 10 functions, but this varies from theory to theory. It is furthermore unlikely (in most cases) that any interesting property will feature more than a handful of different functions.

As for maximum term size, a good heuristic seems to be around 7–9, as this seems to cover plenty of interesting and useful properties; most interesting properties seem to be relatively small. Again, this may vary from theory to theory.

Finally, it is often important to add auxiliary functions to the signature, as in the pretty-printing example. The nature of these functions varies a lot from theory to theory, but functions connecting your datatypes to well-understood datatypes like lists often lead to interesting laws.

5 Related work

The first version of QuickSpec, QuickSpec 1 (Claessen *et al.*, 2010), already performed better than other comparable theory exploration systems like IsaCoSy and IsaScheme (Johansson *et al.*, 2011; Montano-Rivas *et al.*, 2012). We begin with a performance comparison on some of the examples in previous sections between our new version, QuickSpec 2, with the old QuickSpec 1, before discussing other related work.

5.1 Comparison to QuickSpec 1

QuickSpec 1 does not interleave testing and pruning: first it tests terms to discover equations, then it prunes the equations.[15] This causes it to perform a lot of needless testing. Furthermore, it does not use schemas. This increases the number of terms generated, and means that the user has to specify how many variables of each type to try. QuickSpec 2 uses as many variables as needed. Finally, QuickSpec 1 does not support polymorphism: the user must manually instantiate each polymorphic function at all the types required, and the system does not know that these instances are related, leading to the same law being found at several different types.

When running QuickSpec 1, the user must specify both a depth limit and a size limit. It then enumerates terms in order of depth, filtering out the ones that are too large.

Pretty Printing. Looking back to our initial example of Hughes's pretty printing library from Section 2, we already notice a huge performance improvement: while QuickSpec 2 only needs six seconds, QuickSpec 1 runs tests for about 20 minutes before running out of memory. QuickSpec 2 found 16 laws and ran about 25 500 tests for terms up to size 9. QuickSpec 1 only reached terms up to depth 4 before running out of memory, meaning that it was not close to finding all the laws that QuickSpec 2 found, the biggest of which has a term of depth 5.

[15] It is more refined than SlowSpec, because it enumerates terms rather than equations.

Functional Geometry. We tried QuickSpec 1 on the combinators *over*, *above*, *beside*, *rot*, *flip*, *quartet* and *cycle* (see Section 4.1), exploring up to size 7. After 2 hours, QuickSpec 1 was still testing and we gave up. QuickSpec 2, on the other hand takes just 25 seconds on the same example, generates 18 laws, and runs a total of 68 000 tests.

The Large List Library. Giving QuickSpec 1 the large list library signature from Section 4.2 is totally out of the question: we estimate it would generate millions of terms and need to evaluate billions of values.

The Octonions. On the octonions (see Section 4.3), we ran both QuickSpec 1 and QuickSpec 2 up to size 7, and this is the only example where also QuickSpec 1 coped. However, the difference in runtime is huge: 3 seconds for QuickSpec 2 compared to 15 minutes for QuickSpec 1. QuickSpec 1 also finds three extra laws that QuickSpec 2 managed to prune away:

$$(x*y)*(y^{-1}) = x \tag{1}$$

$$x^{-1}*(x*y) = y \tag{2}$$

$$x^{-1}*(y^{-1}*x) = (y*x)^{-1}*x \tag{3}$$

The order in which QuickSpec 2 explores the theory, combined with a new more powerful pruner, allows it to reduce the number of dull laws presented to the user.

5.2 Other Theory Exploration systems

QuickSpec shares some features with several previous theory exploration systems, although these have mainly been used in the context of lemma discovery for theorem proving, rather than program understanding. While QuickSpec can be used either as a light-weight verification tool using testing, or connected to a theorem prover to generate candidate lemmas, most other theory exploration systems have focused only on the latter. IsaCoSy (Johansson *et al.*, 2011) and IsaScheme (Montano-Rivas *et al.*, 2012) are theory exploration systems for the proof assistant Isabelle/HOL. These are predecessors to our system Hipster (Johansson *et al.*, 2014), which combines QuickSpec with an inductive theorem prover i Isabelle/HOL. Hence, the proving power of IsaCoSy, IsaScheme and Hipster/QuickSpec is similar, but they differ in how effectively they discover relevant lemmas. IsaCoSy and IsaScheme are able to discover relevant lemmas about standard recursive library functions with high precision, but are slower than even the first version of QuickSpec. The major flaw in these systems is that they enumerate and test equations, and not terms like QuickSpec (Section 3.2.2). In IsaScheme the user provides the system with schemes, which are higher-order templates with holes that the system instantiates. These are reminiscent of our schemas, except that QuickSpec discovers the schemas automatically, and only instantiates holes with variables. IsaCoSy only generates terms that are irreducible, given the equations discovered so far. QuickSpec uses a similar approach as it discards any term which can be proved equal to one previously seen. IsaCoSy also supports polymorphism, but without a heuristic to limit polymorphic instances, so it spends a lot of time exploring silly instances.

MATHsAiD is a theorem discovery tool aimed at helping mathematicians to discover the standard lemmas and theorems in new theory developments (Bundy *et al.*, 2015), similarly

to how we used QuickSpec in our case study about the octonions in Section 4.3. MATHsAiD has successfully been used to automatically (re)discover some recently published theorems in the theory of Zariski spaces. MATHsAiD generates theorems using many heuristics. It starts by picking a main predicate (e.g. equality) for the new conjecture along with a so-called *term of interest*. These are combined to form a *conjecture shell*, which contains some holes that are filled in by applying forward reasoning guided by heuristic rules. MATHsAiD's terms of interest are similar to QuickSpec's schemas. However, MATHsAiD also has a set of pre-defined "meta-schemas" which contains holes for functions. Terms of interest are generated by instantiating these meta-schemas with available functions. QuickSpec does not rely on such meta-schemas, and does not have any restrictions on the shape of terms it can generate.

Daikon is a tool for automatic discovery of invariants, preconditions and postconditions for imperative programming languages such as Java, C, C++ and Perl (Ernst *et al.*, 2007). Daikon is provided with a grammar describing pre-defined invariant patterns and returns those that are observed to hold as the program is executed. Daikon can also discover conditional invariants, where the condition is a pre-defined predicate provided by the user. Daikon thus discovers invariants which hold at particular program points, essentially assertions which could be inserted in the code. QuickSpec, on the other hand, discovers properties based on the program API. The Daikon approach is tailored towards imperative programs while QuickSpec is more appropriate for analysing pure functions.

6 Conclusion

We have presented QuickSpec, a tool for automatically finding equational laws by testing. We believe that QuickSpec can lower the barrier to entry of formal specifications for functional programmers. As we have showed in our case studies and examples, the discovered specifications are a great way of understanding programs and can even reveal bugs. For Hughes's pretty printing library, described in section 2, we showed that QuickSpec not only automatically found all the laws in the original specification, but also led us to a more general specification through a new auxiliary function. In the case study on Henderson's functional geometry library (section 4.1), we showed how unexpected laws discovered by QuickSpec can reveal bugs in the implementation.

QuickSpec is not only a useful tool for programmers, but is used in several theorem proving applications, in particular for inductive proofs. QuickSpec is used to discover basic lemmas which are required for proving more complex conjectures in two state-of-the-art inductive theorem provers called HipSpec and Hipster (Claessen *et al.*, 2013; Johansson *et al.*, 2014).

The problem of discovering equations without extreme exponential blowup is a challenging one. In QuickSpec, we solve it by efficiently identifying and pruning uninteresting parts of the search space, partly through schemas and partly through not testing redundant terms. This, combined with efficient testing and pruning infrastructure, means that QuickSpec now can deal with realistic programs and find satisfying and complex equational laws in a matter of seconds.

A Source code for pretty-printing example

For reference, we give here the complete source code of the pretty-printing example, both the implementation and the QuickSpec declarations.

```
{-# LANGUAGE DeriveDataTypeable, TypeOperators #-}
import Control.Monad
import Test.QuickCheck
import QuickSpec hiding (background, (◇), text, nest, ($$))
```

First we define the type of layouts, closely following Hughes's model implementation. A layout is a list of lines, each of which is represented as a (indentation level, text) pair:

```
newtype Layout = Layout [(Int, String)]
  deriving (Typeable, Eq, Ord, Show)
```

We define an *Arbitrary* instance for generating random documents, which simply generates a random nonempty list:

```
instance Arbitrary Layout where
  arbitrary = do
    NonEmpty lines ← arbitrary
    return (Layout lines)
```

The definitions of the combinators follow Hughes exactly:

```
text :: String → Layout
text s = Layout [(0, s)]

nest :: Int → Layout → Layout
nest k (Layout l) = Layout [(i + k, s) | (i, s) ← l]

($$) :: Layout → Layout → Layout
Layout xs $$ Layout ys = Layout (xs ++ ys)

(◇) :: Layout → Layout → Layout
Layout xs ◇ Layout ys =
  combine (init xs) (last xs) (head ys) (tail ys)
  where
    combine xs (i, s) (j, t) ys =
      Layout xs $$
      Layout [(i, s ++ t)] $$
      nest (i + length s − j) (Layout ys)
```

The *nesting* function is our own addition to the pretty-printing API, described in the text:

```
nesting :: Layout → Int
nesting (Layout ((i, _) : _)) = i
```

Next we write down the signatures for QuickSpec. We start with a signature which contains all the background functions. For the purposes of this example, we tell QuickSpec to explore all terms up to size 9.

```
background =
  signature {
    maxTermSize = Just 9,
    constants = [
      constant "\"\"" "",    -- the empty string ""
      constant "++" ((++) :: String → String → String),
      constant "0" (0 :: Int),
      constant "+" ((+) :: Int → Int → Int),
      constant "length" (length :: String → Int)] }
```

Next, we define a signature for the pretty-printing combinators themselves. The final line of the signature is QuickSpec's syntax for declaring the *Layout* type.

```
sig =
  signature {
    constants = [
      constant "text" text,
      constant "nest" nest,
      constant "nesting" nesting,
      constant "$$" ($$),
      constant "<>" (◇)],
    instances = [baseType (⊥ :: Layout)] }
```

Finally, we run QuickSpec like so:

```
main = quickSpecWithBackground background sig
```

The full output of QuickSpec, which takes 10 seconds to run on the computer described in Section 4.5, is reproduced below.

```
== Signature ==
    "" :: [Char]
  (++) :: [Char] -> [Char] -> [Char]
     0 :: Int
   (+) :: Int -> Int -> Int
length :: [Char] -> Int

== Laws ==
  1. length "" = 0
  2. x + 0 = x
  3. 0 + x = x
  4. xs ++ "" = xs
  5. "" ++ xs = xs
  6. x + y = y + x
  7. length (xs ++ ys) = length (ys ++ xs)
  8. (x + y) + z = x + (y + z)
  9. (xs ++ ys) ++ zs = xs ++ (ys ++ zs)
 10. length xs + length ys = length (xs ++ ys)
```

```
== Signature ==
   text :: [Char] -> Layout
   nest :: Int -> Layout -> Layout
nesting :: Layout -> Int
   ($$) :: Layout -> Layout -> Layout
   (<>) :: Layout -> Layout -> Layout

== Laws ==
  1. nest 0 x = x
  2. nesting (text xs) = 0
  3. nesting (x $$ y) = nesting x
  4. nesting (x <> y) = nesting x
  5. nesting (nest x y) = x + nesting y
  6. x <> text "" = x
  7. (x $$ y) $$ z = x $$ (y $$ z)
  8. x <> nest z y = x <> y
  9. (x $$ y) <> z = x $$ (y <> z)
 10. (x <> y) <> z = x <> (y <> z)
 11. nest x (y <> z) = nest x y <> z
 12. nest (x + y) z = nest x (nest y z)
 13. text xs <> text ys = text (xs ++ ys)
 14. nest x y $$ nest x z = nest x (y $$ z)
 15. nest (nesting x) (text "") <> x = x
 16. text xs <> (text "" $$ x) = text xs $$ nest (length xs) x
 17. text (xs ++ ys) $$ nest (length xs) x =
       text xs <> (text ys $$ x)
 18. text xs <> (x $$ nest (nesting x) y) =
       (text xs <> x) $$ nest (length xs) y
 19. (text xs <> x) $$ (text "" <> x) =
       text xs <> (nest (length xs) x $$ x)
 20. text "" <> ((text xs <> x) $$ y) = (text xs <> x) $$ y
```

References

Bachmair, Leo, Dershowitz, Nachum, & Plaisted, David A. (1989). *Completion without failure*.

Baez, John C. (2002). The octonions. *Bulletin of the american mathematical society*, **39**, 145–205.

Buchberger, Bruno, Creciun, Adrian, Jebelean, Tudor, Kovacs, Laura, Kutsia, Temur, Nakagawa, Koji, Piroi, Florina, Popov, Nikolaj, Robu, Judit, Rosenkranz, Markus, & Windsteiger, Wolfgang. (2006). Theorema: Towards computer-aided mathematical theory exploration. *Journal of applied logic*, **4**(4), 470 – 504. Towards Computer Aided Mathematics.

Bundy, Alan, McCasland, Roy, & Smith, Patrick. (2015). *MATHsAiD: Automated mathematical theory exploration*. Working paper. University of Edinburgh.

Claessen, Koen, & Hughes, John. (2000). QuickCheck: a lightweight tool for random testing of Haskell programs. *Pages 268–279 of: Proceedings of ICFP*.

Claessen, Koen, Smallbone, Nicholas, & Hughes, John. (2010). QuickSpec: Guessing formal specifications using testing. *Pages 6–21 of: Proceedings of TAP*.

Claessen, Koen, Johansson, Moa, Rosén, Dan, & Smallbone, Nicholas. (2013). Automating inductive proofs using theory exploration. *Pages 392–406 of: Proceedings of the Conference on Auomtated Deduction (CADE)*. LNCS, vol. 7898. Springer.

Claessen, Koen, Duregård, Jonas, & Pałka, Michał H. (2014). Generating constrained random data with uniform distribution. *Pages 18–34 of: International Symposium on Functional and Logic Programming*. Springer.

Duregård, Jonas. (2016). *Automating black-box property based testing*. Ph.D. thesis, Chalmers University of Technology.

Ernst, Michael D., Perkins, Jeff H., Guo, Philip J., McCamant, Stephen, Pacheco, Carlos, Tschantz, Matthew S., & Xiao, Chen. (2007). The Daikon system for dynamic detection of likely invariants. *Science of computer programming*, **69**(1–3), 35–45.

Henderson, Peter. (1982). Functional geometry. *Pages 179–187 of: Symposium on LISP and Functional Programming*.

Henderson, Peter. (2002). Functional geometry. *Higher-order and symbolic computation*, **15**(4), 349–365.

Hughes, John. (1995). The design of a pretty-printing library. *Pages 53–96 of: Advanced Functional Programming*. Springer Verlag.

Johansson, Moa, Dixon, Lucas, & Bundy, Alan. (2011). Conjecture synthesis for inductive theories. *Journal of automated reasoning*, **47**(3), 251–289.

Johansson, Moa, Rosén, Dan, Smallbone, Nicholas, & Claessen, Koen. (2014). Hipster: Integrating theory exploration in a proof assistant. *Conference on Intelligent Computer Mathematics*.

Knuth, D.E., & Bendix, P.B. (1983). Simple word problems in universal algebras. *Pages 342–376 of:* Siekmann, JrgH., & Wrightson, Graham (eds), *Automation of Reasoning*. Symbolic Computation. Springer Berlin Heidelberg.

Martin, Ursula, & Nipkow, Tobias. (1990). Ordered rewriting and confluence. *Pages 366–380 of:* Stickel, M.E. (ed), *Proc. 10th Int. Conf. Automated Deduction*, vol. 449.

Montano-Rivas, Omar, McCasland, Roy, Dixon, Lucas, & Bundy, Alan. (2012). Scheme-based theorem discovery and concept invention. *Expert systems with applications*, **39**(2), 1637–1646.

Moufang, Ruth. (1935). Zur struktur von alternativekörpen. *Mathematische annalen*, **110**, 416–430.

Smith, J.D.H., & Romanowska, Anna B. (1999). *Post-modern algebra*. Wiley-Interscience.