Proceedings Paper:
Braquehais, Rudy and Runciman, Colin orcid.org/0000-0002-0151-3233 (2017) Speculate: discovering conditional equations and inequalities about black-box functions by reasoning from test results. In: Proceedings of the ACM SIGPLAN Haskell Symposium 2017. ACM , pp. 40-51.

https://doi.org/10.1145/3122955.3122961

Speculate: Discovering Conditional Equations and Inequalities about Black-Box Functions by Reasoning from Test Results

Rudy Braquehais
University of York
rmb532@york.ac.uk

Colin Runciman
University of York
colin.runciman@york.ac.uk

Abstract

This paper presents Speculate, a tool that automatically conjectures laws involving conditional equations and inequalities about Haskell functions. Speculate enumerates expressions involving a given collection of Haskell functions, testing to separate those expressions into apparent equivalence classes. Expressions in the same equivalence class are used to conjecture equations. Representative expressions of different equivalence classes are used to conjecture conditional equations and inequalities. Speculate uses lightweight equational reasoning based on term rewriting to discard redundant laws and to avoid needless testing. Several applications demonstrate the effectiveness of Speculate.

CCS Concepts •**Software and its engineering** →**Software testing and debugging;** •**Theory of computation** →Program specifications;

Keywords formal specification, property-based testing, Haskell.

ACM Reference format:
Rudy Braquehais and Colin Runciman. 2017. Speculate: Discovering Conditional Equations and Inequalities about Black-Box Functions by Reasoning from Test Results. In *Proceedings of Haskell'17, Oxford, United Kingdom, September 7–8, 2017,* 13 pages.
DOI: 10.1145/3122955.3122961

1 Introduction

Writing formal specifications for programs is hard, but nevertheless useful. Formally specifying a program can contribute to understanding, documentation, and regression testing using a tool like QuickCheck [6].

This paper presents a new tool called Speculate. Given a collection of Haskell functions and values bound to monomorphic types, Speculate automatically conjectures a specification containing equations *and inequalities* involving those functions. Both equations and inequalities may be *conditional*. In these respects we extend previous work by other researchers on discovering unconditional equations [8, 22]. As Speculate is based on testing, its results are speculative.

Speculate enumerates expressions by combining free variables, functions and values provided by the user (§3). It evaluates these expressions for automatically generated test cases to partition the expressions into apparent equivalence classes. It conjectures equations between expressions in the same equivalence class. Then, it conjectures conditional equations (⇒) and inequalities (≤) from representatives of different equivalence classes (§4). Speculate uses lightweight equational reasoning to discard redundant equations and to avoid needless testing. Speculate is implemented in Haskell.

Example 1.1. When provided with the integer values `0` and `1`, the functions `id` and `abs`, and the addition operator (`+`), Speculate first discovers and prints the following apparent equations:

```
id x == x
x + 0 == x
abs (abs x) == abs x
x + y == y + x
abs (x + x) == abs x + abs x
abs (x + abs x) == x + abs x
abs (1 + abs x) == 1 + abs x
(x + y) + z == x + (y + z)
```

Similar equational laws are found by the existing tool QuickSpec [8, 22]. But Speculate goes on to print the following apparent inequalities:

```
x <= abs x
0 <= abs x
x <= x + 1
x <= x + abs y
x <= abs (x + x)
x <= 1 + abs x
0 <= x + abs x
x + y <= x + abs y
abs (x + 1) <= 1 + abs x
```

Finally, it prints these apparent conditional laws:

```
   x <= y  ==>  x <= abs y
abs x <= y  ==>  x <= y
 abs x < y  ==>  x <  y
   x <= 0  ==>  x <= abs y
abs x <= y  ==>  0 <= y
 abs x < y  ==>  1 <= y
   x == 1  ==>  1 == abs x
    x < 0  ==>  1 <= abs x
   y <= x  ==>  abs (x + abs y) == x + abs y
   x <= 0  ==>        x + abs x == 0
abs x <= y  ==>      abs (x + y) == x + y
abs y <= x  ==>      abs (x + y) == x + y
```

The total execution time for Speculate to generate all the above laws is about 3 seconds. Speculate is implemented as a library, and the total application-specific source code required for this example is less than 10 lines. □

Haskell'17, Oxford, United Kingdom

978-1-4503-5182-9/17/09…$15.00
DOI: 10.1145/3122955.3122961

1.1 Contributions.

The main contributions of this paper are:

1. methods using automated black-box testing and equational reasoning to discover apparent conditional equations and inequalities between functional expressions;
2. the design of the Speculate tool, which implements these methods in Haskell and for Haskell functions;
3. a selection of small case-studies, investigating the effectiveness of Speculate.

1.2 Road-map.

The paper is organized as follows: §2 defines expressions, expression size and a complexity ordering on expressions; §3 describes how to use Speculate; §4 describes how Speculate works internally; §5 presents example applications and results; §6 discusses related work; §7 draws conclusions and suggests future work.

2 Definitions

Expressions and their sizes All expressions formed by Speculate have monomorphic types. Expressions, and their sizes, are:

Constants constant data-value and function symbols of size 1, e.g.,

- `0 :: Int`,
- `'a' :: Char`,
- `(+) :: Int -> Int -> Int`

Variables variable symbols, also of size 1, such as

- `x :: Int`,
- `f :: Int -> Int`;

Applications type-correct applications of functional expressions to one or more argument expressions, including partial applications, such as

- `id y :: Int` of size 2,
- `(1+) :: Int -> Int` of size 2,
- `x + (y + 0) :: Int` of size 5.

The size of an application is the number of constant and variable symbols it contains.

To avoid an explosive increase in the search-space, we do not include other forms of Haskell expression such as lambda expressions or case expressions.

A complexity ordering on expressions When there is redundancy between laws, Speculate has to decide which to keep and which to discard. As a general rule, it *keeps the simplest laws*. It also presents final sets of laws in order of increasing complexity. An expression e_1 is *strictly simpler* than another expression e_2, if the first of the following conditions to distinguish between them is:

1. e_1 is *smaller in size than* e_2,
   e.g.: `x + y < x + (y + z)`;
2. or, e_1 has *more distinct variables* than e_2,
   e.g.: `x + y < x + x`;
3. or, e_1 has *more variable occurrences* than e_2,
   e.g.: `x + x < 1 + x`;
4. or, e_1 has *fewer distinct constants* than e_2,
   e.g.: `1 + 1 < 0 + 1`;
5. or, e_1 *precedes* e_2 *lexicographically*,
   e.g.: `x + y < y + z`.

A similar ordering is used in QuickSpec [8, 22].

```
import Test.Speculate

main :: IO ()
main = speculate args
  { constants =
      [ constant "+"   ((+)  :: Int -> Int -> Int)
      , constant "id"  (id   :: Int -> Int)
      , constant "abs" (abs  :: Int -> Int)
      , background
      , constant "0"   (0 :: Int)
      , constant "1"   (1 :: Int)
      , constant "<="  ((<=) :: Int -> Int -> Bool)
      , constant "<"   ((<)  :: Int -> Int -> Bool)
      ]
  }
```

Figure 1. Program used to obtain the results in §1.

3 How Speculate is Used

Speculate is used as a library (by "`import Test.Speculate`"). Unless they already exist, instances of the `Listable` typeclass [4] are declared for needed user-defined datatypes (step 1). Constant values and functions are gathered in an appropriately formulated list, and passed to the `speculate` function (step 2).

1. Provide typeclass instances for used-defined types Speculate needs to know how to enumerate values to test equality between expressions. So, where necessary, we declare type-class instances for user-defined types. Speculate provides instances for most standard Haskell types and a facility to derive instances for user-defined data types using Template Haskell [19]. Writing

```
deriveListable ''<Type>
```

is enough to create the necessary instances. See [4] for how to define such instances manually, and why that is desirable in some cases.

Then, to provide the instance information to Speculate, for two types named `Type1` and `Type2`, write the following:

```
instances = [ ins "x" (undefined :: Type1)
            , ins "i" (undefined :: Type2) ]
```

2. Call the `speculate` function Constant values and functions are gathered in a record of type `Speculate.Args` and passed to the `speculate` function. Constants we want to know laws about are included in an `Args` field, the `constants` list. Other constants that appear in laws, but not as the primary subjects, are those occurring in the `constants` list after the special constant `background`.

Example 1.1 (revisited). Figure 1 shows the program used to obtain the results in §1. □

Speculate limits the size of expressions considered, and the number of test cases used. By default it:

- considers expressions up to size 5;
- considers inequalities between expressions up to size 4;
- considers conditions up to size 4;
- tests candidate laws for up to 500 value assignments.

The `speculate` function allows variations of these default settings either by setting `Args` fields or in command line arguments.

4 How Speculate Works

In summary, Speculate works by enumerating expressions and evaluating test instances of them. In order for that to work effectively, Speculate uses equational reasoning (§4.1). Speculate determines, in the following order, apparent:

1. equations and equivalence classes of expressions (§4.2);
2. inequalities (§4.3);
3. conditional equations (§4.4).

To encapsulate values of different types, Speculate uses the `Data.Dynamic` module [1] provided with GHC [23] and declares a type to encode Haskell expressions.

4.1 Equational Reasoning based on Term Rewriting

Following QuickSpec [22], Speculate performs basic equational reasoning based on *unfailing Knuth-Bendix Completion* [3, 13]. The aims are to prune the search space avoiding needless testing, and to filter redundant laws so that the output is more useful to the user.

Completion The *Knuth-Bendix Completion* procedure takes a set of equations and produces a *confluent term rewriting system* [2, 13]: a set of rewrite rules that can be used to simplify, or *normalize*, expressions. To check if two expressions are equal, we can check if their *normal forms* are the same. The completion procedure has two problems: failure in the presence of *unorientable equations* and possible *non-termination*. Speculate solves these problems similarly to QuickSpec as detailed in the following paragraphs.

Unorientable equations To deal with unorientable equations, we use the technique of *unfailing completion* [3] which allows unorientable equations to be kept in a separate set from rules. Checking for equivalence using normalization is still sound, but incomplete (the fact that two expressions are equivalent may be undetected). We can use unorientable equations to improve the check for equivalence between expressions e_1 and e_2: first normalize both e_1 and e_2; then take the equivalence closure using the set of unorientable equations; finally, if one of the expressions in the closure of e_1 is equivalent to one of the expressions in the closure of e_2 then they are equivalent. To ensure termination, we impose a configurable bound on the number of closure applications.

Non-termination To deal with non-termination of the completion procedure, we impose a limit on the size of generated rules, discarding any rules where the left-hand size is bigger than the maximum expression size we are exploring.

4.2 Equations and Equivalence Classes of Expressions

Speculate finds equations in a similar way to QuickSpec 2 [22]. As QuickSpec 2 has many features, like support for polymorphism, use of external theorem provers for reasoning and several configuration options, we chose to reimplement a core variant before extending it with support for conditional equations and inequalities. Differences to QuickSpec are highlighted in §6.

This section summarizes how Speculate finds equations.

State Speculate processes each expression in turn, transforming a state. Speculate keeps track of:

- a theory (§4.1) based on equations discovered so far;
- a set of equivalence classes of all expressions considered so far, and for each of them a smallest representative.

Table 1. Equivalence classes and equations after initialization by considering all expressions of size 1.

equivalence classes			equations
type	repr.	others	
`Int`	`x`	—	no equations
`Int`	`0`	—	
`Int`	`1`	—	
`Int -> Int`	`id`	—	
`Int -> Int`	`abs`	—	
`Int -> Int -> Int`	`(+)`	—	

Table 2. Equivalence classes and equations after considering all expressions of size 2.

equivalence classes			equations
type	repr.	others	`id x == x`
`Int`	`x`	`id x`	`abs 0 == 0`
`Int`	`0`	`abs 0`	`abs 1 == 1`
`Int`	`1`	`abs 1`	
`Int`	`abs x`	—	
`Int -> Int`	`id`	—	
`Int -> Int`	`abs`	—	
`Int -> Int`	`(x+)`	—	
`Int -> Int`	`(0+)`	—	
`Int -> Int`	`(1+)`	—	
`Int -> Int -> Int`	`(+)`	—	

Considering an expression Speculate considers an expression E by trying to find an equivalence-class representative R that is equivalent to E:

- If expression E is found equivalent to R *using equational reasoning*, then E is *discarded*. The equations already tell us that $E = R$.
- If expression E is found equivalent to R *using testing*, then the *new equation* $E = R$ is inserted into the theory and E is inserted into R's equivalence class.

Initialization The algorithm starts by considering *single-symbol expressions* in the signature and *one free variable for each type*. After this initialization, Speculate knows all equivalence classes between expressions of size 1.

Example 1.1 (revisited). Table 1 shows the equivalence classes after initialization for the example from §1 with `0`, `1`, `id`, `abs` and `(+)` in the signature. As yet there are no equations. □

Generating and considering expressions Speculate generates expressions in size order until the size limit is reached. Expressions are constructed from type-correct applications of equivalence-class representatives.

Example 1.1 (revisited). Using the size 1 representatives in Table 1, Speculate generates all candidate expressions of size 2: `id x`, `id 0`, `id 1`, `abs x`, `abs 0`, `abs 1`, `(x+)`, `(0+)`, `(1+)`. Then, it considers all those expressions to arrive at the equations and equivalence classes shown in Table 2.

Table 3. Equivalence classes and equations after considering all expressions of size 3.

equivalence classes			equations
type	repr.	others	
Int	x	id x, x + 0	id x == x
Int	0	abs 0	abs 0 == 0
Int	1	abs 1	abs 1 == 1
Int	abs x	abs (abs x)	x + 0 == x
Int	x + 1	1 + x	0 + x == x
Int	x + x	—	x + 1 == 1 + x
Int->Int	id	—	abs (abs x) == abs x
…	…	…	…

The process of considering expressions is repeated with expressions of further sizes. Table 3 shows equivalence classes after considering all expressions of size 3. □

Multiple variables The algorithm described so far is only able to discover laws involving one distinct variable of each type. Following QuickSpec, dealing with multiple variables is based on the following observation and its contrapositive:

Multi ⇒ *Single* For a several-variables-per-type equation to be true, its one-variable-per-type instance should be true as well, for example:

$$\forall x\, y\, z.(x + y) + z = x + (y + z) \Rightarrow \forall x.(x + x) + x = x + (x + x)$$

¬ *Single* ⇒ ¬ *Multi* If a one-variable-per-type equation is false, all its several-variable-per-type generalizations are false as well, for example:

$$\exists x.(x + x) + x \neq x + (x + x) \Rightarrow \exists x\, y\, z.(x + y) + z \neq x + (y + z)$$

So, we only test a multi-variable equation when its single variable instance is true.

Example 1.1 (revisited). When exploring expressions of size 5, Speculate finds that

```
(x + x) + x == x + (x + x)
```

then proceeds to test all its generalizations to find that

```
(x + y) + z == x + (y + z)
```

□

Finding commutativity After processing expressions of size 3 we might expect to have found commutativity of addition (+). However, it is not found by the algorithm just described. To find commutativity and other similar laws, we must also consider generalizations of a representative expression equated with itself. For example, `x + y == y + x` is a generalization of `x + x == x + x`.

Expressions with several variables per type Speculate has to find classes of expressions with several variables per type before searching for inequalities (§4.3) and conditional equations (§4.4). For each representative expression with at most one variable per type, Speculate considers its possible generalizations up to *n* variables, merging expressions into the same equivalence class if either of the following is true:

1. they normalize to the same expression using the theory;
2. they test equal.

Table 4. How the number of expressions and classes increases with the size limit (for example 1.1).

	max. 2 variables		max. 3 variables	
size limit	#-exprs.	#-classes	#-exprs.	#-classes
1	4	4	5	5
2	12	6	15	8
3	44	12	60	18
4	172	23	250	39
5	748	36	1180	68
6	3436	72	5840	153
7	16492	114	30285	287

Summary So far, we have unconditional equations and equivalence classes of expressions.

4.3 Inequalities between Class Representatives

A naïve approach To find inequalities (< and ≤), a naïve approach enumerates all possible expressions and computes all ≤ relations. But it blows up as the size limit increases.

Example 1.1 (revisited). With a limit of 7 symbols, we would have to check over a quarter of a billion pairs of expressions (16492× 16492, see Table 4). Using the default number of tests, 500, we would perform over one hundred billion evaluations. Even if we waited for that computation to complete, we would still have the problem of filtering redundant laws.

A slightly less naïve approach If we instead insert `True` and `<=` in the background signature, then generate equations, inequalities will appear in the output as:

```
(LHS <= RHS) == True
```

In this way, no explicit support for inequalities is needed. For QuickSpec to discover the law `(x + y <= abs x + abs y) == True` it is enough to set it to explore expressions up to size 9. In about 28s, QuickSpec will print this law along with 125 other laws (see Table 9). The algorithm described in the rest of this section is faster, discovering an equivalent law in about 1s among only 43 other laws. See §6 for further comparison with QuickSpec.

A better approach The actual method used in Speculate is based on two observations:

1. the number of non-functional equivalence classes is far smaller than the number of expressions (Table 4);
2. we already have all equivalence classes and their smallest representatives as a by-product of finding unconditional equations.

So, Speculate finds inequalities in three steps:

1. list all pairs of class representatives;
2. test to select pairs that are related by ≤;
3. discard redundant inequalities.

Example 1.1 (revisited). Here are the inequalities found by listing and selecting pairs related by ≤ before discarding redundant inequalities:

```
1.  0 <= 1          4.  0 <= abs x      7.  y <= y + 1
2.  x <= abs x      5.  0 <= abs y      8.  0 <= 1 + 1
3.  y <= abs y      6.  x <= x + 1      9.  1 <= 1 + 1
```

Examples of redundancy include: inequalities 2 and 3 are equivalent; inequalities 4 and 5 are equivalent; inequality 8 is implied by inequalities 1 and 9.

Discarding redundant inequalities. To discard redundant inequalities, Speculate uses the complexity order defined in §2. This is done in three steps, described in the following three paragraphs.

1. Instances Speculate discards more complex inequalities that are instances of simpler inequalities.

Example 1.1 (revisited). The following 4 inequalities are discarded

```
3.  y <= abs y   (implied by 2. x <= abs x)
5.  0 <= abs y   (implied by 4. 0 <= abs x)
7.  y <= y + 1   (implied by 6. x <= x + 1)
9.  1 <= 1 + 1   (implied by 6. x <= x + 1)
```

to arrive at

```
1.  0 <= 1          6.  x <= x + 1
2.  x <= abs x      8.  0 <= 1 + 1
4.  0 <= abs x
```

2. Consequences of transitivity Speculate discards consequences of transitivity $e_1 \leq e_2 \wedge e_2 \leq e_3 \Rightarrow e_1 \leq e_3$ when both antecedents ($e_1 \leq e_2$ and $e_2 \leq e_3$) are *either* simpler than the consequence ($e_1 \leq e_3$), *or* instances of inequalities simpler than the consequence.

Example 1.1 (revisited). The inequality `0 <= 1 + 1` is discarded as it is a consequence of `0 <= 1` and `x <= x + 1`.

3. Instances modulo equivalence closure For all pairs of inequalities I_1 and I_2 where I_1 is simpler than I_2, if any of the expressions in the bounded equivalence closure (§4.1) of I_2 is an instance of any of the expressions in the bounded equivalence closure of I_1, Speculate discards I_2.

4.4 Conditional Equations between Class Representatives

In this section, we detail how conditional equations are generated based on the equational theory (§4.2), class representatives (§4.2) and inequalities (§4.3) between boolean values.

A digraph of candidate conditions There is a connection between conditional laws and inequalities. Using the standard definition of Boolean <= we could define:

```
(==>) = (<=)
```

We already have information about <= from the previous step (§4.3). We can build a digraph of boolean expressions ordered by implication as shown in Figure 2. We include `False` and `True`.

Discovering conditional laws For each pair of representatives e_1 and e_2 from different equivalence classes, we search for the *weakest* conditions under which each of them holds. Instead of searching through all possible conditions from class representatives we use the digraph of conditions to prune the search space. We make a fresh copy of the digraph and repeat the following until there are no unmarked nodes:

1. pick an arbitrary unmarked node with condition c;
2. check $c \Rightarrow e_1 = e_2$ by evaluating it for a set number of test cases;

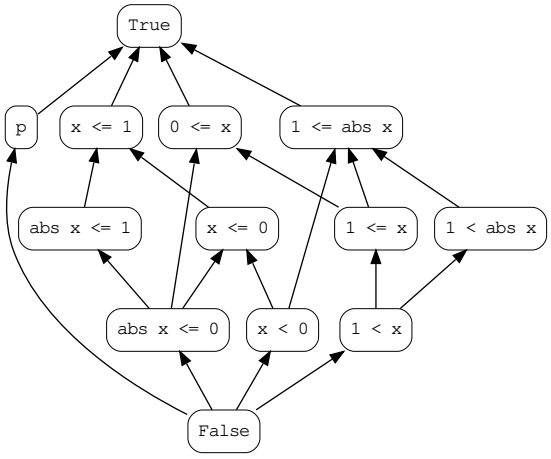

Figure 2. Conditions ordered by logical implication for Example 1.1 from §1 when considering expressions of at most one distinct variable of each type.

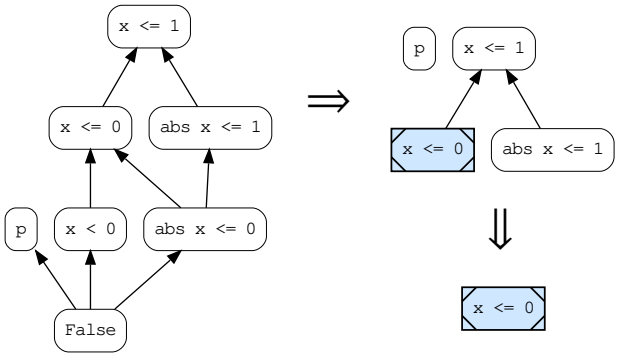

Figure 3. Possible transformations performed on the ordering structure from Figure 2 when searching for the weakest condition for `x + abs x == 0` to hold.

3. if all tests pass then mark c as visited and remove all nodes from which c can be reached as these are for stronger conditions than c.
4. if any test fails remove c and all nodes reachable from it as these are for weaker conditions than c.

The remaining nodes are the weakest conditions for which $e_1 = e_2$. The algorithm is sound modulo testing.

Example 1.1 (revisited). Suppose we are trying to find the weakest condition for which `x + abs x == 0` holds. We may start by considering `1 < x ==> x + abs x == 0` for which tests fail: the node for `1 < x` and all five nodes reachable from it are removed from the graph, yielding the first graph in Figure 3. We may then consider `x <= 0`, for which all tests succeed: we mark it as visited and remove three other nodes from which it can be reached, yielding the second graph in Figure 3. Fast-forwarding to the end, we are left with a single node: the condition `x <= 0` is the weakest condition for `x + abs x == 0` to hold.

Filtering redundant conditional laws In brief, we discard a conditional equation $c_1 \Rightarrow e_1 = e_2$ if we also have a conditional equation $c_0 \Rightarrow e_1 = e_2$ and *either* $c_1 = c_0$ according to the theory (§4.2), *or* $c_1 \Rightarrow c_0$ according to the implication digraph.

5 Example Applications and Results

In this section, we use Speculate:

- to find laws about simple functions on lists (§5.1);
- to find a complete implementation of insertion sort (§5.2);
- to find ordering properties of binary-tree functions (§5.3);
- to find ordering properties of digraph functions (§5.4);
- to find an almost complete axiomatisation for regular-expression equivalence (§5.5).

Then, in §5.6 we give a summary of performance results for all these applications.

We emphasize what is new compared with QuickSpec [8, 22]. So we often omit details of reported unconditional equations where QuickSpec produces similar results. In §6 we shall summarise differences with QuickSpec, including some reasons why the tools may give slightly different sets of unconditional equations.

Sometimes, for the sake of space, we discuss only a selection of inequalities and conditional equations, but always note where others are also generated.

5.1 Finding properties of basic functions on lists

Given the value `[]`, the operators `(:)` and `(++)`, and the functions `head` and `tail`, all with `Int` as element type, Speculate first reports the following equations:

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

Exactly the same laws are found by QuickSpec [8, 22].

Lexicographic ordering But Speculate goes on to print the following inequalities, assuming the default lexicographical ordering Haskell derives for lists.

```
      [] <= xs
      xs <= xs ++ ys
      xs <= head xs:tail xs
xs ++ ys <= xs ++ (ys ++ zs)
```

The law `xs <= head xs:tail xs` may seem strange, but it is correct, even when `xs = []`. As `(:)` is non-strict:

```
      [] <= head []:tail []
      [] <= ⊥:⊥
```

Subsequence ordering Speculate allows the user to request inequalities based on orderings other than an `Ord` instance. For example, if we provide as an `Args` field (§3)

```
instances =
  [ ordWith (isSubsequenceOf :: [Int]->[Int]->Bool) ]
```

then Speculate uses `isSubsequenceOf` (from `Data.List`) as `<=` for lists of `Int`s, and reports the following inequalities:

```
       [] <= xs
       xs <= x:xs
       xs <= xs ++ ys
       xs <= ys ++ xs
       xs <= tail (xs ++ xs)
      [x] <= x:xs
       xs <= head xs:tail xs
     x:xs <= x:(y:xs)
 xs ++ ys <= xs ++ (ys ++ zs)
 xs ++ ys <= xs ++ (zs ++ ys)
     x:xs <= x:(xs ++ ys)
     x:xs <= x:(ys ++ xs)
 xs ++ ys <= xs ++ (x:ys)
    [x,y] <= x:(y:xs)
xs ++ [x] <= xs ++ (x:ys)
```

Automatically checking given orderings Before starting to compute conjectures, Speculate checks by testing that the requested inequality ordering is reflexive and antisymmetric with respect to `(==)`, and transitive. If not, it refuses to go further. For example, if we set `(/=)` as an ordering function for the type `[Int]`, Speculate reports:

```
Error: (<=) :: [Int] -> [Int] -> Bool
       is not an ordering (not reflexive,
       not antisymmetric, not transitive)
```

5.2 Sorting and Inserting: deducing their implementation

With `[]` and `(:)` in the background signature, and functions `insert` and `sort` from `Data.List` in the foreground, Speculate first reports 7 equations. QuickSpec produces a different but similar set of 7 equations. Both QuickSpec and Speculate find the base case of `insert` and the recursive case of insertion `sort`:

```
insert x [] == [x]
sort (x:xs) == insert x (sort xs)
```

By default, Speculate hides laws with no variables. If we switch on the option to reveal them, Speculate also reports the base case for `sort`:

```
sort [] == []
```

If we also include `<=` and `<` for the element type in the background, Speculate reports the two conditional recursive cases

```
x <= y ==> insert x (y:xs) == x:(y:xs)
 x < y ==> insert y (x:xs) == x:insert y xs
```

completing a full implementation of insertion sort synthesised from results of black-box testing.

5.3 Binary search trees

In this section, we apply Speculate to functions on binary search trees, with the following datatype.

```
data BT a = Null | Fork (BT a) a (BT a)
```

We declare two search trees equivalent if they contain the same elements. Also, tree a is less than or equal to tree b if all elements of tree a are present in tree b.

```
instance (Eq a, Ord a) => Eq (BT a) where
  (==)  =  (==) `on` toList

instance (Eq a, Ord a) => Ord (BT a) where
  (<=)  =  isSubsequenceOf `on` toList
```

Equations If we apply Speculate to

```
insert :: Ord a => a -> BT a -> BT a
delete :: Ord a => a -> BT a -> BT a
isIn   :: Ord a => a -> BT a -> Bool
```

it first reports 14 equations, including:

```
insert x (insert x t) == insert x t
delete x (delete x t) == delete x t
  isIn x (insert x t) == True
  isIn x (delete x t) == False
```

We find that insertion and deletion of an element `x` are idempotent, and that they appropriately determine the outcomes of subsequent membership tests.

Inequalities Speculate then reports 11 inequalities. The first three are:

```
          Null <= t
             t <= insert x t
delete x t <= t
```

That is: the least tree is an empty tree; inserting elements makes trees larger; deleting elements makes trees smaller.

Another group of five inequalities are about combinations of some pair of the functions `insert`, `delete` and `isIn`:

```
              delete x t <= delete x (insert y t)
   insert x (delete y t) <= insert x t
   delete x (insert y t) <= insert y (delete x t)
               isIn x t ==> isIn x (insert y t)
     isIn x (delete y t) ==> isIn x t
```

Conditional equation Speculate also reports this conditional equation:

```
x /= y ==>
  insert y (delete x t) == delete x (insert y t)
```

Applied to distinct elements, `insert` and `delete` commute.

5.4 Digraphs

In this section, we apply Speculate to a directed-graph library based on the following adjacency-list datatype

```
data Digraph a = D [(a,[a])]
```

where values of the parametric type `a` are identified with nodes of the digraph.

With `elem` and `[]` in the background, we apply Speculate to the following functions:

```
empty    :: Digraph a
addNode  :: Ord a =>      a -> Digraph a -> Digraph a
addEdge  :: Ord a => a -> a -> Digraph a -> Digraph a
preds    :: Ord a => a -> Digraph a -> [a]
succs    :: Ord a => a -> Digraph a -> [a]
isNode   :: Ord a => a -> Digraph a -> Bool
isEdge   :: Ord a => a -> a -> Digraph a -> Bool
isPath   :: Ord a => a -> a -> Digraph a -> Bool
subgraph :: Ord a =>  [a] -> Digraph a -> Digraph a
```

The `subgraph ns` function extracts the subgraph of its argument with nodes restricted to those listed in `ns`.

We define an ordering on digraphs as follows.

```
instance Ord a => Ord (Digraph a) where
  g1 <= g2 = all (`elem` nodes g2) (nodes g1)
          && all (`elem` edges g2) (edges g1)
```

The ordering relationship holds if all nodes and edges of `g1` are also present in `g2`.

Equations Speculate reports 15 equations. For example, they include these commutativity rules about `addNode` and `subgraph`:

```
addNode x (addNode y a) == addNode y (addNode x a)
subgraph xs (subgraph ys a) ==
                          subgraph ys (subgraph xs a)
```

Conditional Equations Of the two reported conditional equations, the most interesting is:

```
elem x xs ==> subgraph xs (addNode x a)
           == addNode x (subgraph xs a)
```

Indeed, `addNode x` and `subgraph xs` commute when `x` is an element of `xs`.

Inequalities Speculate reports a dozen inequalities. These five are general laws about the relative extent of graphs.

```
           empty <= a
               a <= addNode x a
   subgraph xs a <= a
               a <= addEdge x y a
     addNode x a <= addEdge x y a
```

Other inequalities involve `empty` or give simple rules about `isNode`, `isEdge` and `isPath`. They are all correct, but we omit them to save space.

5.5 Regular Expressions

In this section, we use Speculate to conjecture properties about regular expressions. As we shall see, this is a much more demanding example. We shall reach the limits of what we can do with Speculate.

We declare the following datatype `RE a` with a parametric type `a` for the alphabet.

```
data RE a = Empty
          | None
          | Lit a
          | Star (RE a)
          | RE a :+ RE a
          | RE a :. RE a
```

We declare the `Listable` instance

```
instance Listable a => Listable (RE a) where
  tiers =  cons0 Empty
        \/ cons0 None `ofWeight` 1
        \/ cons1 Lit    \/ cons1 Star
        \/ cons2 (:+)   \/ cons2 (:.)
```

Table 5. Regular Expression Axioms, the size of the largest side (LHS/RHS) and whether each is found by Speculate.

Basic / Common Axioms		**expr. size**	**found**
1. Identity (+)	$E + \emptyset \equiv E$	3	yes
2. Idempotence (+)	$E + E \equiv E$	3	yes
3. Commutativity (+)	$E + F \equiv F + E$	3	yes
4. Associativity (+)	$E + (F + G) \equiv (E + F) + G$	5	yes
5. Null (.)	$E\emptyset \equiv \emptyset E \equiv \emptyset$	3	yes
6. Identity (.)	$E\epsilon \equiv \epsilon E \equiv E$	3	yes
7. Left distributivity	$E(F + G) \equiv EF + EG$	7	yes (after almost 3 days)
8. Right distributivity	$(E + F)G \equiv EG + FG$	7	yes (after almost 3 days)
9. Associativity (.)	$E(FG) \equiv (EF)G$	5	yes
Salomaa (1966) Axioms [18]			
S10. Left expansion ($*$)	$E^* \equiv \epsilon + E^*E$	6	entailed by $E^*E \equiv EE^*$ and K10
S11. Inner expansion ($*$)	$E^* \equiv (\epsilon + E)^*$	4	yes
S12. Inference (ewp)	$E \equiv EF + G \Rightarrow E \equiv GF^*$ **if** $ewp(F)$	10	no
Conway (1971) Axioms [9]			
C10. Elimination ($+^*$)	$(E + F)^* \equiv (E^*F)^*E^*$	8	no
C11. Elimination ($.^*$)	$(EF)^* \equiv \epsilon + E(FE)^*F$	10	no
C12. Idempotence ($*^*$)	$(E^*)^* \equiv E^*$	3	yes
C13. Expansion ($*$)	$E^* \equiv (E^n)^*E^{<n}$ $(n > 0)$	—	no
Kozen (1994) Axioms [14]			
K10. Left expansion ($*$)	$\epsilon + EE^* \equiv E^*$	6	yes
K11. Right expansion ($*$)	$\epsilon + E^*E \equiv E^*$	6	entailed by $E^*E \equiv EE^*$ and K10
K12. Left inequality	$F + EG \leq G \Rightarrow E^*F \leq G$	7	degenerate case: $F + GG \leq G \Rightarrow G(F + G) \leq G$ (3 days)
K13. Right inequality	$F + GE \leq G \Rightarrow FE^* \leq G$	7	degenerate case: $F + GG \leq G \Rightarrow FG^* \leq G$ (3 days)

We declare a three-symbol alphabet, also with a `Listable` instance:

```
newtype Symbol = Symbol Char deriving (Eq, Ord, Show)

instance Listable Symbol where
  tiers = cons0 (Symbol 'a')
        \/ cons0 (Symbol 'b') `ofWeight` 1
        \/ cons0 (Symbol 'c') `ofWeight` 2
```

The `ofWeight` applications make these constructions appear less frequently in the test value enumeration.

Testing equivalence by matching We wish to define equivalence of REs by equality of string-matching outcomes. To do so, we define a function to translate the `RE` representation into the string format used by an existing library [1] for matching.

```
translate :: (a -> Char) -> RE a -> String
```

The library exports `(=~)` where `s =~ e` if `s` matches `e`. Using `translate` and `=~`, we define:

```
match :: (a -> Char) -> [a] -> RE a -> Bool
match f xs r = map f xs =~ translate f r
```

So, for example:

```
> match id "aa" (Star (Lit 'a') :. Lit 'b')
False
> match id "aa" (Star (Lit 'a') :. Star (Lit 'b'))
True
```

[1] `Text.Regex.TDFA` from the `regex-tdfa` package.

With `match` defined, we can now implement approximate equivalence and ordering of regular expressions based on a limited number of membership tests:

```
testMatches :: (Listable a, Show a, Charable a, Ord a)
            => RE a -> [Bool]
testMatches = map (\e -> match toChar e r)
            $ take 120 list


(/==/), (/<=/) :: RE Symbol -> RE Symbol -> Bool
r /==/ s = testMatches r == testMatches s
r /<=/ s =
  and $ zipWith (<=) (testMatches r) (testMatches s)
```

Failing first attempts In our first attempts using this approach, execution times were excessive. Even after caching up to ten million `textMatches` results, a 30-minute run produced some wrong equations due to insufficient testing! Our solution was *down-sizing*.

Starting small We reconfigure Speculate to produce equations only up to size 3. After a couple of minutes, it prints:

```
1.          r :+ r == r
2.  Star (Star r) == Star r
3.       r :+ None == r
4.      r :. Empty == r
5.       r :. None == None
6.      Empty :. r == r
7.       None :. r == None
8.          r :+ s == s :+ r
```

All these are sensible and correct laws about regular expressions. So now we declare canonicalRE as follows:

```
canonicalRE :: (Eq a, Ord a) => RE a -> Bool
canonicalRE (r     :+ s) | r >= s = False  -- by 1&8
canonicalRE (Star (Star r))       = False  -- by 2
canonicalRE (r     :+ None)       = False  -- by 3
canonicalRE (None  :+ r)          = False  -- by 3&8
canonicalRE (r     :. Empty)      = False  -- by 4
canonicalRE (r     :. None)       = False  -- by 5
canonicalRE (Empty :. r)          = False  -- by 6
canonicalRE (None  :. r)          = False  -- by 7
canonicalRE _                     = True
```

and use it to refine our `Listable` instance by adding `'suchThat' canonicalRE`.

Equations of size 4 With the updated `Listable` instance, Speculate considers a greater range of candidate equations with the same number of tests. Configured to produce equations up to size 4, it prints the following new laws:

```
     r :+ Star r == Star r
     Star r :. r == r :. Star r
Star (r :+ Empty) == Star r
  Empty :+ Star r == Star r
```

Now we repeat the process, further refining canonicalRE, and so the `Listable` instance, on the basis of these conjectured laws.

Equations of size 5 and 6 We reduce the number of tests to 200 and again repeat the process for sizes 5 and 6. Speculate prints seven equations of size 5 and nine of size 6 — including axioms 5, 9 and K10 from Table 5.

Inequalities and equations of size 7 Configured to explore equations and inequalities of size 7, Speculate finds the distributive laws 7 and 8 from Table 5. At last, Speculate finds all the common laws from all three axiomatisations of regular expressions. It also finds the following degenerate cases of Kozen's conditional inequalities — crucial ingredients in his complete axiomatisation:

```
r :+ (s :. s) <= s ==> r :. Star s <= s
r :+ (s :. s) <= s ==> s :. (r :+ s) <= s
```

Summary This case study was "a stretch". We wanted to see how far we could get with Speculate. With patience, we can get very close to a complete axiom system, but with the current version of Speculate it is just out of reach.

5.6 Performance Summary

Performance results are summarized in Table 6. Leaving aside the regular-expression application, Speculates takes up to a few seconds to consider expressions for up to size 5. Our tool and examples were compiled using `ghc -O2` (version 8.0.1) under Linux. The platform was a PC with a 2.2Ghz 4-core processor and 8GB of RAM.

6 Related Work

QuickSpec The QuickSpec tool [8, 20–22] discovers equational specifications automatically. Our technique is an extension that allows production of conditional equations and inequalities. QuickSpec inspired us to start working on Speculate. Table 7 shows a summary of differences between QuickSpec 1, QuickSpec 2 and Speculate.

In principle QuickSpec can generate conditional equations, but only with conditions restricted to applications of a set of declared predicates. Consider the following example from [22]. When asked to generate laws about `zip` and `(++)`, both QuickSpec and Speculate produce the following equations:

```
zip xs (xs ++ ys) == zip xs xs
zip (xs ++ ys) xs == zip xs xs
```

These laws are valid but they have conditional generalizations:

```
length xs == length ys ==>
               zip xs (ys ++ zs) == zip xs ys
length xs == length ys ==>
               zip (xs ++ zs) ys == zip xs ys
```

In Speculate, it is enough to have `(==)` and `length` among the background constants to obtain the more general laws.

QuickSpec can only discover these more general laws given quite explicit directions. By providing `length` in the background and setting the following in QuickSpec's `predicates` field

```
predicates =
  [ predicate (undefined :: Proxy "eqLen") eqLen ]
```

where

```
eqLen :: [Int] -> [Int] -> Bool
eqLen xs ys  =  length xs == length ys
```

QuickSpec is able to find the more general laws in the form:

```
eqLen xs ys ==> zip xs (ys ++ zs) == zip xs ys
eqLen xs ys ==> zip (xs ++ zs) ys == zip xs ys
```

With regards to how laws are reported, we made a different design choice to QuickSpec. QuickSpec reports laws as soon as they are discovered, so the user sees progress as QuickSpec runs. Speculate only reports laws after running the completion procedure, so later laws can be used to discard earlier ones. Speculate also, by default, does not report variable-free laws like `sort [] == []`.

QuickSpec has support for polymorphism: if an equation is discovered for a polymorphic version of a function it can be used as a pruning rule for all its monomorphic instances. Speculate does not yet support that polymorphism; it requires monomorphic instances.

To double-check Speculate's reimplementation of the basic equation generating machinery in QuickSpec: (1) we compared Speculate output with QuickSpec output to check if there was any missing equation, and (2) we compared performance of the two tools. This comparison is summarized in Table 8. QuickSpec 2 is a little bit faster than Speculate — early profiling indicates that we were not as smart as the QuickSpec authors when implementing our term rewriting and completion engine.

Table 9 presents needed size limits and times to generate some inequalities and conditional laws for QuickSpec 2 and Speculate. Results in tables 8 and 9 are based on QuickSpec 1 version 0.9.6 and on QuickSpec 2 development version from 11 May 2017 with git commit hash `3c6e010`. At the time of writing, developers are working on improving support for conditional laws in QuickSpec.

Table 6. Summary of Performance Results: figures are mean values across all runs; size limit = maximum number of expression size; #-tests = maximum number of test-cases for any property; time = rounded elapsed time and space = peak memory residency (both from GNU `time`).

		configured size limit for			maximum		resources		number of reported		
Example		eqs.	ineqs.	cond. eqs.	#-vars	#-tests	time	space	eqs.	ineqs.	cond. eqs.
`(+)` and `abs`	(§1)	5	4	4	2	500	3s	7MB	23	17	4
		5	5	5	2	500	25s	7MB	23	44	4
		6	5	5	3	500	2m 37s	8MB	43	44	24
List	(§5.1)	5	4	–	3	500	< 1s	7MB	6	6	–
		7	6	–	3	500	31s	9MB	7	30	–
Insert Sort	(§5.2)	5	–	3	2	500	< 1s	7MB	11	–	2
		6	–	5	3	500	5s	8MB	16	–	8
		7	–	6	3	500	1m 27s	12MB	12	–	12
Binary Trees	(§5.3)	5	4	4	2	500	< 1s	7MB	16	4	1
		6	5	5	3	500	14s	7MB	16	22	5
Digraphs	(§5.4)	5	4	4	2	500	1s	8MB	15	12	2
		6	5	5	3	500	1m 52s	10MB	27	30	34
		6	5	5	3	6000	2m 22s	23MB	25	30	17
Regexes	(§5.5)	3	–	–	–	500	1m 30s	< 6GB	8	–	–
		4	–	–	–	400	9m 11s	< 6GB	12	–	–
		5	–	–	–	200	17m 13s	< 6GB	19	–	–
		6	–	–	–	200	1h 26m 32s	6GB	28	–	–
		7	7	–	2	200	2d 22h 30m 10s	6GB	130	699	–

Table 7. Speculate contrasted with QuickSpec 1 and QuickSpec 2.

	QuickSpec 1	QuickSpec 2	Speculate
Testing Strategy	random (QuickCheck)	random (QuickCheck)	enumerative (LeanCheck)
Direct discovery of equations	yes	yes	yes
of inequalities	no	no	yes
of conditional equations	no	restricted	yes
Reported equations	as discovered	as discovered	after completion
Constant laws (laws with no variables)	yes	yes	hidden by default
How search is bounded	depth-bounded	size-bounded	size-bounded
Explicit treatment of polymorphic functions	no	yes	no
Support for pruning by external theorem provers	no	yes	no
Performance (see Table 8)	slowest	fastest	median

Table 8. Timings and equation counts when generating unconditional equations using Speculate, QuickSpec 1 and QuickSpec 2. In QS1, expressions are primarily explored up to a certain depth [8], so, for a fair comparison, we have introduced a depth limit in QS2 and Speculate.

		size	depth	max.	Runtime in seconds			#-reported equations		
Example		limit	limit	#-tests	QS1	QS2	Speculate	QS1	QS2	Speculate
`(+)` and `abs`	(§1)	6	4	500	4s	< 1s	< 1s	10	13	9
		7	4	500	7s	< 1s	2s	14	15	14
0, 1, +, ×	(`Int`)	7	4	500	95s	3s	6s	9	13	9
List	(§5.1)	7	4	500	52s	< 1s	< 1s	28	7	7
		8	4	500	10m 31s	< 1s	< 1s	40	7	7

Table 9. Needed size limits and times to generate some inequalities and conditional laws for QuickSpec 2 and Speculate. Speculate is able to find some laws much faster as they appear when exploring a smaller size.

			Needed size limit		Needed max #-tests		Runtime		# reported laws	
Example		Target Law	QS2	Spl.	QS2	Spl.	QS2	Spl.	QS2	Spl.
`(+)` and `abs`	(§1)	`x <= abs x`	4	2	500	500	< 1s	< 1s	12	3
		`x <= abs (x + x)`	6	4	500	500	< 1s	< 1s	36	23
		`x + y <= x + abs y`	8	4	500	500	8s	< 1s	82	23
		`x + y <= abs x + abs y` (or `x + y <= abs x + y`)	9	5	500	500	34s	1s	125	43
Binary Trees	(§5.3)	`isIn x t ==> isIn x (insert y t)`	9	5	2000	500	37s	1s	34	39
Regexes	(§5.5)	$F + GE \leq G \Rightarrow E^*F \leq G$	14	7	(open research problem)					

CoCo The CoCo (Concurrency Commentator) tool [24] generates specifications for concurrent Haskell programs containing laws about refinement or equivalence of side effects. Drawing upon the techniques used in QuickSpec and Speculate, CoCo also works by testing, and can be seen as QuickSpec/Speculate to discover equivalences and refinements between concurrent expressions.

HipSpec QuickSpec and Speculate can only provide *apparent* laws as their results are based on testing. The HipSpec system [7] automatically derives and *proves* properties about functional programs. HipSpec first uses QuickSpec to discover conjectures to prove. Then, using inductive theorem proving, it automatically generates a set of equational theorems about recursive functions. Those theorems can be used as a background theory for proving properties about a program.

Hipster The Hipster system [12] integrates QuickSpec with the proof assistant Isabelle/HOL. Hipster speeds up and facilitates the development of new theories in Isabelle/HOL by using HipSpec to discover basic lemmas automatically.

Daikon The Daikon tool [11] automatically discovers apparent invariants in imperative programs. Those invariants include: preconditions and postconditions of statements, equational relationships between variables at a given program point and equations between functions from a library. Unlike QuickSpec and Speculate, Daikon is aimed at *imperative* programs, written in languages such as: C, C++, Java and Perl. Daikon works by testing potential invariants against observed runtime values.

FitSpec The FitSpec tool [4] provides automated assistance in the task of refining specifications. To do so, it tests mutant variations of functions under test against a given property set, recording any surviving mutants that pass all tests. The user is prompted to strengthen the property set or to remove redundant properties. It has been applied to QuickSpec results and could also be applied to Speculate results.

Property-based testing Since the introduction of QuickCheck [6], several other property-based testing libraries and techniques have been developed, such as SmallCheck, Lazy SmallCheck [16, 17] and Feat [10]. These tools automatically test properties describing Haskell functions meaning that Speculate results can be used as properties for regression tests.

7 Conclusions and Future Work

Conclusions In summary, we have presented a tool that, given a collection of Haskell functions, conjectures a specification involving apparent inequalities and conditional equations. This specification can contribute to understanding, documentation and properties for regression tests. As set out in §3 and §4, Speculate enumerates, tests expressions and reasons from test results to produce its conjectures. We have demonstrated in §5 Speculate's applicability to a range of small examples, and we have briefly compared in §6 some of the results obtained with related results from other tools.

Value of reported laws The conjectured equations and inequalities reported by Speculate are surprisingly accurate in practice, despite their inherent uncertainty in principle. These conjectures provide helpful insights into the behaviour of functions. For the sorting example in §5.2, we were even able to synthesise a complete implementation. When Speculate finds an apparent but incorrect law, increasing the number of tests per law is a simple and effective solution (§5.4). The special treatment of inequalities and conditional equations makes possible the generation of laws previously unreachable by a tool such as QuickSpec [8, 22].

Ease of use Arguably, a tool is easier to use if it requires less work from the programmer. As we illustrated in §3, writing a minimal program to apply Speculate takes only a few lines of code. The speculate function parses command-line arguments to allow easy configuration of test parameters. If only standard Haskell datatypes are involved, no extra `Listable` instances are needed. If user-defined data types can be freely enumerated without a constraining data invariant, instances can be automatically derived.

However, often we do need to restrict enumeration by a data invariant, and a crude application of a filtering predicate may be too costly, with huge numbers of discarded values. Effective use of Speculate may require careful programming of custom `Listable` instances, even if suitable definitions can be very concise. The Speculate library does not currently incorporate methods to derive enumerators of values satisfying given preconditions [5, 15].

Future Work We note a few avenues for further investigation that could lead to improved versions of Speculate or similar tools.

Improve performance when generating inequalities The algorithm to generate equations is partly based on the observation that, for an equation to be true, its one-variable-per-type instance must

be true. So, Speculate initially considers one-variable-per-type equations, generalizing them to their several-variable versions only if they are found true (§4.2). The same applies to inequalities: for `x + y <= x + abs y` to be true `x + x <= x + abs x` must be true. Speculate does not yet exploit this and does some unnecessary testing.

Parallelism As a way to improve performance, particularly when dealing with costly test functions such as in the regular expressions example (§5.5), we could parallelise parts of Speculate. For example, divide the testing of laws among multiple processors.

Automated generation of efficient Listable instances Right now, to use Speculate, Listable instances have to be explicitly declared. Speculate could take the constructors of a type in its constants list (§3) and *automatically* construct a generator for values of that type. This generator could be improved as new equations are discovered. If for a given type constructor `Cons`, we discover that `Cons x y == Cons y x`, in further tests, we would only apply `Cons` to ordered `x` and `y`. This is what we did manually in our regular-expressions example (§5.5).

Improve filtering of redundant inequalities and conditions Although Speculate already filters out a lot of redundant inequalities and conditional equations, there is still room for improvement. Recall these laws from Example 1.1:

```
1. x == 1 ==> 1 == abs x
2. abs x <= y  ==>  abs (x + y) == x + y
3. abs y <= x  ==>  abs (x + y) == x + y
```

By interpreting the condition as a variable assignment, the first law is an instance of `1 == abs 1`. The other two laws are equivalent by the commutativity of addition `(+)`.

Special treatment of conjunctions and disjunctions Although not explored much in the examples in this paper, conjunctions (&&) and disjunctions (||) can often occur as conditions of properties [17]. In the current version of Speculate, logical operators are treated as regular functions. In future versions we could treat them specially, exploiting their properties of commutativity and associativity to reduce the search space.

Checking that given equivalences are congruences In §5.1, we mention that before running any tests, Speculate checks whether given equality (==) functions are equivalences (reflexive, symmetric and transitive). Speculate also assumes, but does not check, that given == functions are congruences: in any expression e, suppose we replace a subexpression s by s', where $s \equiv s'$, to obtain e' as the whole: then we require $e \equiv e'$. Future versions of Speculate should check for congruence.

Detecting and using equivalences and orderings In the current version of Speculate, the user has to say which equivalence (==) and ordering (<=) functions to use. Or, in the case of standard types, the user has to explicit provide functions to override the standard ones. The algorithm to compute equations can work with any function that is a congruent equivalence. Similarly, the algorithm to compute inequalities, can work with any function that is an ordering. Speculate could detect any given functions that have these properties and autonomously search for laws based on them.

Availability

Speculate is freely available with a BSD3-style license from:

- https://hackage.haskell.org/package/speculate
- https://github.com/rudymatela/speculate

This paper describes Speculate as of version 0.2.5.

Acknowledgements

We thank Nick Smallbone for hospitality and many interesting discussions about QuickSpec; Maximilian Algehed for helping with running one of QuickSpec's examples; Ivaylo Hristakiev for discovering a bug in our term unification algorithm; and anonymous reviwers for their comments on earlier drafts.

Rudy Braquehais is supported by CAPES, Ministry of Education of Brazil (Grant BEX 9980-13-0).

References

[1] 2017. Haskell's Data.Dynamic library documentation. https://hackage.haskell.org/package/base/docs/Data-Dynamic.html. (2017).
[2] Franz Baader and Tobias Nipkow. 1999. *Term Rewriting and All That.* Cambridge University Press.
[3] Leo Bachmair, Nachum Dershowitz, and David A. Plaisted. 1989. Completion Without Failure. In *Resolution Of Equations In Algebraic Structures.* Vol. 2. Academic Press, Boston, 1–30.
[4] Rudy Braquehais and Colin Runciman. 2016. FitSpec: refining property sets for functional testing. In *Haskell'16.* ACM, 1–12.
[5] Lukas Bulwahn. 2012. Smart Testing of Functional Programs in Isabelle. In *LPAR 2012 (LNCS 7180).* Springer, 153–167.
[6] Koen Claessen and John Hughes. 2000. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *ICFP'00.* ACM, 268–279.
[7] Koen Claessen, Moa Johansson, Dan Rosén, and Nicholas Smallbone. 2012. HipSpec: Automating inductive proofs of program properties. In *Workshop on Automated Theory eXploration: ATX 2012.*
[8] Koen Claessen, Nicholas Smallbone, and John Hughes. 2010. QuickSpec: Guessing Formal Specifications Using Testing. In *TAP 2010.* Springer, 6–21.
[9] John Horton Conway. 1971. *Regular algebra and finite machines.* Chapman and Hall.
[10] Jonas Duregård, Patrik Jansson, and Meng Wang. 2012. Feat: functional enumeration of algebraic types. In *Haskell'12.* ACM, 61–72.
[11] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen Mccamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. 2006. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming* 69, 1 (2006), 35–45.
[12] Moa Johansson, Dan Rosén, Nicholas Smallbone, and Koen Claessen. 2014. *Hipster: Integrating Theory Exploration in a Proof Assistant.* Springer.
[13] Donald Knuth and Peter Bendix. 1983. Simple Word Problems in Universal Algebras. In *Automation of Reasoning.* Springer, 342–376.
[14] Dexter Kozen. 1994. A completeness theorem for Kleene algebras and the algebra of regular events. *Information and Computation* 110, 2 (1994), 366–390.
[15] Fredrik Lindblad. 2007. Property Directed Generation of First-Order Test Data. In *TFP'07.* 105–123.
[16] Jason S. Reich, Matthew Naylor, and Colin Runciman. 2013. Advances in Lazy SmallCheck. In *IFL'13.* Springer, 53–70.
[17] Colin Runciman, Matthew Naylor, and Fredrik Lindblad. 2008. SmallCheck and Lazy SmallCheck: Automatic Exhaustive Testing for Small Values. In *Haskell'08.* ACM, 37–48.
[18] Arto Salomaa. 1966. Two complete axiom systems for the algebra of regular events. *Journal of the ACM (JACM)* 13, 1 (1966), 158–169.
[19] Tim Sheard and Simon Peyton Jones. 2002. Template Meta-programming for Haskell. In *Haskell'02.* ACM, 1–16.
[20] Nicholas Smallbone. 2011. *Property-based testing for functional programs.* Licentiate Thesis. Chalmers University of Technology.
[21] Nicholas Smallbone. 2013. *Lightweight verification of functional programs.* Ph.D. Dissertation. Chalmers University of Technology.
[22] Nicholas Smallbone and Moa Johansson. 2017. Quick specifications for the busy programmer. (2017). http://www.cse.chalmers.se/~nicsma/papers/quickspec2.pdf Accepted for publication in JFP, Cambridge University Press.
[23] The GHC Team. 1992–2017. The Glasgow Haskell Compiler. https://www.haskell.org/ghc/. (1992–2017).
[24] Michael Walker and Colin Runciman. 2017. Cheap Remarks about Concurrent Programs. (2017). Accepted for presentation at TFP'17.