Tools for Discovery, Refinement and Generalization of Functional Properties by Enumerative Testing

Rudy Matela Braquehais

Doctor of Philosophy

University of York
Computer Science

October 2017

Abstract

This thesis presents techniques for discovery, refinement and generalization of properties about functional programs. These techniques work by reasoning from test results: their results are surprisingly accurate in practice, despite an inherent uncertainty in principle. These techniques are validated by corresponding implementations in Haskell and for Haskell programs: Speculate, FitSpec and Extrapolate. Speculate discovers properties given a collection of black-box function signatures. Properties discovered by Speculate include inequalities and conditional equations. These properties can contribute to program understanding, documentation and regression testing. FitSpec guides refinements of properties based on results of black-box mutation testing. These refinements include completion and minimization of property sets. Extrapolate generalizes counterexamples of test properties. Generalized counterexamples include repeated variables and side-conditions and can inform the programmer what characterizes failures. Several example applications demonstrate the effectiveness of Speculate, FitSpec and Extrapolate.

Contents

1 Introduction **1**
1.1 Motivation 3
1.2 Thesis statement 3
1.3 Contributions 4
1.4 Chapter Preview 4

2 Literature Review: property-based testing and its applications **7**
2.1 Property-based Testing Tools 7
2.1.1 QuickCheck: automated random testing 9
2.1.2 SmallCheck: exhaustive testing for small values 11
2.1.3 Lazy SmallCheck: using laziness to guide enumeration 12
2.1.4 Feat: Functional Enumeration of Algebraic Types 13
2.1.5 Neat: Non-strict Enumeration of Algebraic Types 14
2.1.6 GenCheck: generalized testing 15
2.1.7 Irulan: implicit properties 15
2.1.8 Reach: finding inputs that Reach a target expression 16
2.1.9 SmartCheck: improving counterexamples 16
2.1.10 Hedgehog: integrated shrinking 17
2.1.11 Beyond Haskell 17
2.1.12 Discussion and Comparison 19
2.2 Applications of Property-based testing 21
2.2.1 QuickSpec: discovery of equational laws 21
2.2.2 EasySpec: signature inference for property discovery 22
2.2.3 Bach: discovering relational specifications 22
2.2.4 MuCheck: syntactic mutation testing for Haskell 22
2.2.5 Lightweight Mutation Testing in Haskell 23
2.3 Summary 23

3 LeanCheck: enumerative testing of higher-order properties **25**
3.1 Introduction 25
3.2 Listable Data Types and Fair Enumeration 26
3.3 Testable Properties and Tiers of Tests 32
3.4 Conditional Properties and Data Invariants 33

3.5 Higher Order Properties and Listable Functions 35
3.6 Example Applications and Results 40
3.7 Comparison with Related Work 42
3.8 Conclusion 43

4 FitSpec: refining properties for functional testing 45
4.1 Introduction 45
4.2 Definitions 47
4.3 How FitSpec is Used 49
4.4 How FitSpec Works 51
4.4.1 Enumerating Mutants 52
4.4.2 Testing Mutants against Properties 54
4.4.3 Searching for Survivors 55
4.4.4 Conjecturing Equivalences and Implications 56
4.4.5 Controlling the Extent of Testing 57
4.5 Example Applications and Results 58
4.5.1 Boolean Operators 58
4.5.2 Sorting 59
4.5.3 Binary Heaps 61
4.5.4 Operations over Sets 62
4.5.5 Powersets and Partitions 64
4.5.6 Operations over Digraphs 66
4.5.7 Performance Summary 68
4.6 Comparison with Related Work 68
4.7 Conclusions and Future Work 71

5 Speculate: discovering conditional equations and inequalities by testing 75
5.1 Introduction 75
5.2 Definitions 77
5.3 How Speculate is Used 78
5.4 How Speculate Works 80
5.4.1 Equational Reasoning based on Term Rewriting 81
5.4.2 Equations and Equivalence Classes of Expressions 81
5.4.3 Inequalities between Class Representatives 84
5.4.4 Conditional Equations between Class Representatives 86
5.5 Example Applications and Results 89
5.5.1 Finding properties of basic functions on lists 89
5.5.2 Sorting and Inserting: deducing their implementation 90
5.5.3 Binary search trees 91
5.5.4 Digraphs 92
5.5.5 Regular Expressions 93
5.5.6 Performance Summary 97
5.6 Comparison with Related Work 98

5.7 Conclusions and Future Work 101

6 Extrapolate: generalizing counterexamples of test properties 105
6.1 Introduction 105
6.2 How Extrapolate is Used 107
6.3 How Extrapolate Works 109
6.3.1 Searching for counterexamples 109
6.3.2 Unconditional generalization 110
6.3.3 Conditional generalization 110
6.4 Example Applications and Results 113
6.4.1 A sorting function: exact generalization 114
6.4.2 A calculator language 114
6.4.3 Stress test: integer overflows 115
6.4.4 A serializer and parser 117
6.4.5 XMonad 118
6.4.6 Generalizations as property refinements 119
6.4.7 Performance Summary 122
6.5 Comparison with Related Work 122
6.6 Conclusions and Future Work 127

7 Conclusions & Future Work 131
7.1 Summary of Contributions 131
7.2 Conclusions 132
7.3 Future Work 133

Bibliography 137

List of Figures

2.1 Flow of ideas between property-testing tools for Haskell 8
2.2 Property-based testing tools placed within two axes. 9
2.3 MuCheck applied to a sorting function 23

4.1 Conjecture strengths by percent of surviving mutants. 57

5.1 Full program applying Speculate to `+`, `id` and `abs`. 79
5.2 Diagram summarizing how Speculate works 80
5.3 Conditions ordered by logical implication. 87
5.4 Transformations performed on the ordering structure. 87

6.1 Full program applying Extrapolate to properties of `sort`. 108

List of Tables

2.1 Numbers of integer lists in successive sizes or depths. 14
2.2 Summary of differences between property-based testing tools for Haskell. . 20

3.1 Numbers of data values in successive tiers for several example data types. . 28
3.2 Numbers of values in each tier for two alternative `Listable Int` instances. 30
3.3 Numbers of functions in successive tiers for several types 39
3.4 Ratios of repetitions in different function enumerations when *not* enumerating functions as tuples of results. With a fixed number of tests, as the domain increases, the ratio of repetitions decreases. 40
3.5 Numbers of integer lists of successive sizes or depths. 43

4.1 Numbers of inequivalent mutants in successive tiers. 55
4.2 How enlarging the sorted element-type increases convergence parameters. . 60
4.3 Summary of Performance Results 68
4.4 FitSpec contrasted with MuCheck and Duregård's framework. 70

5.1 Equivalence classes and equations after considering all expressions of size 1. 82
5.2 Equivalence classes and equations after considering all expressions of size 2. 82
5.3 Equivalence classes and equations after considering all expressions of size 3. 83
5.4 How the number of expressions and classes increases with the size limit. . . 85
5.5 Regular expression axioms, their sizes and whether each is found 94
5.6 Summary of performance results 97
5.7 Speculate contrasted with QuickSpec 1 and QuickSpec 2. 98
5.8 Timings and equation counts when generating unconditional equations . . 99
5.9 Needed size limits and times to generate inequalities and conditional laws . 100

6.1 Summary of results for five different applications. 121
6.2 Extrapolate contrasted with Lazy SmallCheck and SmartCheck. 123
6.3 Counterexamples for the `count` property of `sort`. 124
6.4 Counterexamples for the property involving `noDiv0`. 125
6.5 Counterexamples for the property about integer overflows. 125
6.6 Counterexamples for the parser property. 126
6.7 Counterexamples for `prop_delete` from XMonad. 126

Acknowledgements

I am grateful to my PhD supervisor, Colin Runciman, for his advice and guidance in my research. He gave me freedom to pursue my research in what I wanted and the way I wanted. I always went out of my weekly supervision meetings really motivated to do my job. Colin is a brilliant supervisor, his experience makes him able to predict where things might go wrong. Often, when I did not follow his advice, indeed they did! In the words of my colleague Glyn Faulkner: *"Colin has a nasty habit of being right"*.

I am grateful to the several people that provided reviews and technical discussion during the writing of this thesis and the papers on which it is based: Maximilian Algehed, Tim Atkinson, Ben Davis, Trevor Eliott, Glyn Faulkner, Ivaylo Hristakiev, Jeremy Jacob, Lee Pike, Jamey Sharp, and anonymous reviewers. I thank Rob Alexander, my internal examiner, for his comments on my work at an earlier stage, his role in meetings of my Thesis Advisory Panel, and an engaging discussion during my viva. I thank John Hughes, my external examiner, for agreeing to examine my thesis and an engaging discussion during my viva.

I thank past and present members of the PLASMA group for interesting technical discussion in the weekly PLASMA meetings and CSE hallways: Tim Atkinson, Chris Bak, José Calderon, Mike Dodds, Nathan von Doorn, Ivaylo Hristakiev, Jeremy Jacob, Firas Moalla, Detlef Plump, Jason Reich, Ben Simner, Ibrahim Shiyam, Michael Walker, Matt Windsor and anyone else I may have forgotten. Cheers for those in this paragraph who were my office mates in CSE/215 for their company and occasional office banter. Additional thanks for the two who are co-authors of one of the papers in this thesis.

I am grateful to Nick Smallbone and Jonas Duregård for hospitality and many interesting discussions about QuickSpec, Feat and FitSpec during a visit to the Chalmers University of Technology in 2016. I am grateful to several people with whom I had interesting discussions during the conferences I attended through the course of my PhD, and to the Haskell community in general for being receptive.

Many thanks to my partner Camila, for her understanding, rationality, company and ongoing support during my entire PhD. Additional thanks for enduring practice runs of my seminars.

I am grateful to my family for their support during the several steps I had to take before starting my PhD. I wouldn't be here if it weren't for them.

I would like to thank *you*, the reader, for interest in my research and thesis. Enjoy reading it!

This work is supported by CAPES, Ministry of Education of Brazil: Grant BEX 9980-13-0. Special thanks for them for providing the funding to do my PhD.

Declaration

I declare that this thesis is a presentation of original work and I am the sole author except as indicated below. This work has not previously been presented for an award at this, or any other, University. All sources are acknowledged as References.

The following chapters were based on papers published by me and my co-authors:

- Chapter 3: Rudy Braquehais, Michael Walker, José Manuel Calderón Trilla, Colin Runciman. A simple incremental development of a property-based testing tool (functional pearl). *Unpublished draft, 2016-2017.*
  - I carried out most of the implementation and experiments;
  - Colin carried out most of the experiments described in §3.6;
  - All 4 authors contributed roughly the same amount to the paper;
  - The chapter version significantly differs from the tutorial in the paper.
- Chapter 4: Rudy Braquehais and Colin Runciman. FitSpec: refining property sets for functional testing. In *Haskell 2016*, pages 1-12. ACM, 2016.
  - I carried out most of the implementation, experiments and comparative work;
  - Colin carried out most of the experiments described in §4.5.4, §4.5.5 and §4.5.6;
  - I contributed roughly 3/5 of the paper, Colin 2/5.
- Chapter 5: Rudy Braquehais and Colin Runciman. Speculate: discovering conditional equations and inequalities about black-box functions by reasoning from test results. In *Haskell 2017*, pages 40-51. ACM, 2017.
  - I carried out most of the implementation, experiments and comparative work;
  - I contributed roughly 2/3 of the paper, Colin 1/3.
- Chapter 6: Rudy Braquehais and Colin Runciman. Extrapolate: generalizing counterexamples of functional test properties. In *IFL'17: Implementation and Application of Functional Languages (Draft Proceedings)*, pages 13-24. ACM, 2017.
  - I carried out most of the implementation, experiments and comparative work;
  - Colin carried out most of the experiments in §6.4.6;
  - I contributed roughly 2/3 of the paper, Colin 1/3.

Throughout this Thesis, I avoid using the "I" pronoun favour of the "we" pronoun, hereinafter referring to either: (most commonly) me and the reader, e.g. "we follow this argument to [...]"; (rarely) me and my co-authors, e.g. "when implementing [...], we chose to [...]". This page contains the last occurrences of the "I" pronoun in this thesis!

Chapter 1

Introduction

This thesis is concerned with enriching properties of functional programs based on results of enumerative testing. We present techniques to automatically discover, refine and generalize properties. These techniques are validated by corresponding implementations in Haskell for testing of Haskell programs [Jones, 2003, Marlow, 2010, Hutton, 2016]. Example applications demonstrate the effectiveness of these techniques and their implementations.

Testing and its cost The most common approach to maintain software quality and to ensure that a program works is by testing. Software testing is expensive, accounting for 50% of a developer's time [Myers et al., 2011, Claessen and Hughes, 2000]. Śliwerski et al. [2005] estimated that between 30% to 40% of changes in Eclipse and Mozilla projects were bug fixes. Bachmann et al. [2010] further shows a rate of 16% on the Apache Web server project. If we could reduce the cost of testing, we would significantly reduce the overall cost of developing software. One purpose of testing is to expose bugs earlier, or even during implementation, potentially reducing the number of bug fixes and total development cost.

Example program Consider the following (faulty) `sort` function in Haskell:

```
sort :: Ord a => [a] -> [a]
sort []      =  []
sort (x:xs)  =  sort (filter (< x)) xs
             ++ [x]
             ++ sort (filter (> x)) xs
```

Tests In a traditional approach to software testing, the programmer explicitly lists tests by providing expected argument–result pairs for functions. This approach is usually known as *unit testing* [Zhu et al., 1997]. For example, the following are unit tests for `sort`:

```
sortTests :: [Bool]
sortTests  =  [ sort []      == []
              , sort [1,2,3] == [1,2,3]
              , sort [3,2,1] == [1,2,3] ]
```

Properties Tests can be parameterized over values. We call such parameterized tests *properties* [Claessen and Hughes, 2000, Tillmann and Schulte, 2005]. The following are two properties of a `sort` function:

```
prop_sortOrdered :: Ord a => [a] -> Bool
prop_sortOrdered xs  =  ordered (sort xs)

prop_sortCount :: Ord a => a -> [a] -> Bool
prop_sortCount x xs  =  count x (sort xs) == count x xs
```

The first property states that for all lists, sorting yields an ordered list. The second states that the number of occurrences of any item does not change after sorting. Together these two properties form a complete specification of `sort`. As *single samples* of test results, we have:

```
prop_sortOrdered [1,2,3] == True
prop_sortCount 1 [1,2,3] == True
```

Property-based testing Property-based testing[1] tools provide a `check` function that takes a property, tests it by automatically generating test values, then reports the results. The following illustrates typical usage:

```
> check (prop_sortOrdered :: [Int] -> Bool)
+++ OK, passed 200 tests.
> check (prop_sortCount :: Int -> [Int] -> Bool)
*** Failed!  Falsifiable (after 4 tests):
0 [0,0]
```

The `sort` function follows the first property but fails the second due to a fault: it discards repeated elements. To fix `sort`, we need to replace `>` with `>=`. □

The above example uses of `check` exemplify what makes property-based testing compelling. When the system is able to find a counterexample it not only reports that the property failed, it also reports the inputs that caused the property to fail. It is not the user's responsibility to find the crucial test values; they are found automatically. This is the heart of property-based testing. The specified properties should hold for *all* inputs and we leave it to the testing tool to generate appropriate test cases.

Property-based testing is particularly useful in the realm of functional programming, a style of programming that models computation by the application of functions [Hutton, 2016]. The implementation of functions with side-effects is discouraged, or, in the case of Haskell, prohibited. Different calls to a function with the same parameters should yield the same values. This means we need to test properties only once per tuple of argument values.

[1]Property-based testing as described here should not be confused with an *unrelated technique*, with the same name, to test the security of Unix programs by partitioning them into sectors related to a high level property [Fink and Levitt, 1994, Fink et al., 1994, Fink and Bishop, 1997]

Property-based testing as means instead of as an end. Property-based testing can also be applied beyond its original scope of simply testing programs. For example, it can be used as part of a process to automatically generate program specifications [Claessen et al., 2010, Smallbone et al., 2017].

1.1 Motivation

The main motivations for the work reported in this thesis are:

- to reduce the human effort needed for property-based testing;
- to reduce the computational effort needed for property-based testing;
- to increase the benefits obtained by property-based testing.

These goals are achieved by making properties and counterexamples the subject of computational explorations and improvement.

1.2 Thesis statement

This thesis supports the following statement[2]:

> Property-based testing can be used to analyse then *refine* property-sets as functional specifications. It can be used to *discover* properties with side conditions or in the form of inequalities. It can be used to *generalize* counterexamples of properties with side conditions.

This claim is supported by the description of techniques to enrich (enhance or improve) properties. We have implemented tools in the Haskell programming language using those techniques, showing that they can be applied *practically*. This work is described in Chapters 3, 4, 5 and 6. The above claim is detailed in the following paragraphs.

Enumerative Testing We describe a technique for size-bounded enumerative property-based testing. We will show that it is possible to define a size-bounded partial enumeration of functions which can be useful to test higher-order properties and to analyse first-order properties (Chapter 3). This technique is the foundation on which we develop other techniques in this thesis.

Refinement Using property-based testing, we will show that it is possible to evaluate minimality and completeness of property sets. Using results from this analysis, we will show techniques to refine properties (Chapter 4).

[2]Note the variation in order compared with the Thesis title. We shall consider *refinement* before *discovery* to reflect the chronological order in which research was carried out, and also because refining already existing properties is easier than discovering properties out of nothing.

Discovery We will go beyond previously reported methods for discovering properties, showing that property-based testing can be used to discover properties involving inequalities and implications between Haskell expressions (Chapter 5).

Generalization We will show that property-based testing can be used to generalize counterexamples of test properties with repeated variables and side conditions. We will show that these generalized counterexamples can inform programmer more fully and more immediately what characterizes failures and that these generalized counterexamples can also serve as another source of property refinement. (Chapter 6).

1.3 Contributions

The main contributions of this thesis are:

1. A technique to enumerate test values by size, including a size-bounded partial enumeration for functions. This technique serves as the foundation on which we build all other techniques described in this thesis.
2. The *LeanCheck* tool for property-based testing in Haskell using the aforementioned technique. This tool serves as the foundation on which we build other tools described in this thesis.
3. A technique to *refine* properties about black-box functions, readily applicable to purely functional programs. In addition to guiding completion of property-sets, this technique is able to guide *minimization*.
4. The *FitSpec* tool that given a set of Haskell functions and properties about them, automatically guides refinements of those properties.
5. A technique to automatically *discover* properties about black-box functions, readily applicable to purely functional programs. In addition to equational laws, this technique is also able to produce *inequalities* and *conditional equations*.
6. The *Speculate* tool that given a set of Haskell functions, automatically conjectures properties about them.
7. A technique to automatically *generalize* counterexamples of properties allowing *repeated variables* and *side-conditions*.
8. The *Extrapolate* tool that is able to automatically discover and generalize counterexamples to properties.

1.4 Chapter Preview

Chapter 2 presents a summary of recent work in the area of property-based testing. It first lists, describes and compares several existing property-based testing tools. It goes on to

list and describe some applications of property-based testing.

Chapter 3 presents LeanCheck and is concerned with the *testing* of properties. It shows the implementation of LeanCheck covering enumeration of data values and how properties are tested. It discusses conditional properties and data invariants. It shows the implementation of an enumeration of functions and how they can be used to test higher order properties.

Chapters 4–6 have the same structure: they show how each tool is used; describe how each underlying technique works; provide example applications and results; and finally compare and contrast each tool with relevant related work. Chapter 4 presents FitSpec and is concerned with the *refinement* of properties. Chapter 5 presents Speculate and is concerned with the *discovery* of properties. Chapter 6 presents Extrapolate and is concerned with the *generalization* of counterexamples to properties.

Chapter 7 presents the conclusions of this research and outlines possible areas for future work.

Chapter 2

Literature Review: property-based testing and its applications

This chapter presents a summary of recent work in the area of property-based testing. We give particular focus on work done in the Haskell language. Section 2.1 describes several tools for property-based testing. Section 2.2 describes applications of property-based testing beyond the testing of programs.

2.1 Property-based Testing Tools

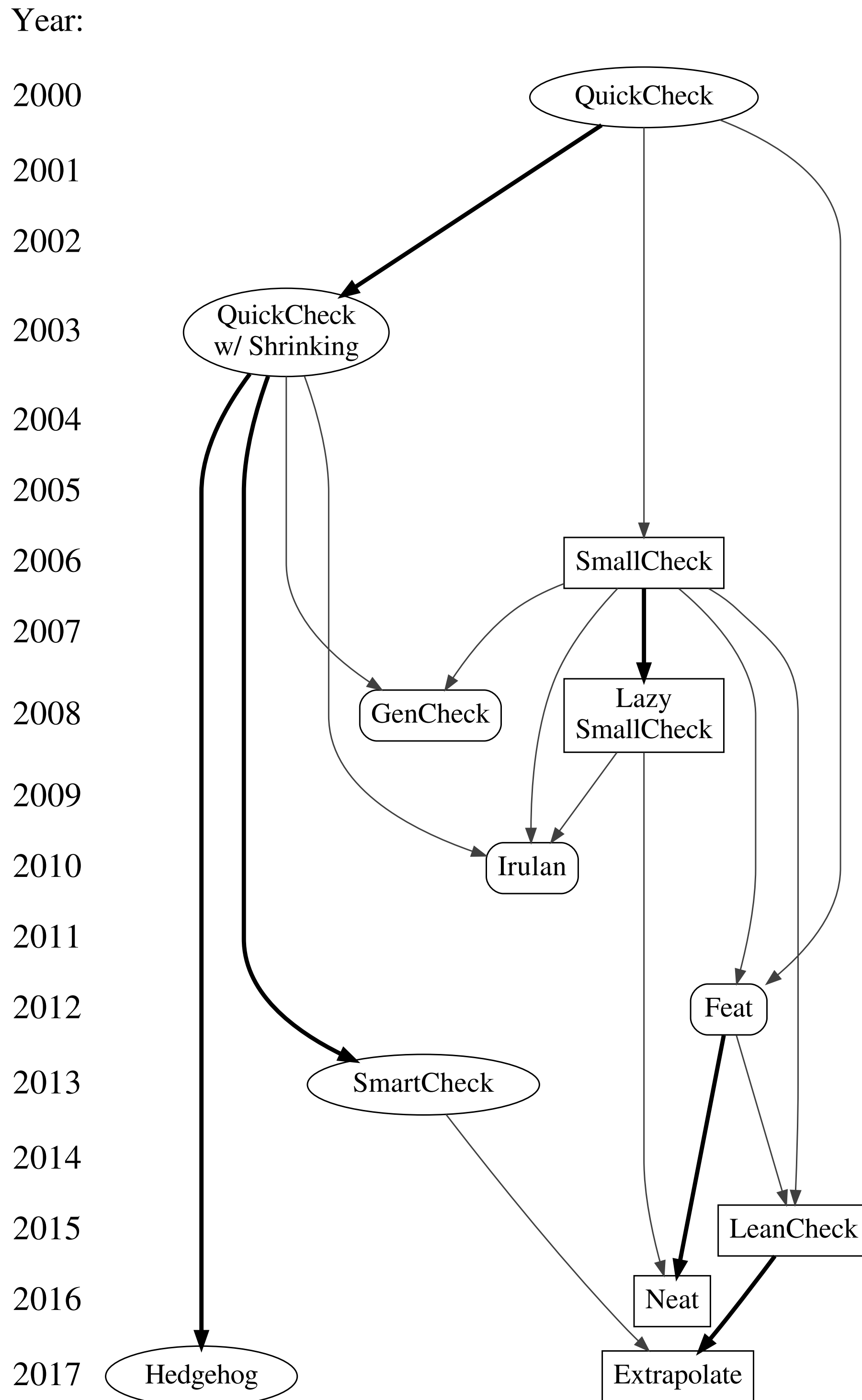
Some early works introduced ideas very similar to property-based testing[1]. One of these works is the DAISTS system [Gannon et al., 1981] in which algebraic properties about a data structure are written to be used both as specification and for testing. In the case of DAISTS though, the user had to manually insert the test cases.

Starting with QuickCheck in 2000 [Claessen and Hughes, 2000], a lot of research on property-based testing has been conducted using the Haskell programming language. Several tools for that purpose have been created, this section is a review of the most notable. Figure 2.1 shows relationships between these tools. Figure 2.2 shows property-based testing tools loosely placed along two axes: strict-lazy and enumerative-random. These tools are later compared in §2.1.12.

Haskell has been the one of main settings for research in this area perhaps for two reasons:

- Haskell is *purely functional*, results of functions only depend on their arguments. So testing a property once per argument value is enough;
- Haskell's typeclass machinery and partial application of functions make it convenient to generate test values to properties (§3.2–§3.3).

[1] Other terms used to describe property-based testing or similar techniques include: random testing [Claessen and Hughes, 2000], enumerative testing [Duregård et al., 2012], property testing [Allwood, 2011], specification-based testing [Stocks and Carrington, 1996], contract-based testing [Aichernig, 2003] and parameterized unit testing [Tillmann and Schulte, 2005]



An arrow from A to B indicates B uses ideas of and/or is inspired by A.
Round, square and rounded nodes indicate random, enumerative and mixed test data generation.

Figure 2.1: Flow of ideas between property-testing tools for Haskell

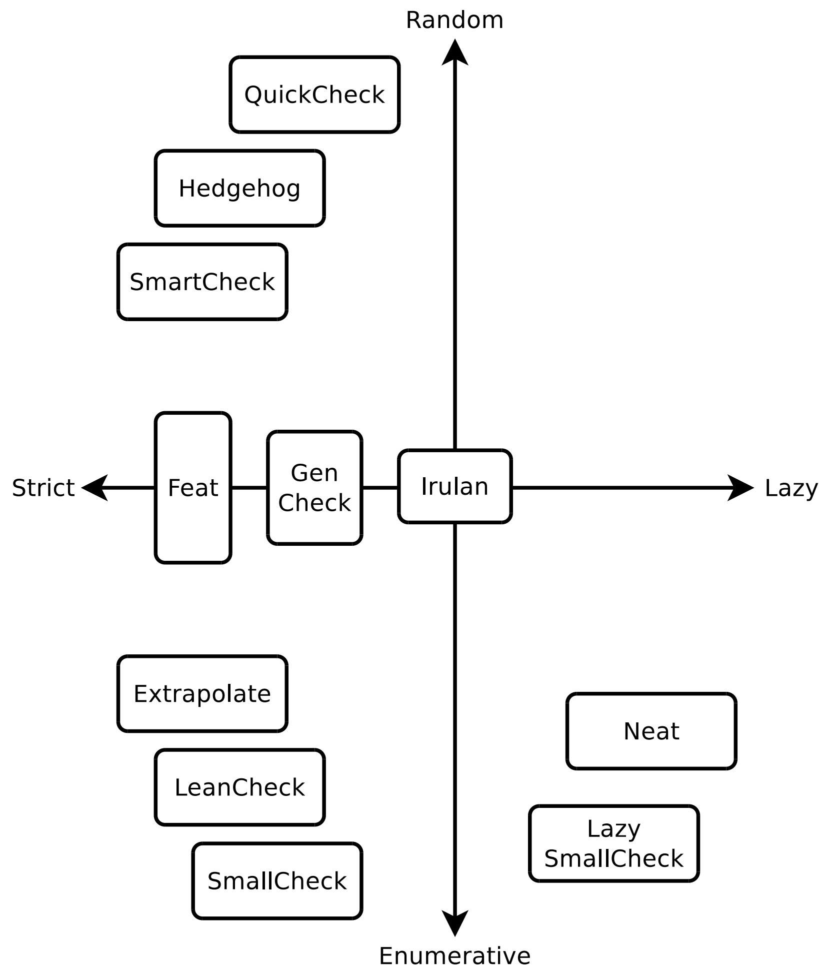


Figure 2.2: Property-based testing tools placed within two axes: strict–lazy and enumerative–random.

2.1.1 QuickCheck: automated random testing

Claessen and Hughes [2000] introduced the first tool for property-based testing in Haskell, QuickCheck. It generates test values randomly based on the types of arguments of the properties being tested.

Testing properties with QuickCheck Recall the faulty `sort` function given in Chapter 1 and a property stating that the counts of elements should not change after sorting:

```
sort :: Ord a => [a] -> [a]
sort []      =  []
sort (x:xs)  =  sort (filter (< x)) xs
             ++ [x]
             ++ sort (filter (> x)) xs
```

```
prop_sortCount :: Ord a => a -> [a] -> Bool
prop_sortCount x xs = count x (sort xs) == count x xs
```

Testing with QuickCheck:

```
> quickCheck (prop_sortCount :: Int -> [Int] -> Bool)
*** Failed! Falsifiable (after 25 tests and 5 shrinks):
2 [2,2]
```

Again, our `sort` implementation had an intentional fault: it does not account for lists with repeated elements. The counterexample is able to illustrate this:

```
> prop_sortCount 2 [2,2]
False
> count 2 (sort [2,2])
1
> sort [2,2]
[2]
```

Since test values are generated randomly, multiple executions of QuickCheck are not guaranteed to find the same failing cases:

```
> quickCheck (prop_length :: [Int] -> Bool)
*** Failed! Falsifiable (after 4 tests and 2 shrinks):
7 [7,7]
```

□

Generating test values QuickCheck is able to generate test values for instances of the `Arbitrary` typeclass. QuickCheck already provides `Arbitrary` instances for most standard Haskell types.

QuickCheck provides a series of combinators that allow definition of `Arbitrary` instances for user-defined types. For example, suppose a `Nat` type defined as a wrapper over `Int`s:

```
newtype Nat = Nat Int  deriving  (Show, Eq, Ord)
```

An `Arbitrary` instance for `Nat` may be defined, using Haskell's monadic do notation, as:

```
instance Arbitrary Nat where
  arbitrary = do
    x <- choose 0 12
    return (Nat x)
```

that is, to generate a `Nat` value, choose an `Int` value between 0 and 12 (inclusive) then apply the `Nat` constructor. □

For some types, it might be difficult to write instances that properly explore the test space. This is recognized as one of the drawbacks of QuickCheck [Duregård et al., 2012] and QuickCheck authors recommend analysing the distribution of generated test data.

Small counterexamples and shrinking The original version of QuickCheck sometimes returned quite large counterexamples due to the nature of random testing. Later versions introduced *shrinking* [Claessen, 2012], a technique to reduce the size of counterexamples. `Arbitrary` instances now can define the `shrink` function of type `a -> [a]`. Given a value, `shrink` produces a finite list of values which are, in some sense, like the original but a little smaller. What smaller means here is decided by the implementor of the `Arbitrary` typeclass instance.

The following two runs of QuickCheck illustrate the benefit of shrinking. The first run has shrinking turned off, as in the original QuickCheck, the second has shrinking activated:

```
> quickCheck (noShrinking (prop_sortCount :: Int -> [Int] -> Bool))
*** Failed! Falsifiable (after 15 tests):
-14 [-9,-14,-4,-10,6,-13,-14,10,-2]
> quickCheck (prop_sortCount :: Int -> [Int] -> Bool)
*** Failed! Falsifiable (after 65 tests and 10 shrinks):
-2 [-2,-2]
```

The problem of shrinking functions as test values is discussed and solved by Claessen [2012].

QuickCheck has become the de-facto standard for property-based testing in Haskell. It is a well documented tool featuring in the Real World Haskell Book [O'Sullivan et al., 2008] and some tutorials [Claessen et al., 2003, Hughes, 2010].

The original QuickCheck authors have created a version for Erlang [Arts et al., 2006], QuviQ QuickCheck. It allows for integrated shrinking: users get the shrink for free by just writing a generator; shrunk values obey data invariants by construction.

2.1.2 SmallCheck: exhaustive testing for small values

With principles similar to QuickCheck, SmallCheck [Runciman et al., 2008] is another property-based testing tool for Haskell. But, instead of generating test values randomly, SmallCheck generates test values enumeratively up to some limiting depth. SmallCheck intentionally starts testing properties for small values. The choice to start with small values is based on the *regularity hypothesis* [Bougé et al., 1986, Zhu et al., 1997]: if a program satisfies a property on all test arguments up to some depth, then the program very likely satisfies the property on all data. In the context of property-based testing, this hypothesis means that counterexamples are almost always found for small values. The *same minimal* counterexample is returned across different runs without the need for shrinking.

Running SmallCheck SmallCheck is run in a similar way to QuickCheck:

```
> smallCheck 3 (prop_length :: [Int] -> Bool)
Failed test no. 4:
0 [0,0]
```

Properties written for QuickCheck can be tested by SmallCheck with little or no alteration. Search is limited by depth of test values rather than number of test cases.

Generating test values SmallCheck is able to test types which are instances of the `Serial` typeclass, parallel to QuickCheck's `Arbitrary`. `Serial` instances define a `series` function which is a mapping of a integer depth to a list of values. Given its simplicity, `Serial` instances are easier to write than `Arbitrary` instances.

Existential properties Since SmallCheck performs an exhaustive search, it provides existential operators, `exists` and `exists1`. These operators are used to express existential properties, like the following property over naturals:

```
prop_lessThanExists :: Nat -> Nat -> Bool
prop_lessThanExists x y = x <= y ==> exists $ \z -> x + z == y
```

If `x` is less than `y`, there exists a `z` such that the sum of `x` and `z` is equal to `y`.

In QuickCheck, existential predicates must be re-expressed as universal ones by introducing a *Skolem*[2] function:

```
prop_lessthanexists :: Nat -> Nat -> Nat
prop_lessthanexists x y = x <= y ==> x + skolem x y == y
  where
  skolem x y == x - y
```

Depending on the property being tested, the Skolem implementation may not be obvious.

Limitations One of the limitations of SmallCheck is the rapid explosion of values on increasing depth. Usually, when searching deeper, the next depth has one or more orders of magnitude more values than the previous one. This is specially problematic for very wide data types[3], for example, the Template Haskell AST. Duregård et al. [2012] show that when testing properties for those types, SmallCheck is not able to find even some very simple faults.

2.1.3 Lazy SmallCheck: using laziness to guide enumeration

In the same paper as SmallCheck, Runciman et al. [2008] present Lazy SmallCheck. Similarly to SmallCheck, Lazy SmallCheck does property-based testing by exhaustive search up to some limiting depth. However, it uses laziness and partially defined values to guide testing, greatly reducing the number of cases that need to be tested.

As an example, let's first take a look at a function that checks if a list is ordered:

```
ordered :: Ord a => [a] -> Bool
ordered []        =  True
ordered [x]       =  True
ordered (x:y:xs) =  x <= y  &&  ordered (y:xs)
```

[2] In the process of *Skolemization*, existential quantifiers are removed by introducing a *Skolem* function that returns a witness for an existential statement using the universal variables.

[3] in which some constructors take a large number of arguments, such as Template Haskell's [Sheard and Jones, 2002] `DataD` constructor: `DataD Cxt Name [TyVarBndr] [Con] [Name]`

This function is lazy. In other words, when passed the list `[2,1,4,5,6,7,8]`, the function will check only the first two elements, and as these are out of order, a `False` result is returned without evaluating the rest of the list.

Now, let's look at a property that uses the `ordered` function. For any ordered list, sorting it will keep it the same:

```
prop_noop xs  =  ordered xs ==>  xs == sort xs
```

If this property is tested using SmallCheck or QuickCheck, many test values for `xs` do not satisfy the antecedent: many unordered lists are generated. Lazy SmallCheck cuts search space by first testing partially defined values, those containing ⊥. Since `2:1:`⊥ do not satisfy the condition in `prop_noop`, no list starting with `2` followed by `1` will. So SmallCheck does not enumerate them.

Lazy SmallCheck can provide speedups over SmallCheck up to a few orders of magnitude for the same depth. For `prop_noop` tested at depth 8, SmallCheck does 4886521 tests while Lazy SmallCheck does only 65525 tests.

Reich et al. [2013] add new features to Lazy SmallCheck, including the ability to test functional values, support for existential properties, nested quantification and the display of partial counterexamples.

Limitations As it is based on laziness, Lazy SmallCheck's search space pruning does not work for strict properties, those that need to fully evaluate all arguments to return `True` or `False`.

2.1.4 Feat: Functional Enumeration of Algebraic Types

Feat [Duregård et al., 2012, Duregård, 2012, 2016] is a size-based enumerative property-based testing tool. It does not suffer from the search space explosion problems of SmallCheck as values are explored by size rather than depth affording greater control on the number of values being tested. Due to the way it is implemented, it allows random generation of test values.

Enumerating test values Feat works for types that are instances of the `Enumerable` type class: those have an `enumerate` function. Predefined `Enumerable` instances for standard Haskell types are provided. The user can define enumerable instances using several available combinators. The resulting code is very similar to SmallCheck's.

Feat enumeration is based on an efficient indexing function `index :: Int -> a` that maps an non-negative integer to a value in the enumeration. Feat partitions the set of values by their *size* to obtain a function `select :: Int -> Int -> a`, that maps the *size* and index of values of that *size* into a value. The size of a value is an *arbitrary* measure depending on the type and defined in the `Enumerable` implementation, but usually counts the number of constructors.

Table 2.1: Numbers of integer lists in successive sizes (for Feat) or depths (for SmallCheck).

SmallCheck's depth	1, 2, 7, 36, 253, 2278, 25059, 325768, 4886521, 83070858, ...
Feat's size	0, 1, 0, 0, 2, 2, 4, 12, 24, 52, 120, 264, 584, 1304, 2896, ...

Generating random test values Feat is able to compute arbitrary values in an enumeration very efficiently. When enumerating `[Bool]`s, It takes less than a second to compute the $(10^{10000})^{\text{th}}$ element:

```
> index (10^10000) :: [Bool]
[False,False,True,True,False,True,True,True,False,False,...]
(0.71 secs, 590625624 bytes)
```

To generate random test values using Feat, it is enough to map the `index` function into a set of random integers.

Testing properties To test properties using Feat, the programmer should call the `featCheck` function, that accepts a size limit and a property to be tested:

```
> featCheck 10 (prop_sortOrdered :: [Int] -> Bool)
--- Done. Tested 97 values
```

Controlling the enumeration As the size increases, the number of generated values grows much more slowly than it does with increasing depth in SmallCheck. For wider types, Feat provides a fine-grained control on the number of generated test cases. Table 2.1 shows the progression in the number of enumerated values using both methods for integer lists.

Comparing with SmallCheck Duregård et al. [2012] compare Feat and SmallCheck in a case study with properties over Template Haskell's AST values. SmallCheck is not able to find some very small counterexamples that Feat does find. This is due to the wideness of AST values and the exploding depth of SmallCheck. Duregård et al. exclude QuickCheck from the study mentioning the difficulty of writing a sensible `Arbitrary` instance for TH's AST values. They also exclude Lazy SmallCheck as the tested properties were strict in all arguments.

Limitations Feat's Hackage package `testing-feat` [Duregård, 2014] provides only the exhaustive testing strategy: the random and mixed strategy must be implemented in an application specific way after importing the library. Feat does not support enumeration of functions: the authors note the relative difficulty of finding a suitable definition of size for functions.

2.1.5 Neat: Non-strict Enumeration of Algebraic Types

Neat [Duregård, 2016] mixes the idea of size-bounded enumeration of Feat with a clever algorithm for search space pruning. Neat can be roughly seen as a size-bounded variation of Lazy SmallCheck. Duregård [2016] provides several examples where Neat is able to find faults where Lazy SmallCheck cannot.

2.1.6 GenCheck: generalized testing

Gordon J. Uszkay and Jacques Carette [2012] describe GenCheck as a *generalized* property-based testing tool, able to perform random and enumerative testing. It has been developed during the same period as Feat, and the two tools have similar goals.

Generating test values GenCheck's generators are represented as a function from a *rank* to a list of values. For lists of integers, the rank represents the length of the resulting generated lists:

```
> take 5 $ genAll stdTestGens 0 :: [[Int]]
[[],[],[],[],[]]
> take 5 $ genAll stdTestGens 1 :: [[Int]]
[[-100],[-99],[-98],[-97],[-96]]
> take 5 $ genAll stdTestGens 2 :: [[Int]]
[[-100,-99],[-98,-97],[-96,-95],[-94,-93],[-92,-91]]
```

Using the same generator for types, one can use different strategies for test value generation by calling a different top-level function. Some strategies test ranks enumeratively. Other strategies selects random values from ranks uniformly.

Limitations As yet there is no published comparison of GenCheck with other property-based testing tools.

2.1.7 Irulan: implicit properties

The tools presented so far all require the tester to provide *explicit* properties over the functions being tested. Allwood [2011] notes that there are already many *implicit* properties expected to hold for all programs. These properties include the absence of assertion errors, incomplete pattern matches and other types of exceptions.

Generating test data and testing implicit properties Irulan infers how to generate test data from the types of constructors; Then, for the module under test, it checks and reports whether any exported functions throw exceptions.

Testing explicit properties Irulan is also able to do property testing. It searches then tests any functions beginning in `prop_` of resulting `Bool` type.

Limitations If functions expect that its arguments follow a constraining data invariant, Irulan may falsely report a bug. Allwood recommends to flag these type of errors with specific exceptions, then filter them out from the resulting Irulan output.

Irulan is currently unmaintained. The latest version, released in 2010, only works if compiled using older versions of the Glasgow Haskell Compiler (6.* series).

2.1.8 Reach: finding inputs that Reach a target expression

Naylor and Runciman [2007] describe Reach, a tool that uses *lazy narrowing* to find inputs that cause evaluation of marked target expressions. to find inputs that evaluate marked target expressions. By marking an expression with an application of the special identity function target, the user requests inputs that *reach* that path of execution. Given an expression:

```
if ordered xs
  then target (...)
  else ...
```

Reach tries to find values for `xs` that reach the "`then`" branch.

Reach has more general applications. It can be used to find inputs that cause a program to crash by marking positions that have unspecified behavior, e.g.: missing cases on pattern matches. It can be used as an input data generator for other testing tools. Lastly, by using a specially crafted identity function over booleans, Reach can be used directly for property-based testing:

```
refute :: Bool -> Bool
refute True  = True
refute False = target False
```

Then, to refute a property, it is enough to create an entry point that wraps around the property being tested:

```
main :: Int -> [Int] -> Bool
main x xs = refute (prop_sortCount x xs)
```

Limitations This tool does not work directly on Haskell, but instead in a core language that can be generated from a Haskell Program by the York Haskell Compiler.

2.1.9 SmartCheck: improving counterexamples

Initially motivated by the difficulty of writing custom instances for the `shrink` function (§2.1.1), specially on very complex data types, Pike [2014] introduces SmartCheck. SmartCheck improves QuickCheck in two main points: a better algorithm for shrinking and generalization of counterexamples.

Smart Shrinking QuickCheck's shrinking algorithm is deterministic and usually returns shrunk counterexamples that have the same constructors as the original. SmartCheck's shrinking algorithm is non-deterministic and does not restrict shrunk values to have the same constructors as the original. According to Pike's results, SmartCheck's shrinking outperforms QuickCheck's.

Counter-example generalization Even with a small counterexample, it may not be simple to determine the cause of a fault. So after obtaining a counterexample, SmartCheck *generalizes* it by exploring possibilities of sub-values. If we run SmartCheck on the property `prop_sortCount`, we get:

```
> smartCheck scStdArgs {format=PrintString} (uncurry prop_sortCount)
*** Failed! Falsifiable (after 4 tests):
*** Smart-shrunk value:
(2,[2,2])

*** Extrapolated value:
forall values x0:
(,) 2 (: 2 (: 2 x0))
```

In other words, the property fails for a pair of arguments of the form `2` and `2:2:x0`. The `prop_sortCount` property is strict on its arguments, as the sorting function needs to evaluate the whole list to return a value. The ability to generalize a counterexample even in a strict property is an advantage over Lazy SmallCheck.

Limitations SmartCheck is not able to generalize functional counterexamples. Its generalized counterexamples do not allow for repeated variables and side conditions – something we explore in Chapter 6.

2.1.10 Hedgehog: integrated shrinking

Stanley [2017] describe Hedgehog as "a modern property-based testing system, in the spirit of QuickCheck". Similarly to QuviQ QuickCheck, it allows for integrated shrinking: users get the shrink for free by just writing a generator; shrunk values obey data invariants by construction. Differently from other property-based testing tools for Haskell, generators are not inferred from types but instead defined alongside properties.

2.1.11 Beyond Haskell

This section notes some work on property-based testing beyond the Haskell programming language.

Curry EasyCheck [Christiansen and Fischer, 2008] is a property-based testing tool for Curry. It explores Curry's features of non-determinism to implement searches for counterexamples. It allows exhaustive testing and random testing through shuffling of counterexamples search trees.

Erlang For Erlang, aside from the previously mentioned QuviQ QuickCheck [Arts et al., 2006], theres PropEr [Papadakis and Sagonas, 2011].

MoreBugs: find more bugs with QuickCheck [Hughes et al., 2016] describes an extension to QuickCheck to avoid rediscovering the same bugs. As each bug is found, they are generalized as a bug pattern. Then, further test cases matching a bug pattern are never generated again.

Clean Koopman et al. [2003] present GAST, a property-based testing tool for the Clean programming language.

Isabelle/HOL When developing theories about programs using a theorem prover, incorrect specifications are often found during failed proof attempts. Such failures can be very time consuming. Motivated by this problem, Berghofer and Nipkow [2004] designed a version of QuickCheck for Isabelle/HOL statements. Before starting lengthy proofs for statements, Isabelle QuickCheck can be used to search for counterexamples. If the statement is wrong, this may save the time that would be wasted in a failed proof attempt. The tool is able to find counterexamples in both case studies. Isabelle QuickCheck is further extended by Bulwahn [2012, 2013] with support for exhaustive testing and a narrowing-based testing approach designed to improve efficiency.

Racket New et al. [2017] describe fair enumeration combinators for Racket with ideas similar to Feat (§2.1.4) [Duregård et al., 2012]. They define a notion of fairness that can be used to identify which enumerations are better suited for property-based testing.

Scala Nilsson [2014] describe ScalaCheck, a property-based testing tool for Scala or Java programs. It performs tests randomly, is able to shrink failing cases and supports testing of stateful functions.

Clojure Rich Hickey and Reid Draper [2013–2017] present test.check, a randomized property-based testing tool for Clojure. Similarly to QuviQ QuickCheck, it allows for integrated shrinking: users get the shrink for free by just writing a generator; shrunk values obey data invariants by construction.

Usability in imperative languages Although still useful, property-based testing is a bit less useful in the realm of imperative programming languages as property-based testing benefits from testing functions and modules that have no side-effects.

.NET Framework: Parameterized Unit Tests Tillmann and Schulte [2005] describe the idea of parameterized unit tests and a system to test programs on the .NET framework. Their ideas are very similar to property-based testing: "Test methods are generalized by allowing parameters."; "[these methods] describe a set of traditional unit tests which can be obtained by instantiating the methods with given argument sets". However, instead of generating test data randomly or enumerative, Tillmann and Schulte use symbolic execution of .NET assemblies to both execute tests and select input values and may sometimes require the user to provide input values for testing.

Interestingly, Tillmann and Schulte [2005] did not seem to be aware of the work in QuickCheck [Claessen and Hughes, 2000] at the time of publication of their paper: it is neither cited nor compared.

Java: unit tests maximizing coverage Pacheco and Ernst [2007] and Fraser and Arcuri [2011] describe RANDOOP and EvoSuite, two test suite generation tools for the Java programming language. Given a Java program, RANDOOP and EvoSuite will produce unit tests that maximize line and branch coverage. These are not property-based testing tools as test properties are not involved. Although test generation involves randomness, it is directed by coverage.

Concolic testing Godefroid et al. [2005] and Sen et al. [2005] describe the DART and CUTE techniques. Based on the results of a syntactic and symbolic analysis of a program, these techniques generate test values while trying to maximize code coverage. This is different from property-based testing, where test values are generated only based on property types thus not requiring access to source code or imposing a language subset on tested programs.

2.1.12 Discussion and Comparison

Several tools for property-based testing were presented in this chapter. Again, Figure 2.1 shows the flow of ideas between property-based testing tools and Figure 2.2 shows property-based testing tools placed along two axes: random–enumerative and strict–lazy. A summary of several distinguishing features is shown in Table 2.2. The starting point for this table was a table by Reich et al. [2013]. The table includes Irulan and Reach, tools with a broader scope of applications: but here evaluated in the context of property-based testing. We have included *LeanCheck* and *Extrapolate*, two of the tools developed in this thesis discussed in Chapters 3 and 6.

Test Data Generation QuickCheck, SmartCheck and Hedgehog generate test values randomly. SmallCheck, Lazy SmallCheck, LeanCheck, Extrapolate and Neat provide only exhaustive testing. Feat, GenCheck and Irulan are more flexible and permit either enumerative or random testing. Tools with size-bounded enumeration offer greater control

Table 2.2: Summary of differences between property-based testing tools for Haskell.

	QuickCheck	SmartCheck	Hedgehog	SmallCheck	Lazy SC	**LeanCheck**	**Extrapolate**	Feat	Neat	GenCheck	Irulan	Reach
Test data generation												
random	●	●	●	○	○	○	○	●	○	●	●	–
enumerative	○	○	○	●	●	●	●	●	●	●	●	–
depth-bounded	○	○	○	●	●	○	○	○	○	○	●	–
size-bounded	○	○	○	○	○	●	●	●	●	●	○	–
mixed random & enumerative	○	○	○	○	○	○	○	●	○	●	●	–
demand-driven / directed	○	○	○	○	●	○	○	○	●	○	●	●
generator instance auto-derivation	●	●	○	●	●	●	●	●	●	○	●	–
Features												
Existential properties	○	○	○	●	●	●	●	○	○	○	○	○
Higher order properties	●	●	○	●	●	●	○	○	○	○	○	○
Generalized counterexamples	○	●	○	○	●	○	●	○	○	○	○	○
Ease of use/writing generators	○	○	○	●	●	●	◐	◐	◐	◐	–	–
Availability												
Cabal package available on Hackage	●	●	●	●	◐	●	●	●	●	●	○	○
Compilable with GHC 8.0 (2017)	●	●	●	●	◐	●	●	●	●	○	○	–
Compilable with GHC 7.8 (2014)	●	●	●	●	●	●	●	●	○	○	○	–
Licensing	BSD3	BSD3	BSD3	BSD3	BSD3	BSD3	BSD3	BSD3	BSD3	BSD3	GPLv3	–

Legend: ● Yes/Good. ○ No/Poor. ◐ Partial/Median.

on the number of test cases. Lazy SmallCheck, Neat, Irulan and Reach are able to eliminate test cases, either by testing partial values or lazy narrowing. Most of those tools can automatically derive generator instances for types which do not have a constraining data invariant.

Writing generator typeclass instances for enumerative property-based testing tools is arguably easier. One of the reasons for this is the need for shrinking on random testing tools.

Existential and higher-order properties Enumerative testing tools have the advantage of allowing for existential properties. SmallCheck, Lazy SmallCheck, LeanCheck and Extrapolate support existential properties out-of-the-box. A few tools support higher-order properties, those that take functions as argument values.

Generalized counterexamples Three tools provide support for generalized counterexamples by using different strategies: Lazy SmallCheck, SmartCheck and Extrapolate. Lazy SmallCheck does that by providing *partial* counterexamples, that is, counterexamples in which the property did not even evaluate part, or parts, of the arguments, even for functional values. SmartCheck on other hand, is able to provide generalized counterexamples by testing a function fixing some part of the arguments. This strategy is more general but risks reporting fake counterexamples. No work has been done yet in doing that for functional values.

Availability Most tools are freely available with open source licences, either BSD-style[4] or GPLv3[5]. Most tools have a package available on the Hackage[6] package repository. GenCheck, Irulan, Reach and the updated Lazy SmallCheck do not compile or work under the latest GHC.

2.2 Applications of Property-based testing

Property-based testing can be applied beyond its original scope of simply testing programs. For example, it can be used as part of a process to automatically generate program specifications. This section reviews works using property-based testing as a component.

2.2.1 QuickSpec: discovery of equational laws

QuickSpec [Claessen et al., 2010, Smallbone, 2011, 2013, Smallbone et al., 2017] is a tool to automatically conjectures equational specifications for Haskell programs. These specifications can contribute to documentation, understanding and testing. Individual properties can be used as test properties for QuickCheck and similar tools.

QuickSpec accepts a set of *functions* to describe and a set of *background functions* allowed to appear in laws. QuickSpec produces a set of equations describing the given functions.

Example application If we pass (a correct) `sort` to QuickSpec, with `[]`, `length`, `ordered`, `elem` and `count` in the background, it reports:

```
1. sort [] = []
2. length (sort xs) = length xs
3. ordered (sort xs) = True
4. sort (sort xs) = sort xs
5. count x (sort xs) = count x xs
6. elem x (sort xs) = elem x xs
```

[4] `https://opensource.org/licenses/BSD-3-Clause`
[5] `https://www.gnu.org/licenses/gpl-3.0.html`
[6] `https://hackage.haskell.org/`

How QuickSpec works QuickSpec works by testing and lightweight reasoning. It builds *terms* of increasing size from type correct applications of given functions. It tests these terms for equality using the configured number of value assignments to discover apparent equalities. These equalities are reported to the user and used to avoid redundant testing. In Chapter 5, we will develop a similar tool and give details on a QuickSpec-like algorithm.

Limitations QuickSpec is not able to formulate inequality laws (>=, >, <, <=) directly. The original version of QuickSpec [Claessen et al., 2010] did not support conditional equations. Although QuickSpec 2 [Smallbone et al., 2017] does support it, conditions are restricted to a set of predicates. QuickSpec has support for *observational equalities* which can be used to compare values without `Eq` instances, such as functions or regular expressions.

2.2.2 EasySpec: signature inference for property discovery

Kerckhove [2017] describes EasySpec, a tool to make QuickSpec easier to use by automating the process of deciding which background functions to use for QuickSpec. Given a *focus function* we are interested in knowing properties about and a collection of modules, EasyCheck automatically decides which background functions to use and passes those to QuickSpec, reporting the results. It offers a simple command line interface, running

```
easyspec discover Sort.hs sort
```

is enough to discover properties of a `sort` function placed in a file called `Sort.hs`.

EasySpec works by running QuickSpec several times with smaller subsets of the universe of available background functions selecting functions that are "better" according to some heuristics.

2.2.3 Bach: discovering relational specifications

Bach [Smith et al., 2017] is a technique to discover likely relational specifications based on test results. These relational specifications include commutativity, transitivity and equivalence between functions. Smith et al. [2017] provide an implementation in OCaml, for OCaml functions.

2.2.4 MuCheck: syntactic mutation testing for Haskell

MuCheck [Le et al., 2014, 2013, Le et al.] is a tool to check for comprehensiveness of properties as test criteria by mutation testing. A series of syntactic mutations is introduced in the program being tested using mutation operators provided by the user. Then, the properties under evaluation are tested against all the mutations. Any surviving mutants might indicate a missing property about the program under test.

A diagram showing this functionality is shown in Figure 2.3: `sort4`, a mutant variation of `sort`, has survived all tests. This mutant is either equivalent to the original `sort` or indicates a missing property.

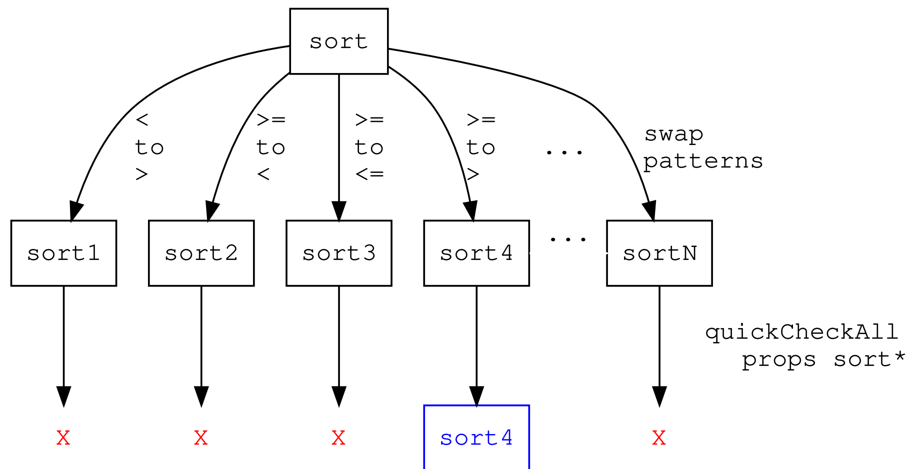


Figure 2.3: MuCheck applied to a sorting function

2.2.5 Lightweight Mutation Testing in Haskell

Duregård [2016] describes "a tiny mutation testing framework" that works on top of QuickCheck. Differently from MuCheck, mutations are black-box, not relying on source-code changes. At runtime, for a random selection of argument values, result values are randomly mutated. Because of this, Duregård's framework has full language support and does not rely on a subset of the Haskell language. Also, it avoids the problem of equivalent mutants [Jia and Harman, 2011] by design.

Because this technique works by damaging result values, it will not be able to generate mutants that MuCheck otherwise would. An experimental comparison could be done to evaluate the practical differences between MuCheck and Duregård's technique.

2.3 Summary

In this chapter, we presented a summary of recent work in the area of property-based testing. §2.1 presented several tools for property-based testing. §2.2 presented a few tools that apply property-based testing beyond its original scope of simply testing properties.

Chapter 3

LeanCheck: enumerative testing of higher-order properties

This chapter presents the core of LeanCheck, a tool for property-based testing in Haskell. LeanCheck generates test values of increasing size enumeratively. Using a partial enumeration of functions, it is able to test higher-order properties. Although its enumeration strategy has similarities to Feat (§2.1.4), the ranking and ordering of values are defined differently to align better with our needs when enumerating functions (§3.5) and functional mutants (Chapter 4).

Tools developed in further chapters, FitSpec (Chapter 4), Speculate (Chapter 5) and Extrapolate (Chapter 6) all use LeanCheck to test properties.

This chapter is partly based on an unpublished draft paper [Braquehais et al., 2017].

3.1 Introduction

LeanCheck is an enumerative property-based testing tool for Haskell. It is size-bounded and allows testing of higher-order properties. Although its enumeration strategy has similarities to Feat [Duregård et al., 2012], the ranking and ordering of values are defined differently to align better with our needs when enumerating functions (§3.5) and functional mutants (Chapter 4).

Roadmap In this chapter we describe the implementation of LeanCheck:

- how data values are enumerated (§3.2);
- how properties are tested (§3.3);
- how to deal with conditional properties and data invariants (§3.4);
- how higher-order properties are tested and functions are enumerated (§3.5).

We show a couple of example applications with higher-order properties (§3.6). Then, we draw some conclusions and suggest future work (§3.8).

3.2 Listable Data Types and Fair Enumeration

Parallel to QuickCheck's `Arbitrary`, SmallCheck's `Serial` and Feat's `Enumerable` type-classes, we define `Listable`:

```
class Listable a where
  tiers :: [[a]]
```

A `Listable` instance's `tiers` value is a possibly infinite list of *finite* sublists of values characterised by some notion of *size*. Each sublist represents a tier: the first tier contains values of size 0, the second tier contains values of size 1, and so on. Size varies with the type being enumerated: for tuples, it is the sum of component sizes; for algebraic data types, the derivable default definition of size is the number of constructor applications of positive arity. If we wish to list all values of a type, we can simply concatenate `tiers`:

```
list :: Listable a => [a]
list = concat tiers
```

Defining `Listable` instances Listable instances can be defined using a family of functions `cons<N>` and an operator `\/`. Each function `cons<N>`, takes as argument a constructor of arity `N`, each of whose argument types is `Listable`, and returns tiers containing all possible applications of the constructor. The operator `\/` produces the sum of two lists of tiers. So, the general form of an instance for algebraic datatypes is:

```
instance <Context> => Listable <Type> where
  tiers = cons<N> ConsA
       \/ cons<N> ConsB
       ...    ...    ...
       \/ cons<N> ConsZ
```

The order between different constructors only affects the order of enumeration between same-sized elements. The form of expression, using `\/` to combine `cons<N>` applications, will be familiar to SmallCheck users: `tiers` and `series` declarations are similar.

Delaying enumeration The function `delay` is defined by:

```
delay :: [[a]] -> [[a]]
delay = ([]:)
```

It prepends an empty list to increase the size assigned to elements in a tier enumeration by one. So:

```
delay [[x,y],[a,b]] = [[],[x,y],[a,b]]
```

The function `delay` is useful when we take the product of tier-lists or define the `cons<N>` family of operators.

Sum of tier-lists The sum of two tier-lists is computed by the \/ operator:

```
(\/) :: [[ a ]] -> [[ a ]] -> [[ a ]]
xss       \/ []        =  xss
[]        \/ yss       =  yss
(xs:xss) \/ (ys:yss)  =  (xs ++ ys) : xss \/ yss
```

So, for example:

```
[xs,ys,zs] \/ [is,js,ks,ls]  =  [xs++is, ys++js, zs++ks, ls]
```

For tier-lists with the same number of tiers, \/ is equivalent to `zipWith (++)`.

Product of tier-lists The product of two tier-lists is computed by the >< operator:

```
(><) :: [[ a ]] -> [[ b ]] -> [[ (a,b) ]]
_         >< []    =  []
[]        >< _     =  []
(xs:xss) >< yss   =  [xs ** ys | ys <- yss]
                   \/ delay (xss >< yss)
  where
  xs ** ys = [(x,y) | x <- xs, y <- ys]
```

So, for infinite tier-lists:

```
[is,js,ks,...] >< [xs,ys,zs,...]  =  [ is**xs
                                     , is**ys ++ js**xs
                                     , is**zs ++ js**ys ++ ks**xs
                                     , ... ]
```

Note the use of `delay`. As we peel-off one tier in the pattern match to extract `xss`, we need to re-add it in the second argument of \/.

Constructing tiers Now, we define the `cons<N>` family of functions that build tier-lists from constructors.

Arity 0 The function `cons0` simply forms in a tier-list with a single tier containing a value of size 0:

```
cons0 :: a -> [[a]]
cons0 x = [[x]]
```

Arity 1 The function `cons1` maps a given constructor into a tiered enumeration, delaying it once.

```
cons1 :: Listable a => (a -> b) -> [[b]]
cons1 f = delay (mapT f tiers)
```

Table 3.1: Numbers of data values in successive tiers for several example data types.

Tier	Number of data values of type:				
	Bool	Nat	(Nat,Nat)	[Nat]	[[Nat]]
0	2	1	1	1	1
1	–	1	2	1	1
2	–	1	3	2	2
3	–	1	4	4	5
4	–	1	5	8	13
5	–	1	6	16	34
6	–	1	7	32	89
7	–	1	8	64	233
8	–	1	9	128	610

The function `mapT`, a variant of `map` for tier-lists, is defined as follows.

```
mapT :: (a -> b) -> [[a]] -> [[b]]
mapT = map . map
```

We use delay so that the application of the constructor argument “counts” on the size of the returned values. Additionally, without the application of `delay`, `cons1` would return an infinite starting tier when applied to a recursive type constructor like `(:)`.

Further arities For `cons2` and others, it is just a matter of uncurrying, mapping and delaying:

```
cons2 :: (Listable a, Listable b) => (a -> b -> c) -> [[c]]
cons2 f = delay (mapT (uncurry f) tiers)

cons3 :: (Listable a, Listable b, Listable c)
      => (a -> b -> c -> d) -> [[d]]
cons3 f = delay (mapT (uncurry3 f) tiers)
  where
  uncurry3 f (x,y,z) = f x y z
```

As `cons<2>`, . . . , `cons<N>` are defined by applying uncurried versions of constructors they need matching `Listable` tuple instances, defined in the next section.

Listable `Bools` The `Listable` instance for `Bools` is defined as follows:

```
instance Listable Bool where
  tiers = cons0 False
       \/ cons0 True
```

There are two `Bool` values, both of size 0:

```
tiers :: [[Bool]]  =  [[False,True]]
```

Listable `Nats` For the following natural-number type, defined as a wrapper over `Ints`,

```
newtype Nat = Nat Int
```

assuming a `Num` instance, a `Listable` instance can be defined by

```
instance Listable Nat where
  tiers = cons0 0
       \/ cons1 (+1)
```

so

```
tiers :: [[Nat]]  =  [ [0], [1], [2], [3], ... ]
```

as the size of each number is just the number itself — or equivalently, the number of applications of `(+1)` used to compute it.

Listable tuples We define `tiers` of pairs using a product of tiers of element values:

```
instance (Listable a, Listable b) => Listable (a,b) where
  tiers = tiers >< tiers
```

`tiers` of triples are defined by:

```
instance (Listable a, Listable b, Listable c) => Listable (a,b,c) where
  tiers = mapT (\(x,(y,z)) -> (x,y,z)) tiers
```

Instances for tuples of further arity are defined similarly.

The size of tuples is given by the sum of sizes of its component values. For pairs of `Nats`, for example, we have:

```
tiers :: [[(Nat,Nat)]]  =  [ [(0,0)]
                           , [(0,1),(1,0)]
                           , [(0,2),(1,1),(2,0)]
                           , [(0,3),(1,2),(2,1),(3,0)]
                           , ...
                           ]
```

Listable Lists The `Listable` instance for lists is defined as follows:

```
instance Listable a => Listable [a] where
  tiers = cons0 []
       \/ cons2 (:)
```

So, for example,

```
tiers :: [[ [Nat] ]]  =  [ [ [] ]
                         , [ [0] ]
                         , [ [0,0] ,[1] ]
                         , [ [0,0,0], [0,1], [1,0], [2] ]
                         , ... ]
```

is the tier-list for lists of natural numbers. The size of lists is given by the sum of sizes of elements added to the length of the list (number of constructor applications).

Table 3.2: Numbers of values in each tier for two alternative `Listable Int` instances. When using the absolute value as size (1), the enumeration of compound types containing `Int`s "blows-up" faster than with one-integer-per-tier (2).

(Enum.) – Type	Numbers of values for tier of size								
	0	1	2	3	4	5	6	7	8
(1) – `Int`	1	2	2	2	2	2	2	2	2
(2) – `Int`	1	1	1	1	1	1	1	1	1
(1) – `(Int,Int)`	1	4	8	12	16	20	24	28	32
(2) – `(Int,Int)`	1	2	3	4	5	6	7	8	9
(1) – `[Int]`	1	1	3	7	17	41	99	239	577
(2) – `[Int]`	1	1	2	4	8	16	32	64	128
(1) – `[[Int]]`	1	1	2	6	18	54	162	486	1458
(2) – `[[Int]]`	1	1	2	5	13	34	89	233	610

Example 3.1 For the following tree type

```
data Tree a = E | N a (Tree a) (Tree a)
```

we may define a `Listable` instance by

```
instance Listable a => Listable (Tree a) where
  tiers = cons0 E \/ cons3 N
```

so:

```
tiers :: [[ Tree Nat ]]  =
  [ [ E ]
  , [ N 0 E E ]
  , [ N 0 E (N 0 E E), N 0 (N 0 E E) E, N 1 E E ]
  , ...
  ]
```

□

Table 3.1 shows the number of values in each tier for several types. The ratios between these quantities for successive sizes is far smaller, for example, than the ratios between quantities of values for successive depths in SmallCheck — where an increase in depth may increase the size of a test-data set by orders of magnitude [Duregård et al., 2012].

Listable integers Here is one way we could define `Listable Int`:

```
instance Listable Int where
  tiers = [[0]] ++ [ [n,-n] | n <- [1..] ]
```

In this definition, the size of an integer is its absolute value. However, from a practical point of view, as we use `Int`s inside other structures, this definition of integer tiers makes the enumeration "blow-up" too quickly (Table 3.2). Having one `Int` per tier works better

in practice, even though the notion of size becomes less intuitive here: 0 has size 0, 1 has size 1, -1 has size 2, 2 has size 3, and so on, alternating between positives and negatives.

```
instance Listable Int where
  tiers = map (:[]) $ [0,-1..] `interleave` [1..]
    where
    interleave :: [a] -> [a] -> [a]
    []     `interleave` ys = ys
    (x:xs) `interleave` ys = x:(ys `interleave` xs)

> tiers :: [[Int]]
[[0],[1],[-1],[2],[-2],[3],...]
```

Convenience The value `tiers` can exist alongside `list` as methods, each has a default definition in terms of the other:

```
class Listable a where
  tiers :: [[a]]
  list  :: [a]
  tiers = map (:[]) list
  list  = concat tiers
```

So the user can define any `Listable` instance in the manner most convenient for their use-case. For types where a notion of tiers is not useful, defining only `list` reduces all tiers to singletons. For example, we can redefine a `Listable` instance for `Int` as follows:

```
instance Listable Int where
  list = [0,-1..] `interleave` [1..]
    where
    interleave :: [a] -> [a] -> [a]
    []     `interleave` ys = ys
    (x:xs) `interleave` ys = x:(ys `interleave` xs)
```

Automatic derivation of Listable instances Except when values have to follow a data invariant (§3.4), `Listable` instances follow a very simple pattern. Their production can be automated using Template Haskell [Sheard and Jones, 2002]. Using these techniques, we could derive an instance of the `Listable` typeclass for algebraic datatypes, with the following top-level declaration:

```
deriveListable ''Type
```

Because the definition of `deriveListable` is straightforward, albeit lengthy, we omit it here. It is provided as part of the LeanCheck package (§3.8).

3.3 Testable Properties and Tiers of Tests

Testable types We now define a `Testable` typeclass with one function, `resultiers`. Given a `Testable` property, it returns tiers of results.

```
class Testable a where
  resultiers :: a -> [[Result]]
```

The simpler `results` list can be obtained by concatenating `resultiers`:

```
results :: Testable a => a -> [Result]
results = concat . resultiers
```

The result type represents a test result by a pair: the first component is list of arguments and the second component is a boolean test result for those arguments:

```
type Result = ([String],Bool)
```

Testable booleans We can now define our first `Testable` instance — the type-level base case `Bool`. A boolean value is a property with no arguments, where the only test result is its value.

```
instance Testable Bool where
  resultiers p = [[([],p)]]
```

This `Bool` instance can be thought of as the base case for a type-level recursion.

Testable functions The recursive, and final, case is our instance for functions:

```
instance (Show a, Listable a, Testable b) => Testable (a -> b) where
  resultiers p = concatMapT resultiersFor tiers
    where
    resultiersFor x = (\(as,x) -> (show x:as,r)) `mapT` resultiers (p x)
```

For testable properties of type `a -> b`, the argument type `a` must have `Show` and `Listable` instances so that we can respectively represent the arguments as strings and generate test argument values. The result type `b` must be `Testable`: the partial application `p x` gives a specialised version of property `p` with `x` fixed as the test value for the first argument. As `(->)` associates to the right, `a -> (c -> Bool)` is the same as `a -> c -> Bool`, and we can instantiate `b` at a function type as long as the final result type is `Bool`.

The `concatMapT` function used to define `resultiers` is the tiered equivalent of `concatMap`:

```
concatT :: [[ [[a]] ]] -> [[a]]
concatT = foldr (\+:/) [] . map (foldr (\/) [])
  where
  xss \+:/ yss = xss \/ delay yss

concatMapT :: (a -> [[b]]) -> [[a]] -> [[b]]
concatMapT f = concatT . mapT f
```

Finding counterexamples of Testable values The counterExamples function lists any counterexamples of a property found by applying it to a limited number of test values:

```
counterExamples :: Testable a => Int -> a -> [[String]]
counterExamples m p = [as | (as,False) <- take m (results p)]
```

Using counterExamples, the checkFor function reports whether a property is true for a given number of test values. In case it is not, it reports a counterexample.

```
checkFor :: Testable a => Int -> a -> IO ()
checkFor n p =
  case counterExamples n p of
    []     -> putStrLn $ "+++ OK!"
    (ce:_) -> putStrLn $ "*** failed for: " ++ unwords ce
```

For convenience, the function check fixes the number of test values to a default (200).

```
check :: Testable a => a -> IO ()
check = checkFor 200
```

Testing properties With what has been defined so far, we can test the properties defined in Chapter 1:

```
> check (prop_sortOrdered :: [Int] -> Bool)
+++ OK
> check (prop_sortCount :: Int -> [Int] -> Bool)
*** Failure:  0 [0,0]
```

□

3.4 Conditional Properties and Data Invariants

Often we do not expect a property to hold in *all* cases, but only those which meet some precondition. For example, for non-negative values of x:

```
\x -> x == abs x
```

We can express such constraints either by embedding a precondition into a property itself or by applying an invariant condition in a generator.

Conditional properties We can define the logical implication operator as a normal Haskell function and use it to reformulate the abs property:

```
infixr 0 ==>
(==>) :: Bool -> Bool -> Bool
False ==> _ = True
True  ==> p = p
```

```
\x -> x >= 0 ==> x == abs x
```

This approach has the advantage of not needing any changes in the property testing tool, but has the significant downside that there is no reduction in the number of cases checked. As our `Listable` instance for `Int` alternates positive and negative values, only half of those values make it past the precondition. The property-testing tool does not care *how* the property passes or fails, only what the result is. A test case for which a property is true because the precondition failed is counted in the same way as a test case for which the actual condition of interest holds.

Data invariants Often we do not want to check a property for every possible value, but just pushing the precondition into the property leaves much to be desired. We can address this problem by instead restricting the generated values: the same number of test cases will be tried, but now they will *all* meet the precondition.

We can redefine the standard function `filter` to work over tiers:

```
filterT :: (a -> Bool) -> [[a]] -> [[a]]
filterT = map . filter
```

For convenience we define a flipped version:

```
suchThat :: [[a]] -> (a -> Bool) -> [[a]]
suchThat = flip filterT
```

that can be used when defining `Listable` instances of types that follow a data invariant. For example:

```
newtype NonNeg n = NonNeg n

instance (Listable n, Num n, Ord n) => Listable (NonNeg n) where
  tiers = cons1 NonNeg `suchThat` nonNegOk
    where
    nonNegOk (NonNeg n) = n >= 0
```

So,

```
> tiers :: [[NonNeg Int]]
[ [], [NonNeg 0], [NonNeg 1], [], [NonNeg 2], ... ]
```

The function `suchThat` generalizes nicely over types with several constructors, allowing different invariants for each (or none):

```
  tiers = cons<N> <Cons1> `suchThat` <someCondition>
       \/ cons<N> <Cons2>
       \/ cons<N> <Cons3> `suchThat` <someCondition>
       \/ cons<N> <Cons4>
```

We can use the `NonNeg` type in the definition of the `abs` property to ensure that only non-negative values are checked:

```
\(NonNeg x) -> x == abs x
```

Sets and Bags An auxiliary function `setsOf :: [[a]] -> [[ [a] ]]` takes as argument tiers of element values; it returns tiers of size-ordered lists of elements *without repetition*. For example:

```
setsOf (tiers :: [[ Bool ]]) =
  [ [ [] ]
  , [ [False], [True] ]
  , [ [False,True] ] ]

setsOf (tiers :: [[ Nat ]]) =
  [ [[]]
  , [[0]]
  , [[1]]
  , [[0,1],[2]]
  , [[0,2],[3]]
  , ... ]
```

Another similar auxiliary function `bagsOf :: [[a]] -> [[ [a] ]]` also takes as argument tiers of element values; but returns tiers of size-ordered lists of elements *possibly with repetition*.

The sizes of lists returned by `setsOf` and `bagsOf` is the same as lists returned by an unrestricted list enumeration: size is given by the sum of sizes of elements added to the length of the list (number of constructor applications).

Another similar auxiliary function `properSubsetsOf :: [[a]] -> [[ [a] ]]` also takes as argument tiers of element values; but returns tiers of proper sublists of values from a given tier-list.

The `setsOf`, `bagsOf` and `properSubsetsOf` functions will be useful when defining tiers of functions (cf. §3.5), tiers of mutants (cf. §4.4.1) and tiers of values satisfying a data invariant (cf. §4.5.3, §4.5.4, §4.5.6).

3.5 Higher Order Properties and Listable Functions

Functional programs often use higher-order functions like `map` and `filter`. As they take functional arguments, often properties about them also require functional arguments. In this section we define a partial enumeration of functions that allows testing properties with functions as arguments. This enumeration requires all arguments of enumerated functions to belong to the `Eq` class. So it is not able to enumerate *higher-order* functions.

Example 3.2 The following is an (incorrect) property from [Claessen, 2012] stating that `map` and `filter` commute:

```
prop_mapFilter :: Eq a => (a -> a) -> (a -> Bool) -> [a] -> Bool
prop_mapFilter f p xs  =  map f (filter p xs) == filter p (map f xs)
```

□

To test it, we need to define a `Listable` instance for functions (`a -> b`). However, a complete enumeration of functions over recursive types is known to be impossible [Kahrs, 2006]. Even for primitive recursive functions, an enumeration without repetition is not feasible in practice [Kahrs, 2006].

Mutating functions We can enumerate a useful though limited class of functions by starting with constant-valued functions, then adding exceptions. Each single-case mutation of a function is defined by an exception pair. The `mutate` function mutates a function given a list of exception pairs:

```
mutate :: Eq a => (a -> b) -> [(a,b)] -> (a -> b)
mutate f ms  =  foldr mut f ms
  where
  mut (x',fx') f x  |  x == x'     =  fx'
                    |  otherwise  =  f x
```

We shall use `mutate` again in the development of FitSpec in Chapter 4. There mutation is applied to given functions under test, not to constant-valued ones.

Enumerating exceptions The `exceptionPairs` function takes tiers of argument and result values, and gives tiers of lists of ordered pairs of argument and result values.

```
exceptionPairs :: [[a]] -> [[b]] -> [[ [(a,b)] ]]
```

For example:

```
> exceptionPairs (tiers :: [[Nat]]) (tiers :: [[Nat]])
[ [[]]
, [[(0,0)]]
, [[(0,1)],[(1,0)]]
, [[(0,2)],[(1,1)],[(0,0),(1,0)],[(2,0)]]
, ...
]
```

Here is how we define `exceptionPairs`:

```
exceptionPairs xss yss  =  concatMapT (`excep` yss) (properSubsetsOf xss)
  where
  excep :: [a] -> [[b]] -> [[ [(a,b)] ]]
  excep xs sbs  =  zip xs `mapT` products (const sbs `map` xs)
```

As the function `properSubsetsOf` returns tiers of proper sublists of values from a given tier-list, we avoid most but not all repetition.

Enumerating functions Now, using `mutate` and `exceptionPairs`, we are ready to enumerate tiers of functions. The combining operator `-->>` takes tiers of argument values and tiers of result values and gives tiers of functions.

```
(-->>) :: Eq a => [[a]] -> [[b]] -> [[a -> b]]
xss -->> yss = (\(r,yss) -> mapT (const r `mutate`)
                                 (exceptionPairs xss yss))
  `concatMapT` (choices yss)
```

The function `choices :: [[a]] -> [[(a,[[a]])]]` returns `tiers` of choices for result values. Each choice is a pair of an element taken from the argument tiers and a copy of the argument tiers without that element. Its definition is omitted here.

So, our `Listable (a -> b)` instance is just:

```
instance (Eq a, Listable a, Listable b) => Listable (a -> b) where
  tiers = tiers -->> tiers
```

As with the `Testable (a -> b)` instance in §3.3, the above definition suffices for function types of any arity: `Listable (a->b->c)`, `Listable (a->b->c->d)` and so on. Arguments must be instances of both `Eq` and `Listable`. Results must also be instances of `Listable`.

Example 3.3 These are the enumerated functions of type `Bool -> Bool`:

```
tiers :: [Bool -> Bool]  =
  [ [ const False
    , const True ]
  , [ const False `mutate` [(False,True)]
    , const False `mutate` [(True,True)]
    , const True  `mutate` [(False,False)]
    , const True  `mutate` [(True,False)] ] ]
```

This enumeration includes two repeated functions:

```
const True  `mutate` [(False,False)]
const True  `mutate` [(True,False)]
```

which are equivalent to, respectively:

```
const False `mutate` [(True,True)]
const False `mutate` [(False,True)]
```

The repetition is more apparent if we show each function extensionally:

```
tiers :: [Bool -> Bool]  =
  [ [ \x -> case x of  False -> False; True -> False
    , \x -> case x of  False -> True;  True -> True ]
  , [ \x -> case x of  False -> True;  True -> False
    , \x -> case x of  False -> False; True -> True
    , \x -> case x of  False -> False; True -> True
    , \x -> case x of  False -> True;  True -> False ]
  ]
```

We can define a `Show` instance for functional types that shows functions in a similar extensional form by enumerating arguments and recording results similarly to Runciman et al. [2008]. We omit details here as this is not central for understanding how LeanCheck works and somewhat off-topic on this thesis. □

Example 3.2 (revisited) We can now use LeanCheck to get a counterexample to `prop_mapFilter` for boolean element values:

```
> check (prop_mapFilter :: (Bool->Bool) -> (Bool->Bool) -> [Bool] -> Bool)
*** failed for:
\x -> case x of  False -> False;  True -> False
\x -> case x of  False -> True;   True -> False
[True]
```
□

Example 3.4 If we enumerate functions of type `Nat->Nat->Nat`, we see the application of `mutate` at different functional levels:

```
[ [ const (const 0) ]
, [ const (const 1) ]
, [ const (const 0) `mutate` [(0,const 1)]
  , const (const 1) `mutate` [(1,const 0)]
  , const (const 0 `mutate` [(0,1)])
  , const (const 1 `mutate` [(0,0)])
  , const (const 2)
  ]
, ...
]
```
□

The class of enumerated functions is limited to mutations of a constant function. Take for example a simple function like `even :: Int -> Bool`. LeanCheck is not able to enumerate it, only approximations, such as:

```
const False `mutate` [(0,True), (2,True), (4,True)]
```

Within tested properties, enumerated functions are evaluated for a finite number of arguments. So in principle, LeanCheck will be able to find functional counterexamples to any properties. However in practice, this is not true as some functional counterexamples will appear only very late in the enumeration.

Table 3.3 shows the numbers of functions in successive tiers for several types. The number grows by a factor of less than three.

Avoiding repetitions Repetitions mentioned on Example 3.3, can actually be avoided by using the following implementation for `-->>`:

Table 3.3: Numbers of functions in successive tiers for several types

Tier	Number of mutants of type				
	`::Int ->Bool`	`::Bool ->Bool ->Bool`	`::Int ->Int`	`::Int ->Int ->Int`	`::[Int] ->[Int]`
0	2	2	1	1	1
1	2	8	1	1	1
2	2	32	3	5	4
3	4	24	5	13	10
4	4	–	10	35	29
5	6	–	16	81	75
6	8	–	30	201	206
7	10	–	48	460	539
8	12	–	80	1063	1428
9	16	–	129	2374	3721

```
(-->>) :: Eq a => [[a]] -> [[b]] -> [[a -> b]]
xss -->> yss
  | finite xss = mapT ((undefined `mutate`) . zip (concat xss))
                      (products $ replicate (length $ concat xss) yss)
  | otherwise  = concatMapT (\(r,yss) -> mapT (const r `mutate`)
                                              (exceptionPairs xss yss))
                            (choices yss)
```

Whenever the argument type has a finite number of values, we enumerate functions by taking the product of results (sort of like enumerating tuples of results). Otherwise, we use the old enumeration. This avoids repetitions so long as the underlying tiers enumerations do not have repetitions. The `finite` function is an approximation, returning `True` when the tier list contain less than 13 values. We could have chosen to move the definition of `finite` to `Listable` instances putting the burden of deciding finiteness on the user. But we find the current solution to be easier to use.

The chosen solution suffices for practical uses. Consider the following `Nat8` type and its `Listable` instance:

```
newtype Nat8 = Nat8 Int

instance Listable Nat8 where
  list = map Nat8 [0..7]
```

Using the first definition of `-->>`, repetitions only appear after enumerating 10000 values (see Table 3.4). Not using the tuples-of-results enumeration for types with 13 values or more will make no difference from the point of view of having repeated values when enumerating up to 10000 values — an arguably resonable number for property test arguments.

Table 3.4: Ratios of repetitions in different function enumerations when *not* enumerating functions as tuples of results. With a fixed number of tests, as the domain increases, the ratio of repetitions decreases.

	max. # enumerated values				
Type	10	100	1000	10000	100000
`Nat1 -> Nat1`	0%	0%	0%	0%	0%
`Nat2 -> Nat2`	33%	33%	33%	33%	33%
`Nat3 -> Nat3`	0%	53%	53%	53%	53%
`Nat4 -> Nat4`	0%	5%	63%	63%	63%
`Nat5 -> Nat5`	0%	0%	8%	69%	70%
`Nat6 -> Nat6`	0%	0%	<1%	7%	55%
`Nat7 -> Nat7`	0%	0%	0%	<1%	6%
`Nat8 -> Nat8`	0%	0%	0%	0%	<1%

Example 3.3 (revisited) Now, these are the enumerated functions of type `Bool -> Bool`:

```
tiers :: [Bool -> Bool]  =
  [ [ undefined `mutate` [(False,False),(True,False)]
    , undefined `mutate` [(False,False),(True,True)]
    , undefined `mutate` [(False,True),(True,False)]
    , undefined `mutate` [(False,True),(True,True)]
    ]
  ]
```

3.6 Example Applications and Results

Equivalence of folds Consider the following incorrect property stating that type-restricted versions of `foldr` and `foldl` are equivalent:

```
prop_foldlFoldr :: Eq a => (a -> a -> a) -> a -> [a] -> Bool
prop_foldlFoldr f z xs  =  foldr f z xs == foldl f z xs
```

With the property restricted to the `Bool` element type, LeanCheck reports:

```
*** Failed! Falsifiable (after 42 tests):
\x y -> case (x,y) of
          (False,False) -> False
          (False,True) -> False
          (True,False) -> True
          (True,True) -> False
False
[True]
```

With the property restricted to the `Int` element type, LeanCheck reports:

```
*** Failed! Falsifiable (after 75 tests):
\x y -> case (x,y) of
          (0,0) -> 1
          (0,1) -> 1
          (1,0) -> 0
          (0,-1) -> 1
          (1,1) -> 0
          (-1,0) -> 0
          (0,2) -> 1
          (1,-1) -> 0
          ...
0
[0,0]
```

The function shown in the counterexample was defined with

```
const (const 0) `mutate` [(0,const 1)]
```

that is, whenever the first argument is 0, the result is 1. □

Segment Decomposition The following property encodes Bird's (1986) segment decomposition theorem, here restricted to concatenations:

```
prop_segmentDecomposition :: Eq a
                          => ([a] -> Bool) -> ([a] -> [a]) -> [a] -> Bool
prop_segmentDecomposition p f xs  =  s xs == concat (map t (tails xs))
  where
  s xs  =  concat (map f (filter p (segs  xs)))
  t xs  =  concat (map f (filter p (inits xs)))
  segs  :: [a] -> [[a]]
  segs  =  concat . map tails . inits
```

LeanCheck reports that this property is correct:

```
> checkFor 100000 (prop_segmentDecomposition :: ([Int]->Bool) -> ...)
+++ OK, passed 100000 tests.
```

And indeed it is.

If we damage the property by swapping `tails` and `inits` (an easy slip to make?)

```
prop_segmentDecomposition p f xs  =  s xs == concat (map t (inits xs))
  where
  s xs  =  concat (map f (filter p (segs  xs)))
  t xs  =  concat (map f (filter p (inits xs)))
  segs  :: [a] -> [[a]]
  segs  =  concat . map tails . tails
```

LeanCheck instead reports:

```
> checkForr 100000 (prop_segmentDecomposition :: ([Int]->Bool) -> ...)
*** Failed! Falsifiable (after 522 tests):
\x -> case x of
        [] -> True
        [0] -> True
        [0,0] -> True
        [1] -> True
        [0,0,0] -> True
        [0,1] -> True
        [1,0] -> True
        [-1] -> True
        ...
\x -> case x of
        [] -> [1]
        [0] -> [0]
        [0,0] -> [0]
        [1] -> [0]
        [0,0,0] -> [0]
        [0,1] -> [0]
        [1,0] -> [0]
        [-1] -> [0]
        ...
[0]
```

The property fails for the arguments:

- `const True`,
- `const [0] ‘mutate‘ [([],[1])]` and
- `[0]`. □

3.7 Comparison with Related Work

SmallCheck Differently from LeanCheck, SmallCheck enumerates values by depth instead of size. Because of this, the number of test values is harder to control (cf. Table 3.5). SmallCheck also does not perform any kind of diagonalization so values reappear on successive depths and tests are repeated.

Feat Like LeanCheck, Feat [Duregård et al., 2012] enumerates test values by size. For some types, the definition of size differs yielding a different number of values for a given size (cf. Table 3.5).

Table 3.5: Numbers of integer lists of successive sizes (for Feat and LeanCheck) or depths (for SmallCheck).

SmallCheck's depth	1, 2, 7, 36, 253, 2278, 25059, 325768, 4886521, 83070858, ...
Feat's size	0, 1, 0, 0, 2, 2, 4, 12, 24, 52, 120, 264, 584, 1304, 2896, ...
LeanCheck's tiers	1, 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, ...

Feat does not support enumeration of functional test values. As the Feat authors note *"for completeness, Feat should support enumerating functional values. ... This is largely a question of finding a suitable definition of size for functions, or an efficient bijection from an algebraic type into the function type."* Perhaps something similar to the enumeration of functions defined in §3.5 would also be suitable in Feat.

3.8 Conclusion

This chapter described the LeanCheck tool for enumerative property-based testing in Haskell. It generates test values of increasing size enumeratively. This chapter details the three necessary basic components of a property-based testing tool:

- a typeclass for test values with a generator (§3.2);
- a combinator library for defining generators (§3.2);
- a typeclass for testable properties (§3.3).

In addition, it covers conditional properties (§3.4) and enumeration of functions (§3.5).

Limitations LeanCheck is not intended for randomized testing: QuickCheck [Claessen and Hughes, 2000] (§2.1.1) or Feat [Duregård et al., 2012] (§2.1.4) could be used for this purpose. LeanCheck does not provide a sized enumeration combined with test-case pruning using laziness, Neat [Duregård, 2016] could be used for this purpose.

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

Improved pretty printing of functions The current version of LeanCheck pretty-prints functions extensionally by printing enumerated arguments paired with function results. Future versions of LeanCheck could have the option to print functions reflecting how they were generated using applications of `‘mutate‘`.

Functions satisfying conditions Methods to generate functions satisfying conditions could be explored. For example, we may want to test properties that expect their functional arguments to be monotonic, injective, commutative or even associative.

Availability

LeanCheck is freely available with a BSD3-style license from either:

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

The full LeanCheck package includes various facilities not discussed here. This chapter describes LeanCheck as of version 0.7.0

Chapter 4

FitSpec: refining properties for functional testing

This chapter presents FitSpec, a tool providing automated assistance in the task of refining sets of test properties for Haskell functions. FitSpec tests mutant variations of functions under test against a given property set, recording any surviving mutants that pass all tests. The number of surviving mutants and any smallest survivor are presented to the user. A surviving mutant indicates incompleteness of the property set, prompting the user to amend a property or to add a new one, making the property set stronger. Based on the same test results, FitSpec also provides conjectures in the form of equivalences and implications between property subsets. These conjectures help the user to identify minimal core subsets of properties and so to reduce the cost of future property-based testing.

This chapter is based on the paper about FitSpec [Braquehais and Runciman, 2016] presented at the Haskell Symposium 2017.

4.1 Introduction

As discussed in Chapter 2, property-based testing tools automatically test a set of properties describing a set of functions. Two interesting questions arise for any specific application of property-based testing:

- Does the set of properties *completely describe* the set of functions? Is there no other set of functions that passes the tests?
- Is this set of properties *minimal*? Is there a property that is redundant? When doing regression tests, can a property be excluded to speed up the process?

This chapter presents FitSpec, a tool providing automated assistance in the task of refining sets of test properties for Haskell functions. FitSpec does not require sources for functions under test: it only requires a tuple of those functions as component values. Sets of test properties are wrapped to become the result of a function, whose argument is such a tuple of functions (§4.3).

FitSpec enumerates small finite black-box mutations of functions under test (§4.4.1). It tests those mutants against the property set, recording the ones that *survive* by passing all the tests (§4.4.2). It presents the *number of surviving mutants* along with any *smallest surviving mutant* (§4.4.3). A surviving mutant indicates incompleteness of the property set, prompting the user to amend a property or to add a new one. When there is apparent redundancy in a property set, FitSpec provides conjectures in the form of equivalences and implications between properties, helping the user to identify minimal core subsets of properties (§4.4.4).

Example 4.1 Consider the following property set describing a `sort` function:

```
1. \xs ->     ordered (sort xs)
2. \xs ->      length (sort xs) == length xs
3. \x xs ->    elem x (sort xs) == elem x xs
4. \x xs -> notElem x (sort xs) == notElem x xs
5. \x xs ->      minimum (x:xs) == head (sort (x:xs))
```

The first property states that the result of sorting is an ordered list. The last property states that, after sorting, the minimum element of a non-empty list is its head. Other properties state that sorting does not change the quantities or values of list elements. If we supply this property set as input, FitSpec reports that it is neither minimal nor complete:

```
Apparent incomplete and non-minimal specification
20000 tests, 4000 mutants

3 survivors (99% killed), smallest:
  sort' [0,0,1] = [0,1,1]
  sort' xs      = sort xs

minimal property subsets:  {1,2,3} {1,2,4}
conjectures:  {3}    =  {4}     96% killed (weak)
              {1,3} ==> {5}     98% killed (weak)
```

Completeness: FitSpec discovers three mutants that survive testing against all properties. The smallest surviving mutant is clearly not a valid implementation of `sort`, but indeed satisfies all properties. As a specification, the property set is *incomplete* as it omits to require that sorting preserves the number of occurrences of each element value: `\x xs -> count x (sort xs) == count x xs`

Minimality: FitSpec discovers two possible *minimal* subsets of properties: {1,2,3} and {1,2,4}. As measured by the number of killed mutants, each of these subsets is as strong as {1,2,3,4,5}. So far as testing has revealed, Properties 3 and 4 are equivalent and Property 5 follows from 1 and 3. It is *up to the user* to check whether these conjectures are true. Indeed they are, so in future testing we could safely omit Properties 4 and 5.

Refinement: If we omit redundant properties, and add a property to kill the surviving mutant, our refined property set is:

```
1. \xs ->   ordered (sort xs)
2. \xs ->    length (sort xs) == length xs
3. \x xs ->  elem x (sort xs) == elem x xs
4. \x xs -> count x (sort xs) == count x xs
```

FitSpec reports that this property set is apparently complete but not minimal: both 2 and 3 now follow from 4. Since that is true, we might remove Properties 2 and 3 to arrive at a minimal and complete property set. □

Contributions The main contributions of this chapter are:

1. an enumerative black-box mutation-testing technique that does not need function sources or mutation operators, and always returns a smallest or simplest surviving mutant if there is one;
2. a technique to conjecture equivalences and implications between subsets of properties based on mutation testing;
3. a tool (FitSpec) that implements these techniques providing key information for Haskell programmers refining sets of test properties;
4. several small case studies illustrating and evaluating the applicability of FitSpec.

Road-map The rest of this chapter is organized as follows.

- §4.2 defines minimality, completeness, equivalence and implication of property sets;
- §4.3 describes how to use FitSpec;
- §4.4 describes how FitSpec works internally;
- §4.5 presents example applications and results;
- §4.6 discusses related work;
- §4.7 draws conclusions and suggests future work.

4.2 Definitions

We need suitable definitions of completeness, equivalence, implication and minimality of property sets. These are given here, each followed by simple examples.

Definition (complete specification) A set of properties specifying a set of typed and distinctly named functions is *complete* if no other binding of functional values to these names, with the same types, satisfies all properties.

Example 4.2 The following property set describing the standard `Prelude` function `not :: Bool -> Bool` is *incomplete*:

```
1. \p  ->  not (not p) == p
```

For example, the identity function `id :: Bool -> Bool` is distinct from `not` and satisfies the above property.

The following property set, again describing `not`, is *complete*:

```
1. \p -> not (not p) == p
2.       not True == False
```

There is no other `Bool -> Bool` function distinct from the standard `not` function that satisfies the above specification. □

We emphasise that we are viewing functions as black-box correspondences between inputs and outputs. For example, though the alternative declarations

```
not True = False        not p = if p  then False
not False = True                      else True
```

differ, they define the same function.

Definition (equivalence of property sets) Two sets of properties for similarly named and typed functions are *equivalent* if the sets of functional-value bindings satisfying them are the same.

Example 4.3 The property set

```
2. not True == False
```

for the `not` function is *not equivalent* to the property set

```
3. not False == True
```

as, for example, the function `const False :: Bool -> Bool` satisfies Property 2 but not Property 3.

The property set

```
1. \p -> not (not p) == p
2. not True == False
```

is *equivalent* to the property set

```
1. \p -> not (not p) == p
3. not False == True
```

as both are satisfied only when the functional-value binding for `not` is the standard one. □

Definition (implication between property sets) A set of properties *implies* another set, if whenever a functional-value binding satisfies the first set, it also satisfies the second. In other words, the set of functional-value bindings satisfying the first is a subset of the bindings satisfying the second.

Example 4.4 The following property set for the `not` function

```
1. \p -> not (not p) == p
2. not True == False
```

implies the property set

```
3. not False == True
```

as all bindings of a functional value to `not` that satisfy both Properties 1 and 2 also satisfy Property 3. The converse implication does *not* hold: the binding `not = const True` is a counterexample. □

Definition (minimal property sets) A set of properties for a set of typed and distinctly named functions is *minimal* if none of its proper subsets is equivalent to it.

Example 4.5 The following property set for `not` is *not minimal*

```
1. \p -> not (not p) == p
2.       not True == False
3.       not False == True
```

as by inspection Properties 1 and 2 completely specify the standard `not` function. This pair of properties is *minimal*, as neither Property 1 nor Property 2 alone is a complete specification. □

4.3 How FitSpec is Used

FitSpec is used as a library (by "`import Test.FitSpec`"). Unless they already exist, instances of the `Listable` and `Mutable` typeclasses are declared for types of arguments and results of the functions under test (Step 1). Properties are gathered in an appropriately formulated list (Step 2), and passed to the `report` function (Step 3). Property sets are then iteratively refined, based on `report` results (Step 4). The following paragraphs detail this process.

Step 1. Provide typeclass instances for user-defined types The types of arguments and results for functions under test must all be members of the `Listable` (§3.2) and `Mutable` (§4.4.1) type-classes. Where necessary, we declare type-class instances for user-defined types.

FitSpec provides instances for most standard Haskell types and a facility to derive instances for user-defined algebraic data types using Template Haskell [Sheard and Jones, 2002]. For types *without* a constraining data invariant, writing

```
deriveMutable ''<Type>
```

is enough to create the necessary instances. For types *with* a constraining data invariant, the user needs to provide suitable `Listable` instances (§3.2) in addition to the `deriveMutable` call. FitSpec only works for types that are `Eq` instances.

Step 2. Gather properties We must gather properties in a list, to form the body of a *property-map* function with the functions under test as argument. Given a potentially mutated version of a (tuple of) function(s), a property map returns a list of properties over it. The typical form of a property-map declaration is:

```
properties :: (<ty0>,<ty1>,...,<tyN>) -> [Property]
properties (<fun0>,<fun1>,...,<funN>) =
  [ property $ \<args1> -> <property1>
  , property $ \<args2> -> <property2>
  , ...        ...          ...
  , property $ \<argsN> -> <propertyN>
  ]
```

The `property` function encodes a `Testable` property in a format suitable for FitSpec, of the type `Property`:

```
property :: Testable a => a -> Property
```

Essentially, `Testable` values are functions with `Listable` argument types and a `Bool` result. The internal representations of these types and classes are described in §4.4.

Step 3. Call the report function Results are presented by the `report` function. It takes as arguments a tuple of functions under test and a property map, each of monomorphic type. It prints on standard output a report about any surviving mutants and conjectured equivalences or implications. A `report` application can be used as the body of a `main` function to form a compilable program

```
main = report (fun0,fun1,...,funN) properties
```

or alternatively `report` applications can be expressed and evaluated using a REPL interpreter.

By default, FitSpec will try to analyse a given property-set for 5 seconds. A `reportWith` function allows variations of the default settings for controlling values such as: the time limit, the number of test values, and the number of mutant variations.

Step 4. Use results to refine the property-set If a surviving mutant is reported, a typical response from the user is to add a new property, or to strengthen an existing one, so as to “kill” this mutant; then re-test (Step 3).

If there are reported conjectures, a typical response from the user is first to examine these conjectures to see if they are indeed true. (Where this cannot be determined, the relevant subset of properties might be re-tested using larger test-control values.) Where a conjecture is verified, there is an opportunity to remove one or more properties from the set used for testing; then re-test (Step 3).

If no surviving mutant or plausible conjecture is reported, we can stop. Provided that we take care when removing properties, in the end we may hope to obtain a property-set that is stronger than the one we started with, yet simpler. At the least it will be no weaker, and without any redundancies discovered by testing.

When there are no surviving mutants, we may *conjecture* that a property-set is complete. However, because of the inevitable limitations of testing, this conjecture could turn out to be false: there may be a mutant beyond those tested that would have survived. (Here again, one option for the user is re-testing with larger test-control values.)

Example 4.1 (revisited) The following Haskell program analyses the final property-set from the example in the introduction.

```
import Test.FitSpec
import Data.List (sort) -- function under test

properties :: ([Int]->[Int]) -> [Property]
properties sort =
  [ property $ \xs ->   ordered (sort xs)
  , property $ \xs ->    length (sort xs) == length xs
  , property $ \x xs ->  elem x (sort xs) == elem x xs
  , property $ \x xs -> count x (sort xs) == count x xs
  ]

main = reportWith args{names = ["sort xs"]} sort properties
```

Values of the `sort` argument of the `properties` function will be mutated variants of the original and definitive `sort` function passed as argument to `reportWith`. Since FitSpec uses type-guided enumeration, we have to bind `sort` to a specific type in the type signature of `properties`. □

4.4 How FitSpec Works

This section presents details of how FitSpec works. We explore how mutants are enumerated (§4.4.1) and how mutants are tested against properties (§4.4.2) in searches for surviving mutants (§4.4.3), how conjectures are made based on test results (§4.4.4), how we control the extent of testing (§4.4.5), and how we show mutants (§4.4.5).

4.4.1 Enumerating Mutants

Unlike traditional mutation-testing techniques [DeMillo et al., 1978], FitSpec adopts a *black-box* view of functions under test. Mutants have a finite list of exceptional cases in which their results differ from those of the original function. So mutants of a function `f` can be expressed in the following form:

```
\x -> case x of
        <value1> -> <result1>
        <value2> -> <result2>
        ...      -> ...
        <valueN> -> <resultN>
        _        -> f x
```

This section explains how such mutants are enumerated.

Mutants defined in this way may be stricter than the original function. As we test properties only with finite and fully defined arguments, strictness is rarely an issue in practice. However, if the result of a property test is undefined, we catch the exception and treat the test as a failing case.

Mutable typeclass Instances of a `Mutable` typeclass define a `mutiers` function computing tiers (§3.2) of mutants of a given value:

```
class Mutable a where
  mutiers :: a -> [[a]]
```

The first tier contains the equivalent mutant, of size 0, the second tier contains mutants of size 1, the third tier contains mutants of size 2, and so on. The size of a mutant is defined by the instance implementor. As a default, mutant-size can be calculated as the sum of the number of mutated cases and the sizes of arguments and results in these cases.

The *equivalent mutant* is the original function without mutations. As the first tier contains exactly the equivalent mutant, a product of `mutiers` can be computed by `><` (§3.2). Also, `tail mutiers` contains exactly the non-equivalent mutants.

The `mutants` function lists mutants of a given value of some `Mutable` type:

```
mutants :: Mutable a => a -> [a]
mutants = concat . mutiers
```

Enumerating Data Mutants For `Listable` datatypes in the `Eq` class, the following function can be used as the definition of `mutiers`:

```
mutiersEq :: (Listable a, Eq a) => a -> [[a]]
mutiersEq x = [x] : deleteT x tiers
```

The `deleteT` function deletes the first occurrence of a value in a list of tiers. Assuming the underlying `Listable` enumeration has no repeated element, this definition guarantees

that there is no repeated mutant. Having no repeated data mutant will be necessary to avoid equivalent and repeated functional mutants.

Example 4.6 Recalling the natural-number type `Nat` from §3.2, defined as a wrapper over `Int`s:

```
newtype Nat = Nat Int
```

a `Mutable` instance for `Nat` is given by:

```
instance Mutable Nat where
  mutiers = mutiersEq
```

Evaluating `mutiers 3 :: [[Nat]]` yields:

```
[[3], [0], [1], [2], [], [4], [5], [6], [7], ...]
```

The original value has size zero; other mutant values have one added to their sizes; the fifth tier is empty as there is no inequivalent mutant to occupy it. □

Enumerating Functional Mutants Each single-case mutation of a function is defined by an exception pair. Recall the `mutate` function (§3.5) that mutates a function given a list of exception pairs:

```
mutate :: Eq a => (a -> b) -> [(a,b)] -> (a -> b)
mutate f ms  =  foldr mut f ms
  where
  mut (x',fx') f x  |  x == x'    =  fx'
                    |  otherwise  =  f x
```

The `mutationsFor` function returns tiers of exception pairs for a given function in a given single case.

```
mutationsFor :: Mutable b => (a -> b) -> a -> [[(a,b)]]
mutationsFor f x = ((,) x) `mapT` tail (mutiers $ f x)
```

The `mutiersOn` function takes a function and a list of arguments for which results should be mutated; it returns tiers of mutant functions.

```
mutiersOn :: (Eq a, Mutable b) => (a -> b) -> [a] -> [[a -> b]]
mutiersOn f xs = mutate f `mapT` products (map (mutationsFor f) xs)
```

We can now give a `Mutable` instance for functional types:

```
instance (Eq a, Listable a, Mutable b) => Mutable (a -> b) where
  mutiers f = mutiersOn f `concatMapT` setsOf tiers
```

The function `concatMapT` is a variation of `concatMap` for tiers (§3.2). The function `products` takes the product of n lists of tiers producing lists of length n:

```
products :: [ [[a]] ] -> [[ [a] ]]
products = foldr (productWith (:)) [[[]]]
```

Example 4.7 The function `not :: Bool -> Bool` has three inequivalent mutants:

```
\p -> case p of  False -> False;  _     -> not p
\p -> case p of  True  ->  True;  _     -> not p
\p -> case p of  False -> False;  True -> True
```

The first two are of size 1. The last is of size 2. □

Example 4.8 The first four inequivalent mutants for the function `id :: Nat -> Nat` are:

```
\x -> case x of  0 -> 1;  _ -> id x
\x -> case x of  1 -> 0;  _ -> id x
\x -> case x of  0 -> 2;  _ -> id x
\x -> case x of  2 -> 0;  _ -> id x
```

The first two are of size 2, and the last two are of size 3. □

Example 4.9 The first three inequivalent mutants of the natural-number addition function `(+)` are:

```
\x y -> case (x,y) of (0,0) -> 1; _ -> x + y
\x y -> case (x,y) of (0,1) -> 0; _ -> x + y
\x y -> case (x,y) of (1,0) -> 0; _ -> x + y
```

□

Table 4.1 shows, for a few example functions, the number of inequivalent mutants in successive tiers. In the worst case, this number increases by around 3× as size increases by one.

4.4.2 Testing Mutants against Properties

As we saw in §4.3, in order to collect property functions of different types into a single list, we apply FitSpec's `property` function to each of them. The `property` function is polymorphic over the class of `Testable` types (§3.3):

```
property :: Testable a => a -> Property
```

The `Property` type is defined as a synonym:

```
type Property = [([String],Bool)]
```

Here each list of strings is a printable representation of one possible choice of argument values for the property. Each boolean paired with such a list indicates whether the property holds for this choice. The outer list is potentially infinite and lazily evaluated.

A function `propertyHolds`, similar to `holds`, takes as arguments a number of tests and a `Property`; it returns `True` if the property holds in all tested cases, and `False` otherwise.

```
propertyHolds :: Int -> Property -> Bool
propertyHolds n = and . map snd . take n
```

Table 4.1: Numbers of inequivalent mutants in successive tiers for several original functions.

Tier	Number of mutants of:			
	`not :: Bool -> Bool`	`id :: Nat -> Nat`	`(+) :: Nat -> Nat -> Nat`	`sort :: [Nat] -> [Nat]`
1	2	0	0	0
2	1	2	3	2
3	–	2	4	4
4	–	5	12	13
5	–	7	24	32
6	–	13	56	87
7	–	19	113	220
8	–	34	247	581
9	–	49	499	1470
10	–	80	1034	3772

Example 4.1 (revisited) Consider the following `sort` mutant:

```
sort' :: [Nat] -> [Nat]
sort' [0,0,1] = [0,1,1]
sort' xs      = sort xs
```

To test whether `sort'` satisfies the final property-set in Example 4.1 for 1000 test lists, we evaluate

```
propertyHolds 1000 `map` properties sort'
```

obtaining

```
[True, True, True, False]
```

as `sort'` gives ordered results, preserving length and membership, but not preserving element count in the exceptional case. □

4.4.3 Searching for Survivors

Surviving mutants are those for which every test result returned by `propertyHolds` is `True`.

Example 4.1 (revisited) Recall the *incomplete* property set describing `sort` given in §4.1. Testing up to 4000 mutants for 4000 test arguments

```
[m | m <- take 4000 . tail $ mutants sort
   , and $ propertyHolds 4000 `map` properties1 m]
```

three mutants survive:

```
[ \x -> case x of [0,0,1] -> [0,1,1]; _ -> sort x
, \x -> case x of [0,1,0] -> [0,1,1]; _ -> sort x
, \x -> case x of [1,0,0] -> [0,1,1]; _ -> sort x ]
```

If instead we use the *complete* property set, the result of the same test is an empty list. □

In the actual FitSpec implementation, any reported surviving mutant is taken from the list of surviving mutants for the strongest property-set equivalence class — see the next section.

4.4.4 Conjecturing Equivalences and Implications

This section describes how FitSpec conjectures equivalences and implications between subsets of properties.

Properties × Mutants Using `propertyHolds` and `mutants`, we test m mutants against each of p properties using n choices of test arguments. We derive $p \times m$ boolean values each indicating whether a mutant survives testing against a property. These results are computed as a value of type `[(Int,[Bool])]` where each `Int` is a property number, paired with test outcomes for each mutant.

Property sets × Mutants Then, for each mutant, we generate $2^p \times m$ boolean values — the conjunctions of test results for each property subset. These results are computed as a value of type `[([Int],[Bool])]` where each `[Int]` represents a property subset.

Equivalence Classes × Mutants Next, property sets are grouped into equivalence classes. Two sets are put in the same class if they kill the same mutants. Equivalence classes are then sorted by the number of surviving mutants. The results are now of type `[([[Int]],[Bool])]` where each `[[Int]]` represents an equivalence class of property subsets.

Finally, we identify apparent equivalences and implications, according to the following definitions, and report those not subsumed by any other.

Definition (apparent equivalence) Two property sets are *apparently equivalent* (with respect to specified sets of mutant functions and test arguments) if the property sets kill the same mutants. □

Definition (apparent implication) A set of properties *apparently implies* another set (with respect to specified sets of mutant functions and test arguments) if whenever a mutant survives testing against the first set it also survives testing against the second. □

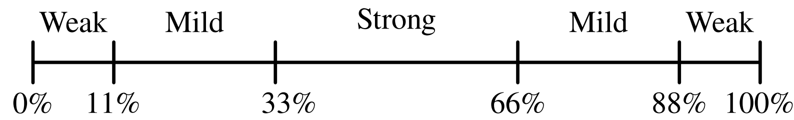


Figure 4.1: Conjecture strengths by percent of surviving mutants.

Strength We have observed that conjectures often do not hold when a supporting survival rate is either 0% or 100%. By interpolation, we *speculate* that equivalences and implications are more likely to hold when survival rates for mutants are closer to 50%, and less likely to hold when survival rates are closer to 0% or 100%. So when FitSpec reports equivalences and implications it sorts them accordingly, reporting first those most likely to hold. Each conjecture is also labelled “strong”, “mild” or “weak” according to the scale in Figure 4.1. This describes how we have implemented the tool, but there is no good reason: again, this is a choice based on *speculation*.

4.4.5 Controlling the Extent of Testing

Choosing the Numbers of Tests and Mutants There is no general rule for choosing appropriate numbers of mutants and test arguments. The most effective values vary between different applications.

By default, FitSpec starts with 500 mutants and 1000 test values per property. As we saw in §4.3, `reportWith` allows the user to choose different values. After each round of testing, both numbers are increased by 50%. Testing continues until a time limit is reached (by default, 5 seconds).

Choosing the Sizes of Types During case studies (§4.5) involving polymorphic functions, we found it helpful to limit generated test values using small instance types. FitSpec predefines types for small signed integers (`IntN`) and unsigned integers (`WordN`), where `N` is a bit-width in the range `1..4`. See §4.5.2 for further discussion.

Showing mutants FitSpec provides two different functions to show mutants: one shows mutants as a tuple of lambdas; the other shows the inequivalent mutants only, as top-level declarations. Both have the following type:

```
ShowMutable a => [String] -> a -> a -> String
```

The `[String]` argument gives names of functions. The other arguments are a tuple of original functions and a tuple of mutated functions. The `ShowMutable` class has a method to show a mutated value given also the original value; instances for user-defined datatypes can be automatically derived. We omit details of the implementation as they are not central to the technique described here.

Example 4.10 One mutant of `id :: Int -> Int` swaps results for argument values 1 and 2:

```
id' :: Int -> Int
id' = id `mutate` [(1,2),(2,1)]
```

Evaluating

```
showMutantAsTuple ["id","not"] (id, not) (id', not)
```

yields (as a string):

```
( \x -> case x of
          0 -> 1
          1 -> 0
          _ -> id x
, not )
```

If we instead use `showMutantDefinition`, we get:

```
id' 0 = 1
id' 1 = 0
id' x = id x
```

□

`ShowMutable` instances for user-defined types can be automatically derived by the function `deriveMutable` (§4.3).

4.5 Example Applications and Results

In this section, we use FitSpec to refine properties of: boolean negation and conjunction operators (§4.5.1); sorting (§4.5.2); merge on min-heaps (§4.5.3); set membership, insertion, deletion, intersection, union (§4.5.4), powersets and partitions (§4.5.5); path and subgraph on digraphs (§4.5.6). These example applications are of increasing complexity.

In §4.5.1 and §4.5.3, we use QuickSpec [Claessen et al., 2010] to generate initial property sets. QuickSpec already incorporates some techniques to refine its output, but we hope for further refinements in the light of FitSpec results. In §4.5.2, our evaluation includes measurements showing the influence of element-type on FitSpec's performance. In most of the examples where functions have polymorphic types, we use instances of the `Word2` type. The non-standard `Word2` type represents two-bit positive integers and is provided by FitSpec. It is analogous to the standard `Word8` and `Word16` types.

4.5.1 Boolean Operators

As a very simple first application, we apply FitSpec to properties generated by QuickSpec [Claessen et al., 2010] for boolean negation and conjunction. We chose this application to double-check FitSpec's functionality as results are easy to verify by hand. Nevertheless, we were not left unsurprised by the results.

Given the functions `not` and `(&&)`, and the value `False`, QuickSpec generates the following set of properties.

```
1. \p ->     not (not p)    == p
2. \p q ->   p && q         == q && p
3. \p ->     p && p         == p
4. \p ->     p && False     == False
5. \p q r -> p && (q && r)  == (p && q) && r
6. \p ->     p && not p     == False
7. \p ->     p && not False == p
```

There are four different minimal subsets of these properties that completely specify the pair of functions `(not,(&&))`. By testing 63 mutant pairs, FitSpec finds and reports this result.

```
Complete but non-minimal specification
22 tests (exhausted), 63 mutants (exhausted)

0 survivors (100% killed)
minimal property subsets:  {1,3,6} {1,4,7}
                           {3,6,7} {4,6,7}
conjectures:  {3}       ==> {5}           76% killed (mild)
              {2,7}     ==> {5}           88% killed (mild)
              {2,4}     ==> {5}           88% killed (mild)
              {1,5,6}   ==> {2}           93% killed (weak)
              {6,7}     ==> {1}           95% killed (weak)
              {2,6,7}    =  {5,6,7}       96% killed (weak)
              {1,2,4,6}  =  {1,4,5,6}     96% killed (weak)
              {3,6}     ==> {4}           96% killed (weak)
              {4,7}     ==> {2,3,5}       98% killed (weak)
```

The absence of commutativity (Property 2) and associativity (Property 5) from all four minimal property subsets might seem surprising, but both are indeed entailed by each of these subsets. The first conjecture `{3} ==> {5}` was even more surprising to one of our colleagues, and to at least one reviewer of our paper about FitSpec [Braquehais and Runciman, 2016], but it is correct — all idempotent binary boolean operators are associative.

4.5.2 Sorting

Now we increase the complexity in relation to the boolean example while still keeping it simple. Consider the following properties of `sort`, which are similar to those given in §4.1. This set of properties is a complete but not minimal specification of `sort`.

Table 4.2: How enlarging the sorted element-type increases the parameters necessary to reach convergence. In practice, `Word2` is sufficient to obtain good results.

Type	**Parameters for converging results**			
	#-mutants	#-tests / prop.	Time	Memory
`Bool`	1000	1000	1s	28MB
`Word1`	2000	2000	3s	39MB
`Word2`	4000	4000	12s	62MB
`Word3`	4000	100000	6m 43s	114MB
`Int`	4000	100000	6m 36s	114MB

```
1. \xs ->    ordered (sort xs)
2. \xs ->     length (sort xs) == length xs
3. \x xs ->   elem x (sort xs) == elem x xs
4. \x xs ->  count x (sort xs) == count x xs
5. \xs ->      permutation xs (sort xs)
6. \x xs -> insert x (sort xs) == sort (x:xs)
```

Effect of element type on performance As `sort` is polymorphic, testing depends on the choice of a specific element type. This choice affects both the results obtained and the resources needed to obtain them.

We say that FitSpec results have *converged* when increasing the number of test-cases used makes no significant difference to the results obtained. After convergence, the reported minimal property-subsets and conjectures stay the same, despite changes in the number of tests, number of mutants and percentages of surviving mutants.

The smaller the type, the lower the values of test-control parameters, and the less run-time, we need to obtain convergence (see Table 4.2). For all the examples we present, `Word2` (or `Int2`) offers a good balance between diversity of values and performance. So, we shall use `Word2` for this and other examples involving polymorphic functions.

In Table 4.2, it might seem surprising that converging parameters for the isomorphic types `Bool` and `Word1` are different. However, their `Listable` instances differ, hence the difference:

```
tiers :: [[ Bool ]] = [ [False,True] ]
tiers :: [[ Word1 ]] = [ [0], [1] ]
```

Still in Table 4.2, the reported memory and time usage are those for the listed parameters.

FitSpec results Given the above properties, FitSpec reports:

```
Apparent complete but non-minimal specification
28000 tests, 4000 mutants

0 survivors (100% killed)
apparent minimal property-subsets: {6} {1,4} {1,5}
conjectures:  {4}  =  {5}       99% killed (weak)
              {4} ==> {2,3}     99% killed (weak)
```

Two of the reported apparent minimal sets, {1,4} and {1,5}, are indeed minimal and complete specifications for a sorting function. The two reported conjectures are also correct.

Property 6 is also reported as an *apparently* complete specification, but consider the function `sort'` defined by

```
sort' :: (Ord a, Bounded a) => [a] -> [a]
sort' = foldr insert [maxBound]
```

or equivalently (for finite and fully-defined arguments):

```
sort' xs = sort xs ++ [maxBound]
```

Substituting `sort'` for `sort` in Property 6, it is easy to see that it holds: unfold both uses of `sort'` and then the right-hand `foldr` application. Yet the results of `sort` and `sort'` differ for *all* finite and fully-defined arguments!

Mutants like `sort'`, which alter the result in an *unbounded* number of cases, are not generated by FitSpec. If a user realises there is a counterexample of this kind, their best option currently is to declare it as a user-defined mutant. If we declare `sort'` as a mutant, Property 6 alone is correctly reported as an incomplete specification. For further discussion see §4.7.

4.5.3 Binary Heaps

We now apply FitSpec to the Heap example provided with the QuickSpec tool package. To limit the extent of this example, we only explore properties of the function `merge`:

```
merge :: Ord a => Heap a -> Heap a -> Heap a
```

If we run QuickSpec with all other functions declared as part of the background algebra, it generates the following properties:

```
1. \h h1 ->    merge h h1 == merge h1 h
2. \h ->       merge h Nil == h
3. \x h h1 ->  merge h (insert x h1) == insert x (merge h h1)
4. \h h1 h2 -> merge h (merge h1 h2) == merge h1 (merge h h2)
5. \h ->       findMin (merge h h) == findMin h
6. \h ->       null (merge h h) == null h
7. \h ->       merge h (deleteMin h) == deleteMin (merge h h)
8. \h h1 ->    (null h && null h1) == null (merge h h1)
```

We soon discover that we should add a pre-condition to Properties 5 and 7, as they only work for non-null heaps.

```
5. \h -> not (null h) ==> findMin (merge h h) == findMin h
7. \h -> not (null h) ==> merge h (deleteMin h) == deleteMin (merge h h)
```

In order to apply FitSpec, we first wrap the properties appropriately, to form a declaration of a FitSpec property-map. We then declare an appropriate `Listable` instance (§3.2) for Heaps:

```
instance (Ord a, Listable a) => Listable (Heap a) where
  tiers = mapT fromList (bagsOf tiers)
```

Running FitSpec, we obtain this report:

```
Apparent complete but non-minimal specification
32000 tests, 2000 mutants

0 survivors (100% killed)
apparent minimal property subsets: {3} {4} {1,2,5,7}
```

Property 4 alone is reported as an apparent minimal (and complete) property subset but it is not. For example, a `merge` function always giving `Nil` as result follows Property 4 but does not follow Properties 2, 5, 6, 7 and 8.

```
conjectures: {2,5}   ==> {6}    64% killed (strong)
             {2,7}   ==> {6}    68% killed (mild)
             {8}     ==> {6}    74% killed (mild)
             {1,2,6} = {1,2,8}  98% killed (weak)
             {1,2,5} ==> {8}    99% killed (weak)
             {1,2,7} ==> {8}    99% killed (weak)
```

It is striking that three conjectures suggest Property 6 is implied by other properties. Indeed, it is easy to see that it follows from Property 8 (let `h1=h`). Property 3 alone cannot specify `merge` despite being reported as a minimal-complete specification. But looking at `insert`'s definition, we can see why FitSpec reports it as such:

```
insert :: Ord a => a -> Heap a -> Heap a
insert x h = merge h (branch x Nil Nil)
```

The function `insert` is defined by `merge` — and since FitSpec treats functions as black-box, it does not mutate the application of `merge` in `insert`'s definition.

Properties 1, 2, 5 and 7 give the best refinement of the initial property set.

4.5.4 Operations over Sets

We next apply FitSpec to a basic repertoire of six functions from a set library: set membership (`<~`), insertion (`insertS`), deletion (`deleteS`), intersection (`/\`), union (`\/`) and set containment (`subS`). For FitSpec runs reported in this section, the time limit was the default 5s, and the declared type of element values was `Word2`.

First, we need a suitable `Listable` instance for sets, for which the underlying representation is ordered lists without repetition.

```
instance (Ord a, Listable a) => Listable (Set a) where
  tiers = mapT set (setsOf tiers)
```

Turning now to properties, our approach for this example is to begin by formulating the first properties that come to mind, ensuring that each function under test occurs in at least one property. We then let FitSpec results guide us in a process of refinement towards a minimal and complete specification. Our initial properties are:

```
1. \x s   -> x <~ insertS x s
2. \x s   -> not (x <~ deleteS x s)
3. \x s t -> (x <~ (s \/ t)) == (x <~ s || x <~ t)
4. \x s t -> (x <~ (s /\ t)) == (x <~ s && x <~ t)
5. \s t   -> subS s (s \/ t)
6. \s t   -> subS (s /\ t) s
7. \s t   -> (s \/ t) == (t \/ s)
8. \s t   -> (s /\ t) == (t /\ s)
```

FitSpec reports that this initial set of properties is neither complete nor minimal:

```
Apparent incomplete and non-minimal specification
3200 tests (exhausted), 750 mutants

49 survivors (93% killed), smallest:
  subS' {0} {} = True
  subS' s   t  = subS s t

apparent minimal property-subsets:  {1,2,3,4,5}
                                    {1,2,3,4,6}
conjectures:  {3} ==> {7}       45% killed (strong)
              {4} ==> {8}       31% killed (mild)
            {3,6} ==> {5}       72% killed (mild)
            {3,4,5} = {3,4,6}   75% killed (mild)
```

Prompted by the surviving mutant, we realise that no property involving `subS` ever demands a `False` result. All the reported implications do indeed hold, so we choose to remove Properties 7 and 8 — from any minimal specification and test set, at least. We also replace properties 5 and 6 by a stronger combined property about `subS` using a minor variant `allS` of the standard `all` function already defined in the `Set` library.

```
1. \x s   -> x <~ insertS x s
2. \x s   -> not (x <~ deleteS x s)
3. \x s t -> (x <~ (s \/ t)) == (x <~ s || x <~ t)
4. \x s t -> (x <~ (s /\ t)) == (x <~ s && x <~ t)
5. \s t   ->        subS s t == allS (<~ t) s
```

FitSpec now reports minimality but incompleteness of the property set, indicating the following surviving mutant:

```
deleteS' 0 {} = {1}
deleteS' x s  = deleteS x s
```

There are no further conjectures for us to think about. But the surviving mutant draws attention to a remaining weakness: Property 2 requires that `deleteS` removes the given element, but not that it retains others. Other surviving mutants point to a similar deficiency for Property 1 about `insert`. We strengthen both properties accordingly:

```
1. \x y s -> x <~ insertS y s == (x == y || x <~ s)
2. \x y s -> x <~ deleteS y s == (x <~ s && x /= y)
3. \x s t ->  (x <~ (s \/ t)) == (x <~ s || x <~ t)
4. \x s t ->  (x <~ (s /\ t)) == (x <~ s && x <~ t)
5. \s t   ->         subS s t == allS (<~ t) s
```

FitSpec reports no conjectures and no surviving mutants:

```
Apparent complete and minimal specification
2816 tests, 750 mutants
```

Indeed, these five properties provide an exact specification, by correspondence with results of Boolean membership test, for these operations on sets.

4.5.5 Powersets and Partitions

Two further functions from the same library each take a set as argument. One computes all subsets (`powerS`) and the other all divisions into pair-wise disjoint non-empty subsets (`partitionsS`).

For properties of these functions, we proceed in a similar way. The basic functions, including those for which properties were developed in the previous section, are now fixed. We work instead with properties of `powerS` and `partitionsS` — and mutant variations of these functions.

Our initial properties are as follows.

```
1. \s t -> (t <~ powerS s) == subS t s
2. \s -> allS (allS (`subS` s)) (partitionsS s)
```

For `powerS` we show we have learnt our lesson from §4.5.4! For `partitionsS` we know Property 2 is not enough, but will FitSpec results point to the deficiencies?

```
Apparent minimal but incomplete specification.
2542 survivors (91% killed), smallest:
  partitionsS' {} = {}
  partitionsS' s  = partitionsS s
```

We add a limited refinement driven directly by the reported mutant.

```
3. \s -> nonEmptyS (partitionsS s)
```

Now FitSpec reports

```
Apparent minimal but incomplete specification.
459 survivors (97% killed), smallest:
  partitionsS' {} = {{{}}}
  partitionsS' s  = partitionsS s
```

so we add:

```
4. \s -> allS (\p -> unionS p == s && allS nonEmptyS p) (partitionsS s)
```

Again we run FitSpec:

```
Apparent incomplete and non-minimal specification
288 tests, 19210 mutants

6 survivors (99% killed), smallest:
  partitionsS' {0,1} = {{{0,1}}}
  partitionsS' s     = partitionsS s

apparent minimal property-subsets:  {1,3,4}
conjectures:  {4} ==> {2}     71% killed (mild)
```

Seeing the conjecture is indeed true, we remove Property 2. Prompted by the unduly restrictive mutant, which excludes the valid partition `{{0},{1}}`, we combine and reformulate Properties 3 and 4 to form a new Property 2:

```
1. \s t -> (t <~ powerS s) == subS t s
2. \s p -> (p <~ partitionsS s)
        == (unionS p == s &&
            allS nonEmptyS p &&
            sum (map sizeS (elemList p)) == sizeS s)
```

FitSpec reports that these two properties apparently form a minimal and complete specification of `powerS` and `partitionsS` — as indeed they do.

Bug report During our work on this example, we actually found a long-concealed bug. As we were refining properties of `partitionsS`, at one stage FitSpec reported:

```
ERROR: The original function-set does not follow property-set.
Counter-example to property 2:  {0,1,2} {{0,1,2}}
Aborting.
```

A data invariant for the set representation requires ordered lists. The definition of the function `partitionsS` was intended to list partitions in a "clever" way to avoid reordering, but in some cases could break the invariant for the outer set. We fixed it. Conclusion: applying a new tool can be insightful!

4.5.6 Operations over Digraphs

Lastly, we apply FitSpec to a directed-graph library based on the datatype

```
data Digraph a = D {nodeSuccs :: [(a,[a])]}
```

where values of some ordered type `a` are node identifiers — or more simply "nodes". Each pair in a strictly ordered `nodeSuccs` list represents a node and an ordered list of its digraph successors.

To limit the extent of this example, we focus on two functions:

```
isPath :: (Ord a, Eq a) => a -> a -> Digraph a -> Bool
subGraph :: Eq a => [a] -> Digraph a -> Digraph a
```

Given two nodes and a digraph, `isPath` tests whether there is a path in the digraph from the first node to the second. Given a list of nodes and a digraph, `subgraph` returns a restricted version of the digraph excluding any nodes not in the list.

We first declare a `Listable` instance for `Digraph`:

```
  tiers = concatMapT graphs $ setsOf tiers
    where
    graphs ns = mapT (D . zip ns) . listsOfLength (length ns) . setsOf
              $ toTiers ns
```

Then we formulate a few properties we expect the two functions to satisfy, including a property involving both of them. Our properties make use of three of the more basic functions from the digraph library: `nodes` lists the nodes in a graph; `isNode` and `isEdge` check whether a given node or edge occur in a graph.

```
1. \n d         -> isPath n n d == isNode n d
2. \n1 n2 n3 d -> isPath n1 n2 d && isPath n2 n3 d ==> isPath n1 n3 d
3. \d           -> subgraph (nodes d) d == d
4. \ns1 ns2 d  -> subgraph ns1 (subgraph ns2 d)
               == subgraph ns2 (subgraph ns1 d)
5. \n1 n2 ns d -> isPath n1 n2 (subgraph ns d) ==> isPath n1 n2 d
```

Strengthening the property set FitSpec reports a surviving `isPath` mutant:

```
isPath’ 0  1  (D [(1,[1])]) = True
isPath’ n1 n2 d             = isPath n1 n2 d
```

Except in the case where they are equal (Property 1), we have not said that starting and finishing nodes of a path at least occur in the digraph! More generally, for distinct nodes, we realise that transitivity (Property 2) only holds `isPath` to account by self-consistency. As a remedy, we add:

```
6. \n1 n2 d -> isPath n1 n2 d ==> isNode n1 d && isNode n2 d
7. \n1 n2 d -> isPath n1 n2 d && n1 /= n2
           ==> any (\n1’ -> isPath n1’ n2 d) (succs n1 d)
```

FitSpec now reports a surviving `subgraph` mutant:

```
subgraph' [1] (D [(0,[]),(1,[])]) = D []
subgraph' ns  d                   = subgraph ns d
```

Aside from the special all-nodes case (Property 3) we have not said what nodes or edges `subgraph` should retain or discard. Again an algebraic law, this time commutativity (Property 4), only requires self-consistency. We add a definitive property about subgraph nodes, and another about subgraph edges:

```
8. \n ns d -> isNode n (subgraph ns d) == (isNode n d && n `elem` ns)
9. \n1 n2 ns d -> isEdge n1 n2 (subgraph ns d)
               == (isEdge n1 n2 d && n1 `elem` ns && n2 `elem` ns)
```

FitSpec reports the following mutant:

```
isPath' 1  0  (D [(0,[]),(1,[0])]) = False
isPath' n1 n2 d                    = isPath n1 n2 d
```

By making Property 7 an implication with an `isPath` test on the left, we allow a false-for-true mutant to survive. Our reformulation involves `subgraph`:

```
7. \n1 n2 d -> n1 /= n2 ==> isPath n1 n2 d
                         == let d' = subgraph (nodes d \\ [n1]) d in
                            any (\n1' -> isPath n1' n2 d') (succs n1 d)
```

At last it seems we have a specification:

```
Apparent complete but non-minimal specification
0 survivors (100% killed)
```

Minimizing the property-set FitSpec's report continues:

```
apparent minimal property subsets:
  {1,4,7,8} {1,7,8,9} {4,5,6,7,8} {5,6,7,8,9}
conjectures:
  {1,7}   ==> {6}         52% killed (strong)
  {6}     ==> {2}         47% killed (strong)
  {7}     ==> {2}         41% killed (strong)
  {4,8}    =  {8,9}       68% killed (mild)
  {4,8}   ==> {3}         68% killed (mild)
  {5}     ==> {2}         80% killed (mild)
  {1,5,7}  =  {5,6,7}     87% killed (mild)
  {1,4,6} ==> {5}         96% killed (weak)
  {1,6,8} ==> {5}         98% killed (weak)
```

In brief, the following property set indeed minimally specifies `subgraph` and `isPath`:

Example	#-mutants	#-tests	time	space
Bool (§4.5.1)	63	8	< 1s	18MB
Sorting (§4.5.2)	4000	4000	12s	62MB
Heaps (§4.5.3)	4000	2000	42s	102MB
Basic Sets (§4.5.4)	750	1024	5s	22MB
Sets of Sets (§4.5.5)	17441	256	5s	58MB
Digraphs (§4.5.6)	750	1500	12s	1853MB

Table 4.3: Summary of Performance Results: figures are mean values across all runs;#-mutants = number of mutants; #-tests = maximum number of test-cases for any property; time = rounded elapsed time and space = peak memory residency (both from GNU `time`).

```
1. \n d -> isPath n n d == isNode n d
7. \n1 n2 d ->  n1 /= n2 ==> isPath n1 n2 d
                          == let d' = subgraph (nodes d \\ [n1]) d in
                             any (\n1' -> isPath n1' n2 d') (succs n1 d)
8. \n ns d -> isNode n (subgraph ns d) == (isNode n d && n `elem` ns)
9. \n1 n2 ns d -> isEdge n1 n2 (subgraph ns d)
               == (isEdge n1 n2 d && n1 `elem` ns && n2 `elem` ns)
```

4.5.7 Performance Summary

Our tool and examples were compiled using `ghc -O2` (version 7.10.3) under Linux. The platform was a PC with a 2.2Ghz 4-core processor and 8GB of RAM. Some performance results are summarized in Table 4.3.

When using FitSpec, ideally users should decide how long they want to wait for FitSpec to run; the simplest parameter to adjust with confidence is the time limit. Reported figures for numbers of mutants and test-cases help the user decide whether to re-run FitSpec allowing more time.

As noted in §4.5.2, for polymorphic functions, the element type affects both the results obtained and resources needed to obtain them. For the examples we present, `Word2` offers a good balance between diversity of values and performance.

4.6 Comparison with Related Work

QuickSpec Claessen et al. [2010] present the QuickSpec tool, which is able to generate algebraic specifications automatically (§2.2.1). Although QuickSpec has rules by which some properties can be discarded as redundant, the goal of its developers was not to generate minimal sets of properties, but instead *interesting* properties.

Bool (§4.5.1) and Heaps (§4.5.3) As we show in §4.5.1 and §4.5.3, FitSpec can assist in the refinement of specifications generated by QuickSpec.

Basic Sets (§4.5.4) For comparison, consider again the basic functions of the set library (§4.5.4), an example where we did *not* start with QuickSpec-generated properties. We can compare our final specification with QuickSpec's output. In the `Set` library example, QuickSpec 1 [Claessen et al., 2010] generates a complete specification with 70 properties. QuickSpec 2 [Smallbone et al., 2017] generates a complete specification with 43 properties, not including any of ours. This striking difference is maybe due to the fact that in our final specification, we appeal to functions outside the set library.

MuCheck Le et al. [2014] present MuCheck, a tool for mutation testing in Haskell. Both MuCheck and FitSpec provide a measure for property-set completeness (§2.2.4). Table 4.4 summarizes the differences between MuCheck and FitSpec. Unlike FitSpec, MuCheck:

- depends on source-code annotations;
- generates mutants by transformations of the source code;
- does not provide conjectures or any form of automated guidance towards minimization;
- may generate mutants equivalent to the original function;
- by default, does not show surviving mutants as they might be equivalent to the original function.

For comparison, we apply MuCheck (version 0.3.0.0, with QuickCheck test adapter version 0.3.0.4) to two of the case studies from §4.5.

Sorting (§4.5.2) Consider the following explicit definition of `sort`, which is used as an example by Le et al. [2014].

```
sort [] = []
sort (x:xs) = sort l ++ [x] ++ sort r
  where
  l = filter (<  x) xs
  r = filter (>= x) xs
```

Given this definition, and Properties 1–6 listed in §4.5.2, MuCheck with default settings gives the following output.

```
Total mutants: 13
  alive: 1/13
  killed: 12/13 (92%)
```

MuCheck does not detect that the only surviving mutant is actually an *equivalent* mutant formed by swapping pattern match cases:

Table 4.4: FitSpec contrasted with MuCheck and Duregård's framework: ●=yes; ○=no

	MuCheck	Duregård's	FitSpec
Random mutant generation	●	●	○
Enumerative mutant generation	○	○	●
Syntactic mutants	●	○	○
Semantic/black-box mutants	○	●	●
Avoids equivalent mutants	○	●	●
Guidance towards completion of property sets	●	●	●
Guidance towards minimization of property sets	○	○	●

```
sort' (x:xs) = sort' l ++ [x] ++ sort' r
  where
  l = filter (< x) xs
  r = filter (>= x) xs
sort' [] = []
```

MuCheck does not consider property subsets. However, if we manually select subsets of properties, results include:

- 1 (equivalent) surviving mutant for Properties 2, 4, 5 and 6 alone;
- 3 surviving mutants for Properties 1 and 3 combined (e.g.: the mutant in which `>=` is changed to `>`);
- 5 surviving mutants for Property 1 (e.g.: changing `>=` to `==`);
- 5 surviving mutants for Property 3 (e.g.: changing `>=` to `/=`).

It takes from 2 to 4 seconds to run MuCheck for each property subset. MuCheck's default settings allow up to 300 mutants, but for this example it only generates 13.

In this example, with regards to evaluating minimality and completeness, FitSpec outperforms MuCheck with default settings. However, MuCheck results might be improved by the definition of custom mutation operators.

Basic Sets (§4.5.4) MuCheck derives no mutants for any of `insertS`, `deleteS`, `subS`, `\/` or `/\` (cf. §4.5.4). The reason may be that there are no MuCheck mutation operators specific to the `Set` type, as we did not add any. For `<~`, MuCheck does derive three mutants, but it then fails because of an internal error. We did not investigate this error, nor did we try applying MuCheck to other functions in the set library.

Ultra-lightweight black-box mutation testing During the Haskell Implementor's Workshop 2014, Jonas Duregård gave a five-minute "lightning talk" about a lightweight technique for mutation testing in Haskell [Duregård, 2014]: ultra-lightweight black-box mutation testing. The technique damages result values randomly. In his thesis [Duregård, 2016], he provides a refined version. Unlike on FitSpec, the focus is only on completeness. Table 4.4 summarizes the differences to FitSpec.

Mutation testing beyond Haskell In a survey of the development of mutation testing, Jia and Harman [2011] specifically identify *equivalent mutants* as one of the barriers to wider adoption of mutation testing. They propose several possible approaches to the problem of equivalent mutants. The approach we have adopted in our work on FitSpec can be characterised in their terms as: (1) "avoiding their initial creation", and (2) "interest in the semantic effects of mutation". The *competent programmer hypothesis* [DeMillo et al., 1978] states: "[Competent programmers] create programs that are *close* to being correct". In mutation-testing literature, mostly concerned with imperative languages [Jia and Harman, 2011, Le et al., 2014], closeness is usually regarded as syntactic closeness. We suggest that a semantic notion of closeness is even more suitable for pure strongly-typed functional programs: minor syntactic slips are very often caught by the type-checker; errors that are harder to detect involve incorrect associations between input and output values.

Mutant subsumption graphs Kurtz et al. [2014] develop a mutant subsumption graph model to describe redundancy among mutations. Subsumption is calculated according to a specific collection of test sets. This is a visualization technique, to support the analysis of the relationships between mutants. In FitSpec, we use the results of testing mutants to build subsumption relations between properties.

Haskell Program Coverage The coverage tool HPC [Gill and Runciman, 2007] records fine-grained expression-level coverage, and value coverage in syntactically boolean contexts. By applying HPC to sources of properties, test-value generators and functions under test, we can check the scope and reach of property-based testing. We can also detect automatically when further exploration of the test-space seems unproductive. However, there are well-known limitations of code-coverage measures: for example, they do not reveal *faults of omission* [Marick, 1999]. HPC does not provide the kind of information needed to discover apparent completeness or minimality of test properties.

4.7 Conclusions and Future Work

Conclusions In summary, we have presented the FitSpec tool to evaluate minimality and completeness of sets of test properties for Haskell functions, providing automated assistance in the task of refining those sets. As set out in §4.3 and §4.4, FitSpec tests mutant variations of the functions under test and reports the number of surviving mutants and, if present, a smallest surviving mutant. When there is apparent redundancy in a property set, FitSpec

reports conjectures in the form of equivalences and implications between property subsets. We have demonstrated in §4.5 FitSpec's applicability for a range of small examples, and we have briefly compared in §4.6 some of the results obtained with related results from other tools.

Completeness and the Value of Surviving Mutants Our experience, as represented by our account of example applications in §4.5, is that details of surviving mutants do point out weaknesses of property sets in a specific and helpful way. Though any mutant-killing refinement of properties depends on the programmer, the smallest-mutant reports are indeed valuable prompts.

Reports of no surviving mutants suggest completeness. However, inherent limitations of a test-based approach make these suggestions uncertain in most cases, and this is one reason for the somewhat repetitive preambles at the head of all FitSpec reports: "Apparent ... specification, N tests, M mutants". We saw in §4.5 examples where property sets are incomplete yet kill all mutants. In some cases uncertainty can be resolved by increasing the numbers of mutants and tests, but in other cases would-be survivors are never generated. As a limited remedy, FitSpec allows the user to provide manually defined mutants to be tested alongside those automatically generated.

Minimality and the Value of Conjectures The conjectured equivalences and implications reported by FitSpec are surprisingly accurate in practice, despite their inherent uncertainty in principle. As we hoped, these conjectures provide helpful pointers to apparently redundant properties. Because conjectures are not guaranteed, before removing any test properties programmers should seek to verify a conjecture that would justify the removal. As we illustrated in §4.5, once we have a conjecture, verifying it often only requires a few straightforward steps appealing to the properties involved — though in general, of course, verification can be a difficult task.

Achieving minimality is useful, but it is comparatively less useful than achieving completeness. Sometimes a minimal property set may not be what the programmer wants as redundant properties may find bugs faster or equivalent properties may find bugs a different speeds. Also, the human reader may find some redundant properties easier to understand. So, it may be useful to keep redundant properties as documentation.

Ease of use Arguably, a tool is easier to use if it requires less work from the programmer. As we illustrated in §4.3, writing a minimal program to apply FitSpec takes only a few lines of code. FitSpec provides functions `mainDefault` and `mainWith`, similar to `report` and `reportWith` but parsing command-line arguments to configure 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. The wrapping of any existing test properties into a property-map declaration is a minor chore.

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 FitSpec may require careful programming of custom `Listable` instances, even if suitable definitions can be very concise. The FitSpec library does not currently incorporate methods to derive enumerators of values satisfying given preconditions [Bulwahn, 2012].

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

Alternative mutation techniques The current mutation technique based on individual exception cases has the advantage of simplicity, but its limitations are most apparent in reports of zero survivors despite incomplete properties. A hybrid approach could generate in addition mutants that alter results for *all* arguments, or a large class of arguments. This is a characteristic of source-based mutants, but there are suitable classes of black-box mutants too. For example, as we saw in §4.5, constant-result mutants may survive properties that kill exception-based mutants; or where there is an argument of the result type, projection-based mutants would be another possibility.

Mutation of higher-order functions Our current mutation technique only works for first-order functions. We might investigate ways to mutate higher-order functions.

Relaxed specifications Our current definition of completeness requires *equality* of results for all functions satisfying a property set. FitSpec regards the results of an original unmutated function as canonical; any other result computed by a mutant function is incorrect. But the natural specification of some functions is more relaxed. For example, sometimes the order of elements in a list is immaterial; in this case, the programmer could resolve the issue by defining a `newtype` for which the equality test disregards order. Not all examples are so simply resolved however: a function to find a shortest path between two nodes in a digraph may return any one of several shortest paths. We might therefore investigate more general ways to characterize equivalence of functional results, with respect to argument values if necessary.

Availability

FitSpec is freely available with a BSD3-style license from either:

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

This chapter describes FitSpec as of version 0.3.1.

Chapter 5

Speculate: discovering conditional equations and inequalities by testing

This chapter 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 these 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.

This chapter is based on the paper about Speculate [Braquehais and Runciman, 2017b] presented at the Haskell Symposium 2017.

5.1 Introduction

This chapter presents a 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 [Claessen et al., 2010, Smallbone et al., 2017]. As Speculate is based on testing, its results are speculative.

Speculate enumerates expressions by combining free variables, functions and values provided by the user (§5.3). 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 ($\Rightarrow$) and inequalities ($\leq$) from representatives of different equivalence classes (§5.4). Speculate uses lightweight equational reasoning based on term rewriting to discard redundant equations and to avoid needless testing. Speculate is implemented in Haskell

and it is designed to find laws about Haskell functions.

Example 5.1 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 unconditional 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 [Claessen et al., 2010, Smallbone et al., 2017]. 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 (Figure 5.1). □

Contributions The main contributions of this chapter are:

1. methods using automated black-box testing and equational reasoning to discover not only apparent unconditional equations but also 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.

Road-map The rest of this chapter is organized as follows:

- §5.2 defines expressions, expression size and a complexity ordering on expressions;
- §5.3 describes how to use Speculate;
- §5.4 describes how Speculate works internally;
- §5.5 presents example applications and results, including laws about functions on directed graphs and axioms of regular expressions;
- §5.6 discusses related work;
- §5.7 draws conclusions and suggests future work.

5.2 Definitions

Now, we define a few terms used in the rest of the chapter.

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,
- `0 + x :: Int` of size 3,
- `x + (y + 0) :: Int` of size 5.
- `x + (id y + x) :: Int` of size 6.

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 [Claessen et al., 2010, Smallbone et al., 2017].

5.3 How Speculate is Used

Speculate is used as a library (by "`import Test.Speculate`"). Unless they already exist, instances of the `Listable` typeclass (§3.2) 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).

Step 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 algebraic data types using Template Haskell [Sheard and Jones, 2002]. For types *without* a constraining data invariant, writing "`deriveListable ''<Type>`" is enough to create the necessary instances (§3.2).

Speculate needs access to reified information about typeclass instances for each user-defined type. This information is provided on the `instances` field in the following form:

```
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 5.1: Full program applying Speculate to `+`, `id` and `abs` used to obtain the results in example 5.1 from §5.1.

```
instances = [ ins "x" (undefined :: <Type1>)
            , ...
            , ins "x" (undefined :: <TypeN>) ]
```

Step 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 5.1 (revisited) Figure 5.1 shows the program used to obtain the results in §5.1. □

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.

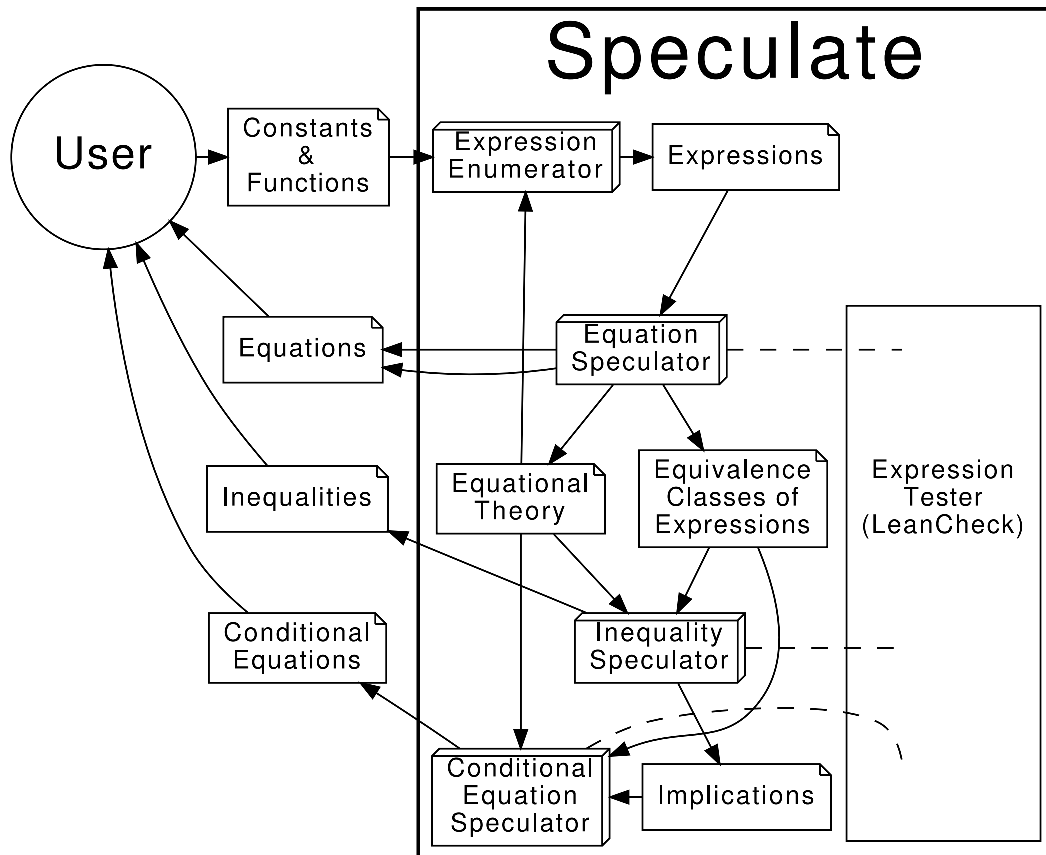


Figure 5.2: Diagram summarizing how Speculate works

5.4 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 (§5.4.1). Speculate determines, in the following order, apparent:

1. equations and equivalence classes of expressions (§5.4.2);
2. inequalities (§5.4.3);
3. conditional equations (§5.4.4).

Figure 5.2 summarizes how Speculate works. Later stages use by-products of earlier stages.

To encapsulate values of different types, Speculate uses the `Data.Dynamic` module [dat, 2017] provided with GHC [GHC Team, 1992–2017] and declares a type to encode Haskell expressions.

5.4.1 Equational Reasoning based on Term Rewriting

Following QuickSpec [Smallbone et al., 2017], Speculate performs basic equational reasoning based on *unfailing Knuth-Bendix Completion* [Bachmair et al., 1989, Knuth and Bendix, 1983]. 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* [Knuth and Bendix, 1983, Baader and Nipkow, 1999]: 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 in a similar way to QuickSpec as detailed in the following paragraphs.

Unorientable equations and bounded equivalence closure To deal with unorientable equations, we use the technique of *unfailing completion* [Bachmair et al., 1989] 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 and size limit 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.

5.4.2 Equations and Equivalence Classes of Expressions

Speculate finds equations in a similar way to QuickSpec 2 [Smallbone et al., 2017]. 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 §5.6.

This section summarizes how Speculate finds equations.

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

- a theory (§5.4.1) 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 5.1: 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 5.2: 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 5.1 (revisited) Table 5.1 shows the equivalence classes after initialization for the example from §5.1 with `0`, `1`, `id`, `abs` and `(+)` in the signature. As yet there are no equations. □

Table 5.3: Equivalence classes and equations after considering all expressions of size 3.

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

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 5.1 (revisited) Using the size 1 representatives in Table 5.1, 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 5.2.

The process of considering expressions is repeated with expressions of further sizes. Table 5.3 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 5.1 (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 involving rotation of variables, 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 (§5.4.3) and conditional equations (§5.4.4). 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.

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

5.4.3 Inequalities between Class Representatives

A naïve approach To find inequalities ($\leq$), a naïve approach enumerates all possible expressions and computes all $\leq$ relations by testing for the configured number of arguments. But it blows up as the size limit increases.

Example 5.1 (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 5.4). 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
```

Table 5.4: How the number of expressions and classes for Example 5.1 increases with the size limit.

	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

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 5.9). The algorithm described in the rest of this section is faster, discovering an equivalent law in about 1s among only 43 other laws. See §5.6 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 (see Table 5.4);
2. we already have all equivalence classes and their smallest representatives as a by-product of finding unconditional equations.

So, based on the equivalence classes of expressions with several variables per type, Speculate finds inequalities in three steps:

1. list all pairs of class representatives;
2. test to find pairs that are related by $\leq$;
3. discard redundant inequalities.

Example 5.1 (revisited) Here are the inequalities found by finding all pairs of expressions related by $\leq$ 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 §5.2. 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 5.1 (revisited) The following 4 inequalities are discarded

```
3.  y <= abs y   (instance of 2. x <= abs x)
5.  0 <= abs y   (instance of 4. 0 <= abs x)
7.  y <= y + 1   (instance of 6. x <= x + 1)
9.  1 <= 1 + 1   (instance of 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 5.1 (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 (§5.4.1) of I_2 is an instance of any of the expressions in the bounded equivalence closure of I_1, Speculate discards I_2.

5.4.4 Conditional Equations between Class Representatives

In this section, we detail how conditional equations are generated based on the equational theory (§5.4.2), class representatives (§5.4.2) and inequalities (§5.4.3) 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 (§5.4.3). We can build a digraph of boolean expressions ordered by implication as shown in Figure 5.3. We include `False` and `True`.

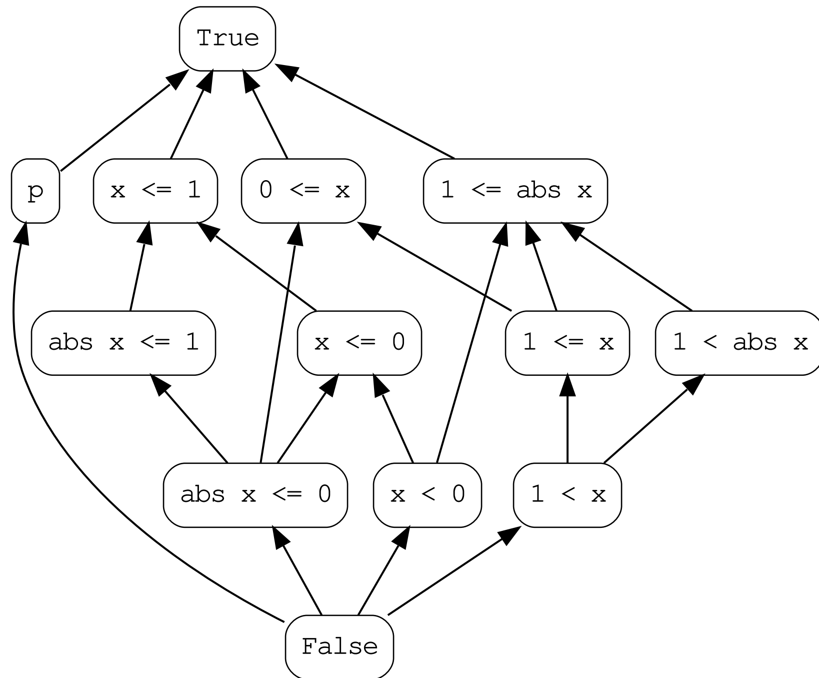


Figure 5.3: Conditions ordered by logical implication for Example 5.1 from §5.1 when considering expressions of at most one distinct variable of each type.

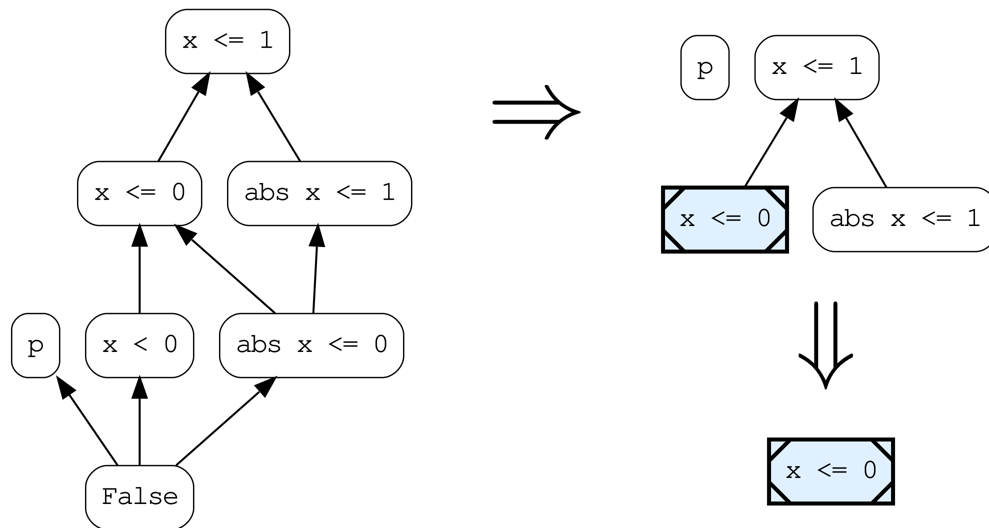


Figure 5.4: Possible transformations performed on the ordering structure from Figure 5.3 when searching for the weakest condition for `x + abs x == 0` to hold. Shaded square nodes are those tested and found to be true.

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 $e_1 = e_2$ 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;
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*: assuming the results of testing are correct, the algorithm's result is correct.

Example 5.1 (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 5.4. 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 5.4. 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.

Discarding redundant conditional equations To discard redundant conditional equations, Speculate uses the two rules described in the following paragraphs.

Discarding consequences of other conditional equations We discard a conditional equation $c_1 \Rightarrow e_3 = e_4$ if we also have a conditional equation $c_0 \Rightarrow e_1 = e_2$ with $c_1 = c_0$ or $c_1 \Rightarrow c_0$ according to the implication digraph; and, e_3 is shown equivalent to e_4 by replacing e_1 with e_2 (or e_2 with e_1).

Discarding consequences of substitution We discard a conditional equation $c \Rightarrow e_1 = e_2$ if: the condition c or any other expression from its equivalence class is of the form $e_3 = e_4$; and, after adding $e_3 = e_4$ to the theory, e_1 is shown equivalent to e_2.

Example 5.2 If the following conditional equation is discovered,

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

it is discarded. The expression `abs x == x` is in the same equivalence class as `0 <= x`. After adding `abs x == x` to the theory, `abs (x + 1)` can be shown equivalent to `x + 1`.

5.5 Example Applications and Results

In this section, we use Speculate:

- to find laws about simple functions on lists (§5.5.1);
- to find a complete implementation of insertion sort (§5.5.2);
- to find ordering properties of binary-tree functions (§5.5.3);
- to find ordering properties of digraph functions (§5.5.4);
- to find an almost complete axiomatisation for regular-expression equivalence (§5.5.5).

Then, in §5.5.6 we give a summary of performance results for all these applications. These example applications are of increasing complexity.

We emphasize what is new compared with QuickSpec [Claessen et al., 2010, Smallbone et al., 2017]. So we often omit details of reported unconditional equations where QuickSpec produces similar results. In §5.6 we shall summarise differences with QuickSpec, including some reasons why the tools may give slightly different sets of unconditional equations.

Sometimes, to avoid being repetitive, we discuss only a selection of inequalities and conditional equations, but always note where others are also generated.

5.5.1 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.

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=[]`, due to laziness.

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 (§5.3)

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

then Speculate uses `isSubsequenceOf` (from `Data.List`) as `<=` for lists of `Ints` 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)
```

5.5.2 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.

5.5.3 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.

5.5.4 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 Speculate reports two conditional equations:

```
elem x xs  ==>  subgraph xs (addNode x a) == addNode x (subgraph xs a)
isEdge x y a  ==>  addEdge x y a == a
```

Indeed, `addNode x` and `subgraph xs` commute when `x` is an element of `xs`. Less excitingly, adding an edge to a graph that has it has no effect.

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 avoid being repetitive.

5.5.5 Regular Expressions

In this section, we use Speculate to conjecture properties about regular expressions (see Table 5.5). It took mathematicians many years to come up with these axiomatizations, it would be quite an achievement if Speculate could generate these automatically. 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

Table 5.5: Regular expression axioms, the size of the largest side (LHS/RHS) and whether each is found by Speculate.

Basic / Common Axioms		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
8. Right distributivity	$(E + F)G \equiv EG + FG$	7	yes
9. Associativity (.)	$E(FG) \equiv (EF)G$	5	yes
Salomaa (1966) Axioms [Salomaa, 1966]			
S10. Left expansion (∗)	$E^* \equiv \epsilon + E^*E$	6	entailed
S11. Inner expansion (∗)	$E^* \equiv (\epsilon + E)^*$	4	yes
S12. Inference (ewp)	$E \equiv EF + G \Rightarrow E \equiv GF^*$ **if** $\neg ewp(F)$	10	no
Conway (1971) Axioms [Conway, 1971]			
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 (∗)	$E^* \equiv (E^n)^*E^{<n} \quad (n > 0)$	—	no
Kozen (1994) Axioms [Kozen, 1994]			
K10. Left expansion (∗)	$\epsilon + EE^* \equiv E^*$	6	yes
K11. Right expansion (∗)	$\epsilon + E^*E \equiv E^*$	6	entailed
K12. Left inequality	$F + EG \leq G \Rightarrow E^*F \leq G$	7	restricted
K13. Right inequality	$F + GE \leq G \Rightarrow FE^* \leq G$	7	restricted

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

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 `RE`s 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 for matching: `Text.Regex.TDFA` from the `regex-tdfa` package.

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

So, for example:

```
> translate id (Lit 'a' :+ Star (Lit 'b' :. Lit 'c'))
"^(a|(bc)*)$"
```

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
```

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. So, we can reduce the number of tests to 400 for a speedup. 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 5.5. Axioms S10 and K11 are not discovered directly, but are entailed by $E^*E \equiv EE^*$ and K10.

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 5.5. At last, Speculate finds all the common laws from all three axiomatisations of regular expressions. It also finds the following restricted cases of Kozen's conditional inequalities (Table 5.5: K12 and K13), crucial ingredients in his complete axiomatisation:

Table 5.6: Summary of performance results: figures are mean values across all runs; size limit = maximum expression size; #-tests = maximum number of test-cases for any property; time = rounded elapsed time and space = peak memory residency;

Example	size limit for eqs./ineqs./c.eqs.			max. vars	#-tests	resources time	space	#-reported eqs./ineqs./c.eqs.		
`+` and `abs`	5	4	4	2	500	3s	7MB	23	17	4
(§5.1)	5	5	5	2	500	25s	7MB	23	44	4
	6	5	5	3	500	2m 37s	8MB	43	44	24
List	5	4	–	3	500	$<$ 1s	7MB	6	6	–
(§5.5.1)	7	6	–	3	500	31s	9MB	7	30	–
Insert Sort	5	–	3	2	500	$<$ 1s	7MB	11	–	2
(§5.5.2)	6	–	5	3	500	5s	8MB	16	–	8
	7	–	6	3	500	1m 27s	12MB	12	–	12
Bin. Trees	5	4	4	2	500	$<$ 1s	7MB	16	4	1
(§5.5.3)	6	5	5	3	500	14s	7MB	16	22	5
Digraphs	5	4	4	2	500	1s	8MB	15	12	2
(§5.5.4)	6	5	5	3	500	1m 52s	10MB	27	30	34
	6	5	5	3	6000	2m 22s	23MB	25	30	17
Regexes	3	–	–	–	500	1m 30s	$<$ 6GB	8	–	–
(§5.5.5)	4	–	–	–	400	9m 11s	$<$ 6GB	12	–	–
	5	–	–	–	200	17m 13s	$<$ 6GB	19	–	–
	6	–	–	–	200	1h 26m 32s	6GB	28	–	–
	6	6	–	2	200	3h 43m 14s	6GB	61	377	–
	6	6	–	3	200	19h 32m 14s	6GB	61	351	–
	7	7	–	2	200	2d 22h 30m 10s	6GB	130	699	–

```
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.

5.5.6 Performance Summary

Performance results are summarized in Table 5.6. 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.

5.6 Comparison with Related Work

QuickSpec The QuickSpec tool [Claessen et al., 2010, Smallbone et al., 2017, Smallbone, 2013, 2011] discovers equational specifications automatically (§2.2.1). Our technique is an extension that allows production of conditional equations and inequalities. QuickSpec inspired us to start working on Speculate. Table 5.7 shows a summary of differences between QuickSpec 1, QuickSpec 2 and Speculate.

Table 5.7: 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 polymorphic functions	no	yes	no
Pruning by external theorem provers	no	yes	no
Performance (see Table 5.8)	slowest	fastest	median

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 [Smallbone et al., 2017]. 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
```

Table 5.8: 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 [Claessen et al., 2010], so, for a fair comparison, we have introduced a depth limit in QS2 and Speculate.

Example	size limit	depth limit	max. #-tests	Runtime in seconds QS1	QS2	Spclt.	#-reported equations QS1	QS2	Spclt.
`(+)` and `abs` (§5.1)	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 (§5.5.1)	7	4	500	52s	< 1s	< 1s	28	7	7
	8	4	500	10m 31s	< 1s	< 1s	40	7	7

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 5.8. QuickSpec 2 is a little bit faster than Speculate — profiling indicates that we were not as smart as the QuickSpec authors when implementing our term rewriting and completion engine.

Table 5.9 presents needed size limits and times to generate some inequalities and conditional laws for QuickSpec 2 and Speculate. Results in Tables 5.8 and 5.9 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.

CoCo The CoCo (Concurrency Commentator) tool [Walker and Runciman, 2017] 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.

Table 5.9: 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 expressions of smaller size.

	Needed size limit		Needed max #-ts.		Runtime		# found laws	
Example & Law	QS2	Spl.	QS2	Spl.	QS2	Spl.	QS2	Spl.
`(+)` and `abs` (§5.1)								
`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 (§5.5.3)								
`isIn x t ==> isIn x (insert y t)`	9	5	2000	500	37s	1s	34	39
Regexes (§5.5.5)								
$F + GE \leq G \Rightarrow E^*F \leq G$	14	7	(open research problem)					

HipSpec QuickSpec and Speculate can only provide *apparent* laws as their results are based on testing. The HipSpec system [Claessen et al., 2012] 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 [Johansson et al., 2014] 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 [Ernst et al., 2007] 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 (Chapter 4) provides automated assistance in the task of refining specifications. It has been applied to QuickSpec results and could also be applied to Speculate results.

5.7 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 §5.3 and §5.4, Speculate enumerates, tests expressions and reasons from test results to produce its conjectures. We have demonstrated in §5.5 Speculate's applicability to a range of small examples, and we have briefly compared in §5.6 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 §5.5.2, 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 often a simple and effective remedy (§5.5.4). The special treatment of inequalities and conditional equations makes possible the generation of laws previously unreachable by a tool such as QuickSpec [Claessen et al., 2010, Smallbone et al., 2017].

Ease of use Arguably, a tool is easier to use if it requires less work from the programmer. As we illustrated in §5.3, 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.

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 (§5.4.2). The same applies to inequalities, e.g., 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.

In addition, Speculate computes all `<=` relations between class representatives. Future versions of Speculate, could explore the transitivity properties of `<=` to avoid some testing.

Parallelism As a way to improve performance, particularly when dealing with costly test functions such as in the regular expressions example (§5.5.5), 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 (§5.3) 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 (§5.5.5).

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 5.1:

```
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 chapter, conjunctions (`&&`) and disjunctions (`||`) can often occur as conditions of properties [Runciman et al., 2008]. 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 §5.5.1, 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 either:

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

This chapter describes Speculate as of version 0.2.9.

Chapter 6

Extrapolate: generalizing counterexamples of test properties

This chapter presents a new tool called Extrapolate that automatically generalizes counterexamples found by property-based testing in Haskell. Example applications show that generalized counterexamples can inform the programmer more fully and more immediately what characterises failures. Extrapolate is able to produce more general results than similar tools. Although it is intrinsically unsound, as reported generalizations are based on testing, it works well in practice.

This chapter is based on the paper about Extrapolate [Braquehais and Runciman, 2017a] presented at IFL 2017.

6.1 Introduction

Most programmers are familiar with the following situation: a failing test case has been discovered during testing; but it is not immediately apparent what more general class of tests would trigger the same failure. The programmer may resort to painstaking step-by-step reevaluation of the reported failure in the hope of realizing where a fault lies. In this chapter, we examine a less explored approach: the *generalization* of failing cases informs the programmer more fully and more immediately what characterises such failures. This information helps the programmer to locate more confidently and more rapidly the causes of failure in their program. We present Extrapolate, a tool to generalize counterexamples of test properties in Haskell. Several example applications demonstrate the effectiveness of Extrapolate.

Example 6.1 Consider once again our running example of the faulty `sort` function from Chapter 1:

```
sort :: Ord a => [a] -> [a]
sort []       =  []
sort (x:xs)   =  sort (filter (< x) xs)
              ++ [x]
              ++ sort (filter (> x) xs)
```

Again, the function `sort` should have the following properties

```
prop_sortOrdered :: Ord a => [a] -> Bool
prop_sortOrdered xs  =  ordered (sort xs)

prop_sortCount :: Ord a => a -> [a] -> Bool
prop_sortCount x xs  =  count x (sort xs) == count x xs
```

that together completely specify `sort`.

If we pass both properties to an established property-testing library, such as QuickCheck [Claessen and Hughes, 2000] or SmallCheck [Runciman et al., 2008], we get something like:

```
> check (prop_sortOrdered :: [Int] -> Bool)
+++ OK, passed 360 tests.

> check (prop_sortCount :: Int -> [Int] -> Bool)
*** Failed! Falsifiable (after 4 tests):
0 [0,0]
```

That is, `prop_sortCount 0 [0,0] = False`. If instead we test using Extrapolate, then for the failing property, in addition to a specific counterexample, Extrapolate goes on to print:

```
Generalization:
x (x:x:_)
```

Some values have been generalized: the specific value `0` does not matter, `prop_sortCount x (x:x:_) = False` for any integer `x`; the tail value `_` does not affect the result.

Extrapolate also prints:

```
Conditional Generalization:
x (x:xs)  when  elem x xs
```

This hints that our faulty `sort` function fails for lists that have repeated elements. We shall return to this example in §6.4.1. □

Contributions The main contributions of this chapter are:

1. methods using automated black-box testing to generalize counterexamples of functional test properties by replacing constructors with variables, where these variables may be repeated or subject to side-conditions;
2. the design of the Extrapolate library, which implements these methods in Haskell and for Haskell test properties;
3. a selection of small case-studies, investigating the effectiveness of Extrapolate;
4. a comparative evaluation of generalizations performed by Extrapolate and similar tools for Haskell.

Despite the Haskell setting of the implementation and experiments, we expect similar techniques to be readily applicable in other functional programming languages.

Road-map This chapter is organized as follows:

- §6.2 describes how to use Extrapolate;
- §6.3 describes how Extrapolate works internally;
- §6.4 presents example applications and results;
- §6.5 discusses related work;
- §6.6 draws conclusions and suggests future work.

6.2 How Extrapolate is Used

Extrapolate is a library: modules using it include "`import Test.Extrapolate`". Unless they already exist, instances of the `Listable` (§6.3.1) and `Generalizable` (§6.3.2) typeclasses are declared for needed user-defined datatypes (step 1). The `check` function is then applied to each test property (step 2).

Step 1. Provide class instances for used-defined types Extrapolate needs to know how to generate test values for property arguments — this capability is provided by instances of the `Listable` typeclass (§3.2). Extrapolate also needs to manipulate the structure of test values so that it can perform its generalization procedure — this capability is provided by instances of the `Generalizable` typeclass (§6.3.2). Extrapolate provides instances for most standard Haskell types and a facility to derive instances for user-defined data types using Template Haskell [Sheard and Jones, 2002]. For datatypes without constraining data invariants, writing

```
deriveGeneralizable ''<Type>
```

is enough to create the necessary instances. Extrapolate's online documentation provides details on how to define these instances manually.

```
import Test.Extrapolate

sort :: Ord a => [a] -> [a]
sort []      =  []
sort (x:xs)  =  sort (filter (< x) xs)
             ++ [x]
             ++ sort (filter (> x) xs)

prop_sortOrdered :: [Int] -> Bool
prop_sortOrdered xs  =  ordered (sort xs)
  where
  ordered (x:y:xs)  =  x <= y && ordered (y:xs)
  ordered _         =  True

prop_sortCount :: Int -> [Int] -> Bool
prop_sortCount x xs  =  count x (sort xs) == count x xs
  where
  count x  =  length . filter (== x)

main :: IO ()
main = do
  check prop_sortOrdered
  check prop_sortCount
```

Figure 6.1: Full program applying Extrapolate to properties of `sort`

Step 2. Call the `check` function The function `check` tests properties, and, if counterexamples are found, it generalizes and reports these counterexamples.

Example 6.1 (revisited) Figure 6.1 shows the program applying Extrapolate to properties of `sort` used to obtain the results in §6.1. □

When exploring conditional generalizations, Extrapolate limits conditions by size. Size is defined as the number of symbols in an expression added to the sizes of constants as defined by LeanCheck's `Listable` enumeration (§6.3.1). This definition is similar to the one used in Speculate (Chapter 5). By default, Extrapolate:

- tests properties for up to 500 value assignments;
- considers side conditions up to size 4;
- uses Speculate to find equivalences between expressions of up to size 4 to avoid testing equivalent conditions (§6.3.3).

Extrapolate allows variations of these default settings. The number of configured tests affects the runtime linearly so long as the underlying `Listable` enumeration has linear runtime as well. The size limit of side conditions affects runtime exponentially so it should be adjusted with caution.

6.3 How Extrapolate Works

Extrapolate works in three steps:

1. it tests properties searching for counterexamples, and if any is found, steps 2 and 3 are performed (§6.3.1);
2. it tries to generalize counterexamples by substituting variables for constants (§6.3.2);
3. it tries to generalize counterexamples by introducing variables subject to side conditions (§6.3.3).

Throughout this section we shall use the following example to illustrate how each step of Extrapolate works.

Example 6.2 In Haskell, the `Data.List` module declares the function `nub`, that removes duplicate elements from a list, keeping only their first occurrences. Consider the following *incorrect* property about `nub`:

```
prop_nubid :: [Int] -> Bool
prop_nubid xs  =  nub xs == xs
```

When Extrapolate's `check` function is applied to the above property, it produces the following output:

```
> check prop_nubid
*** Failed!  Falsifiable (after 3 tests):
[0,0]

Generalization:
x:x:_

Conditional Generalization:
x:xs  when  elem x xs
```

Coincidentally, the result for this example is similar to the result we saw in §6.1 — but it is simpler. □

6.3.1 Searching for counterexamples

To test properties searching for counterexamples, Extrapolate uses LeanCheck (Chapter 3) which defines the `Listable` typeclass used to generate test values *enumeratively*.

Example 6.2 (1st revisit). When Extrapolate's `check` function is applied to `prop_nubid`, it is tested for each of the list arguments: `[]`, `[0]`, `[0,0]`. The property fails for the third list `[0,0]` and Extrapolate produces its first two lines of output:

```
*** Failed!  Falsifiable (after 3 tests):
[0,0]
```

□

6.3.2 Unconditional generalization

Listing generalizations After finding a counterexample, Extrapolate lazily lists all of its possible generalizations, from most general to least general, formed by replacing one or more subexpressions with variables. Generality order is not total as some generalizations are incomparable. So we consider candidate replacements left-to-right.

Example 6.2 (2nd revisit). Recall the counterexample to `prop_nubid`: `[0,0]`, or, more verbosely, `0:0:[]`. Its generalizations from most general to least general are: `xs`, `x:xs`, `x:y:xs`, `x:x:xs`, `x:y:[]`, `x:x:[]`, `x:0:xs`, `x:0:[]`, `0:xs`, `0:x:xs`, `0:x:[]` and `0:0:xs`. □

Testing generalizations For each of these generalizations, Extrapolate tests for a configured number of value assignments whether the property *always* fails. It reports the first generalization for which this test succeeds. Although this process is unsound, as it is based on testing, it works well in practice (§6.4). Any variables that appear only once in generalized counterexamples are reported as "`_`".

Example 6.2 (3rd revisit). The first three generalizations are not counterexamples as there are possible assignments of values for which the property returns `True`:

- `xs` — `prop_nubid []` `= True`
- `x:xs` — `prop_nubid (0:[])` `= True`
- `x:y:xs` — `prop_nubid (0:1:[]) = True`

The fourth generalization `x:x:xs` is tested and the property fails for all tested assignments of values to the variables `x` and `xs`. So, Extrapolate produces its third and fourth lines of output:

```
Generalization:
x:x:_
```

Since the variable `xs` appears only once in the generalized counterexample, it is reported as "`_`". □

6.3.3 Conditional generalization

Background functions Before trying to discover conditional generalizations we must first decide which *background* functions are allowed to appear in conditions.

The larger the number of functions in the background, the longer Extrapolate will take to produce a conditional generalization. So, we refrain from including a large set such as the entire Haskell Prelude. If the algorithm used is ever refined to something faster we may be able to include a larger set by default (§6.6).

Each type has a default list of functions to be considered, declared as part of their `Generalizable` typeclass instance:

- for `Int`s, the functions `(==)`, `(/=)`, `(<=)`, `(<)`;

- for `Char`s, the functions `(==)`, `(/=)`, `(<=)`, `(<)`;
- for `Bool`s, the functions `(==)`, `(/=)`, `not`;
- for lists, the functions `(==)`, `(/=)`, `(<=)`, `(<)`, `length` and `elem`;
- for `Maybe`s, the functions `(==)`, `(/=)`, `(<=)`, `(<)`, `Just`;
- for `Either`s, the functions `(==)`, `(/=)`, `(<=)`, `(<)`, `Left`, `Right`;
- for tuples, the functions `(==)`, `(/=)`, `(<=)`, `(<)`.

For user-defined datatypes, the implementor of instances of the `Generalizable` typeclass decides which functions to include in the background. When using `deriveGeneralizable`, by default, `Eq` instances have `==` and `/=` in the background and `Ord` instances have `<` and `<=` in the background. Additional background functions can be provided using the `‘withBackground‘` combinator. In our experiments (§6.4), we found it useful to include in the background functions appearing in properties.

Whenever a property is tested, these background functions are gathered for all types and component types of arguments of the property being tested. The background of `Bool` is always included.

Background constants Constants of the types being tested, obtained from their `Listable` instances, are also included in the background.

Example 6.2 (4th revisit). The background functions used when testing `prop_nubid :: [Int] -> Bool` are

```
(==), (/=)             :: Bool -> Bool -> Bool
(==), (/=), (<=), (<)  :: Int -> Int -> Bool
(==), (/=), (<=), (<)  :: [Int] -> [Int] -> Bool
not                    :: Bool -> Bool
length                 :: [Int] -> Int
elem                   :: Int -> [Int] -> Bool
```

along with enumerated constants of `Bool`, `Int` and `[Int]` types. □

Enumerating expressions Extrapolate uses Speculate (Chapter 5) to recursively enumerate expressions formed by type-correct applications of background functions to background constants and variables. Speculate avoids generating many distinct but semantically equivalent expressions by using testing and term rewriting. The expressions enumerated are limited by the configured maximum size. This process is done only once for each property that has a counterexample.

Enumerating candidate conditions Expressions which have a boolean type are used as candidate side-conditions.

Example 6.2 (5th revisit). With Extrapolate configured to consider conditions up to size 3, Speculate returns the following 24 conditions

1. `p`
2. `False`
3. `True`
4. `not p`
5. `xs == ys`
6. `xs == []`
7. `xs /= ys`
8. `xs /= []`
9. `xs <= ys`
10. `xs < ys`
11. `elem x xs`
12. `elem 0 xs`
13. `p == q`
14. `p /= q`
15. `x == y`
16. `x == 0`
17. `x /= y`
18. `x /= 0`
19. `x < y`
20. `x < 0`
21. `0 < x`
22. `x <= y`
23. `x <= 0`
24. `0 <= x`

where `p :: Bool`; `x,y :: Int`; `xs,ys :: [Int]`. Single-variable instances are ommited due to the way Speculate works (those are reintroduced later when listing candidate conditional generalizations). □

The use of Speculate is optional and can be turned of by:

```
check `maxSpeculateSize` 0
```

Using Speculate has two effects:

1. runtime is significantly reduced;
2. detection of neutral conditions is improved.

Discarding neutral side-conditions Neutral conditions are those yielding generalizations equivalent to a simpler counterexample. For example:

```
[x,x]  when  x == 0
```

is equivalent to simply

```
[0,0]
```

So any conditions of the form <var> == <value> is discarded. In addition, conditions that are tested and found to be true for only one value for a variable are discarded.

Listing conditional generalizations For each generalization, Extrapolate produces all possible boolean conditions involving its variables based on the list of candidate conditions — this includes all possible variable renamings.

Testing conditional generalizations Extrapolate tests to find side conditions for which the generalization falsifies all tests. Of those, Extrapolate selects the ones with the greatest number of failures (weakest condition).

This is similar to what we have done in Speculate (Chapter 5): there we find side-conditions to properties, whereas here we apply side-conditions to generalized counterexamples.

Example 6.2 (6th revisit). Suppose Extrapolate has been configured to explore conditions of up to size 3. For the first candidate generalization, replacing `[0,0]` by `xs`, two candidate conditions are listed:

1. `xs /= []`
2. `elem 0 xs`

Testing shows that the candidate generalization `xs` does *not* always falsify the property under either of these conditions. For example, the property does not fail for the list `[0]`.

For the second candidate generalization, replacing `0:0:[]` by `x:xs`, Extrapolate lazily lists 8 candidate conditions:

1. `xs /= []`
2. `elem x xs`
3. `elem 0 xs`
4. `x /= 0`
5. `x < 0`
6. `0 < x`
7. `x <= 0`
8. `0 <= x`

The property only fails for all test values of the form `x:xs` under the second condition, `elem x xs`. Extrapolate reports its fifth and sixth lines of output:

```
Conditional Generalization:
x:xs  when  elem x xs
```

□

6.4 Example Applications and Results

In this section, we use Extrapolate to generalize counterexamples of properties about:

- a sorting function (§6.4.1);
- a calculator library (§6.4.2);
- integer overflows (§6.4.3);
- a serializer and parser (§6.4.4);
- the XMonad window manager (§6.4.5).

These example applications are adapted from Pike [2014]. The example applications in §6.4.2, §6.4.4 and §6.4.5 are respectively of relative small, medium and large scale. The example application in §6.4.3 was originally crafted with the intention to to make the use of enumerative testing libraries, like Extrapolate, infeasible — we shall challenge that intention. In §6.4.6, we use generalizations as property refinements. In §6.4.7 we give a summary of performance results for all these applications.

In this section, we evaluate Extrapolate on its own. Comparison with related work can be found in §6.5.

6.4.1 A sorting function: exact generalization

Carrying on from the example described in the introduction (§6.1), if we include `count` as a *background function* (§6.3.3):

```
> let chk = check `withBackground` [ constant "count" count ]
>                 `withConditionSize` 6
> chk (prop_sortCount :: Int -> [Int] -> Bool)
```

Extrapolate prints:

```
Conditional Generalization:
x xs  when  count x xs > 1
```

This is the exact description of the test cases that fail.

It may seem like a stretch to expect users to figure out that they need to include the `count` function in the background. However, during our experiments (cf. §6.4.2 and §6.4.6), we found a simple rule of thumb for improving the results of Extrapolate: to include functions occurring in properties. Automatically including these functions in the background is left as future work (§6.6).

6.4.2 A calculator language

In this section, we use Extrapolate to find and generalize counterexamples to a property about the simple calculator language described by Pike [2014].

Expressions to be calculated are represented by the Haskell datatype `Exp` and may contain integer constants, addition and division.

```
data Exp = C Int
         | Add Exp Exp
         | Div Exp Exp
```

The function `eval` evaluates `Exp`s and returns a `Maybe` value:

- `Nothing` when the calculation involves a division by 0;
- `Just` an integer otherwise.

```
eval :: Exp -> Maybe Int
eval (C i)       = Just i
eval (Add e0 e1) = liftM2 (+) (eval e0) (eval e1)
eval (Div e0 e1) = let e = eval e1
                   in if e == Just 0
                        then Nothing
                        else liftM2 div (eval e0) e
```

The following function `noDiv0`[1], returns `True` when no *literal* division by `0` occurs in an expression.

[1]Pike [2014] originally called this function `divSubTerms`, we found it clearer to call it `noDiv0`.

```
noDiv0 :: Exp -> Bool
noDiv0 (C _)           =  True
noDiv0 (Div _ (C 0))   =  False
noDiv0 (Add e0 e1)     =  noDiv0 e0 && noDiv0 e1
noDiv0 (Div e0 e1)     =  noDiv0 e0 && noDiv0 e1
```

Using `noDiv0`, we define the following test property:

```
\e -> noDiv0 e ==> eval e /= Nothing
```

That is, if an expression contains no literal division by 0, evaluating it returns a `Just` value.

Using Extrapolate, we find a counterexample and two generalizations:

```
> check $ \e -> noDiv0 e ==> eval e /= Nothing
*** Failed! Falsifiable (after 20 tests):
Div (C 0) (Add (C 0) (C 0))

Generalization:
Div (C _) (Add (C 0) (C 0))

Conditional Generalization:
Div e1 (Add (C 0) (C 0))  when  noDiv0 e1
```

The property fails because it is not enough to test whenever any denominator is a literal zero constant, we should test that any denominator *evaluates* to zero. The generalized counterexamples provide improved information for the programmer. Specifically, constructors that are unrelated to the fault are generalized to variables.

The generate the above conditional generalization we manually included `noDiv0` in the list of *background* functions.

The following maximal generalizations (§6.5) are out-of-reach for the current implementation of Extrapolate:

```
Div e1 e2  when noDiv0 (Div e1 e2)
             && eval e2 == Just 0

Div e1 e2  when noDiv0 e1
             && noDiv0 e2
             && e2 /= (C 0)
             && eval e2 == Just 0
```

Their conditions have 9 and 16 symbols respectively. The search space of conditions of those sizes is to big for Extrapolate.

6.4.3 Stress test: integer overflows

As an initial motivating example, Pike [2014] provides the following program and test property:

```
type I  =  [Int16]
data T  =  T I I I I I  deriving  Show

toList :: T -> [[Int16]]
toList (T i j k l m) = [i,j,k,l,m]

pre :: T -> Bool
pre t  =  all ((< 256) . sum) (toList t)

post :: T -> Bool
post t  =  (sum . concat) (toList t) < 5 * 256

prop :: T -> Bool
prop t  =  pre t ==> post t
```

The property is incorrect because it fails to account for overflows caused by lists containing very large negatives. Pike [2014] describes this as an example where enumerative property-based testing tools are infeasible due to the size of input space.

Infeasible by default And indeed, using the standard way to enumerate integers, Extrapolate, an enumerative property-based testing tool, is not able to cope with finding a counterexample to `prop`.

By default Extrapolate enumerates integers by alternating between positives and negatives of increasing magnitude:

```
list :: [Int16] = [0, 1, -1, 2, -2, 3, -3, ...]
```

With this standard definition, Extrapolate does not find a counterexample, even when testing a large number of test values — say a million:

```
> check `for` 1000000 $ prop
+++ OK, passed 1000000 tests.
```

Counterexamples will appear only much later in the enumeration.

Feasible by tweaking enumeration Consider the following alternative definition:

```
list :: [Int16] = [ 0, 1, -1, maxBound,   minBound
                  , 2, -2, maxBound-1, minBound-1
                  , 3, -3, maxBound-2, minBound-2, ... ]
```

It alternates not only between positives and negatives but also between extremely small and extremely large values. The intuition is that, `maxBound` (32767) and `minBound` (-32768) are extreme values that are *likely* to break the property.

With this simple change, Extrapolate does find a counterexample in less than a second:

```
> check `for` 10000 $ prop
*** Failed! Falsifiable (after 8792 tests):
T [] [] [] [-1] [-32768]
```

Extrapolate does not find a generalization.

Despite using enumerative testing, Extrapolate is able to find faults caused by overflows by tweaking the integer enumeration. This technique can be extended to other enumerative property-based testing libraries, like SmallCheck [Runciman et al., 2008], Lazy SmallCheck [Reich et al., 2013], Feat [Duregård et al., 2012] or Neat [Duregård, 2016].

Conditional generalization The function `sum` occurs twice in the property. If we include `sum` as a background function, Extrapolate is able to find the following conditional generalization:

```
> chk = check `for` 10000 `withConditionSize` 4
>             `withBackground`  [constant "sum" (sum :: [Int16] -> Int16)]
> chk prop
Conditional Generalization:
T xs xs xs ((-1):xs) ((-32768):xs)  when  0 == sum xs
```

Unfortunately, Extrapolate takes half an hour to find this conditional generalization. We do not expect users of Extrapolate to wait for that long for a conditional generalization – this specific example is a stress test. In §6.6 we list some avenues of future work to potentially reduce this runtime.

6.4.4 A serializer and parser

In this section, we apply Extrapolate to the parser and pretty printer for a toy language described by Pike [2014]. For brevity, we omit details of the implementation here. It has two main functions:

```
show' :: Prog -> String
read' :: String -> Prog
```

The serializer code (`show'`) has approximately 100 lines of code. The parser code (`read'`) has approximately 200 lines of code. The parser includes a bug that switches the arguments of conjunction expressions.

When we test the property that serializing followed by parsing is an identity, Extrapolate reports a counterexample along with generalizations:

```
> check $ \e -> read' (show' e) == e
*** Failed! Falsifiable (after 96 tests):
Prog [] [Func (Var "a") [And (Int 0) (Bool False)] []]

Generalization:
Prog _ (Func _ (And (Int _) (Bool _):_) _:_)

Conditional Generalization:
Prog _ (Func _ (And e f:_) _:_)  when  e /= f
```

The reported conditional generalization clearly characterizes a set of failing cases: the property fails whenever there is an `And` expression with different operands. This characterization strongly hints at a programming error failing to distinguish operands correctly.

6.4.5 XMonad

XMonad [Stewart and Sjanssen, 2007] is a tiling window manager written in roughly 1700 lines of Haskell code. The XMonad developers defined over a hundred test properties.

In this section, we use Extrapolate to find an artificial bug introduced by Pike [2014] in XMonad.

The function XMonad has a function

```
removeFromWorkspace ws = ws { stack = stack ws >>= filter (/=w) }
```

which removes the current item (or window) from a given workspace.

The bug As described by Pike [2014], we introduce an artificial bug, replacing `/=` by `==` simulating a typo:

```
removeFromWorkspace ws = ws { stack = stack ws >>= filter (==w) }
```

The property The following property `prop_delete` is one of XMonad's original test properties.

```
prop_delete x =
  case peek x of
    Nothing -> True
    Just i  -> not (member i (delete i x))
```

A counterexample In a regular enumerative property-based testing tool, we would get the following minimal counterexample for `prop_delete`:

```
> check prop_delete
Failed! Falsifiable (after 15 tests):
StackSet
  { current = Screen
      { workspace = Workspace
          { tag = 0
          , layout = 0
          , stack = Just (Stack {focus = 'a', up = "", down = ""})
          }
      , screen = 0
      , screenDetail = 0
      }
  , visible = []
  , hidden = []
  , floating = fromList []
  }
```

A generalization When we pass prop_delete to Extrapolate, we instead get the following output:

```
> check prop_delete
*** Failed! Falsifiable (after 15 tests):
StackSet (Screen (Workspace 0 0 (Just (Stack 'a' "" ""))) 0 0)
         [] [] (fromList [])

Generalization:
StackSet (Screen (Workspace _ _ (Just _)) _ _) _ _ _
```

Compare the non-generalized counterexample above with its generalization. We can see quite clearly which parts of are actually related to the fault: what characterizes the failing cases is that an optional third argument (of type Maybe Stack) is present in the argument workspace. Or; as Pike [2014] explained "it turns out what matters is having a Just value, which is the stack field that deletion works on!"

6.4.6 Generalizations as property refinements

Whenever Extrapolate finds a generalised condition C for a property P to fail and we feel that the property is incorrect rather than the code under test, we can directly derive from it a candidate variant of that property: not C ==> P. In this way, Extrapolate is also a tool that assists in the *refinement* of initially conjectured properties, too wide in their scope to be generally true, into more precise variants with scopes defined by conditions.

The actual antecedent introduced in such a refinement may be a simplified equivalent of 'not C'. Or it may be a different condition, prompted and informed by C, but which the programmer conjectures to be (closer to) the exact necessary and sufficient condition for the property to hold. A programmer using testing without any generalising extrapolation has far more need for such conjectures and has to work harder to find them.

The process of extrapolated checking followed by property refinement may be iterative, terminating when testing finds no counterexample at all. There may be intermediate steps as Extrapolate generalisations are often still approximations — either too general or not general enough. Adding or strengthening an antecedent condition allows us to make progress as further testing reveals new residual counterexamples and their generalisations.

Consider the following property about words:

```
prop_lengthWords :: String -> Bool
prop_lengthWords s  =  s /= ""
                   ==> length (words s) == length (filter isSpace s) + 1
```

When passed prop_lengthWords, Extrapolate reports:

```
*** Failed! Falsifiable (after 4 tests):
" "

Generalization:
' ':_
```

```
Conditional Generalization:
c:_  when  c <= ' '
```

We forgot to account for lists that start (or end) with spaces. The conditional generalization is perhaps clearer when we consider the following list comprehension:

```
> [c | c <- list, c <= ' ']
" \n\t"
```

The function `words` also considers `'\n'` and `'\t'` to be spaces. Prompted by this, we add `Data.Char.isSpace` to the background and Extrapolate reports:

```
> check `withBackground` [constant "isSpace" isSpace] $ prop_lengthWords
...
Conditional Generalization:
c:_  when  isSpace c
```

Using this information, we refine our property:

```
prop_lengthWords s  =  s /= "" && not (isSpace (head s))
                               && not (isSpace (last s))
                   ==> length (words s) == length (filter isSpace s) + 1
```

Extrapolate then reports:

```
> check `withBackground` [constant "isSpace" isSpace] $ prop_lengthWords
*** Failed! Falsifiable (after 43 tests):
"a  a"

Conditional Generalization:
c:' ':' ':c:""  when  not (isSpace c)
```

We forgot to account for lists with double spaces. This is made even clearer by adding `Data.List.isInfixOf` and `"  "` to the background and running Extrapolate again on the original `prop_lengthWords`:

```
Conditional Generalization:
cs  when  "  " `isInfixOf` cs
```

We finally refine the property to a correct property of `words`:

```
prop_lengthWords s  =  noDoubleSpace (" " ++ s ++ " ")
                   ==> length (words s) == length (filter isSpace s) + 1
  where
  noDoubleSpace s = and [ not (isSpace a && isSpace b)
                        | (a,b) <- zip s (tail s) ]
```

The number of `words` in a string is given by the number of spaces plus one so long as there are no leading, trailing or double spaces.

Table 6.1: Summary of results for five different applications, testing properties with Extrapolate, SmartCheck and Lazy SmallCheck; #-symbols = #-constants + #-variables; #-constants = number of constants in the reported counterexamples; #-variables = number of variables in the reported counterexamples; generality = how general is the counterexample (○ = least general; ● = most general; × = degenerate); runtime = rounded elapsed time; space = peak memory residency.

Example & Property	Tool	#-symbols	#-constants	#-variables	generality	Runtime	Memory
Faulty `sort` (§6.1)	Ungeneralized	6	6	0	○	< 1s	23MB
`\x xs -> count x (sort xs)`	Lazy SmallCheck	6	6	0	○	< 1s	33MB
`== count x xs`	SmartCheck (min)	6	5	1	◔	< 1s	22MB
	SmartCheck (median)	12	11	1	◔	< 1s	22MB
	Extrapolate	6	2	4	◕	< 1s	26MB
	Extrapolate (side)	8	4	4	●	5s	34MB
Faulty `noDiv0` (§6.4.2)	Ungeneralized	8	8	0	○	< 1s	23MB
`\e -> noDiv0 e`	Lazy SmallCheck	8	7	1	◔	< 1s	33MB
`==> eval e /= Nothing`	SmartCheck	7	6	1	×	< 1s	22MB
	Extrapolate	8	7	1	◔	< 1s	26MB
	Extrapolate (side)	10	8	2	◕	< 1s	28MB
Integer Overflow (§6.4.3)	Ungeneralized	10	10	0	○	< 1s	30MB
`\t -> pre t -> post t`	Lazy SmallCheck	– c.e. not found –				60m 00s	36MB
	SmartCheck (min)	10	5	5	×	< 1s	22MB
	SmartCheck (median)	16	11	5	×	< 1s	22MB
	Extrapolate	10	10	0	○	< 1s	36MB
	Extrapolate (side)	15	9	6	◔	25m 18s	60MB
Faulty parser (§6.4.4)	Ungeneralized	17	17	0	○	< 1s	25MB
`\e -> read' (show' e) == e`	Lazy SmallCheck	17	17	0	○	12s	36MB
	SmartCheck (min)	17	17	0	○	< 1s	23MB
	SmartCheck (median)	27	27	0	○	< 1s	23MB
	Extrapolate	14	7	7	◔	< 1s	27MB
	Extrapolate (side)	16	7	9	◕	9s	35MB
Faulty XMonad (§6.4.5)	Ungeneralized	16	16	0	○	< 1s	28MB
`prop_delete`	SmartCheck	13	9	4	◔	–	–
	Extrapolate	12	4	8	◕	< 1s	30MB
	Extrapolate (side)	15	7	8	◕	< 1s	31MB

6.4.7 Performance Summary

Performance results are summarized in Table 6.1. For all example applications, Extrapolate takes up to a second to produce unconditional generalizations. For the calculator and XMonad examples, Extrapolate also takes up to a second to produce conditional generalizations. For the sorting and parser applications, Extrapolate takes respectively 5 and 9 seconds to consider conditional generalizations. The Table also includes results for other tools, to be discussed in §6.5.

Our tool and examples were compiled using `ghc -O2` (version 8.2.1) under Linux. The platform was a PC with a 2.2Ghz 4-core processor and 8GB of RAM.

6.5 Comparison with Related Work

Tracing and step-by-step evaluation A lot of research has been done on tracing and step-by-step evaluation. In the realm of Haskell, we can note tools such as Freja [Nilsson and Sparud, 1997, Nilsson, 1998], Hat [Wallace et al., 2001, Chitil et al., 2016, 2001] and Hood [Gill, 2001]. These tools facilitate the process of locating faults in programs. Extrapolate on other hand does not directly improve this process, but rather gives the programmer improved results to inform it. Except when a generalized counterexample makes it very obvious where the fault is, Extrapolate does not replace tools like Freja, Hat and Hood: it is more of a complement. Claessen et al. [2003] elaborate on the relation between property-based testing and tracing.

Property discovery QuickSpec [Claessen et al., 2010, Smallbone et al., 2017] and Speculate (Chapter 5) are tools capable of automatically conjecturing properties when given a collection of Haskell functions. Like Extrapolate, these tools rely mainly on testing to achieve their results. Extrapolate does not conjecture properties like these tools, but its generalized counterexamples can be *seen* as properties about faults. In Example 6.1, the counterexample `x (x:x:_)` can be seen as the following property:

```
\x xs -> not $ prop_sortCount x (x:x:xs)
```

Within this view, Extrapolate's generalization aims to find largest test space in which the negation of the property under test succeeds. As stated in §6.3, Extrapolate uses Speculate internally.

Program Synthesis Magic Haskeller [Katayama, 2004, 2005, 2010, 2012] and IGOR II [Kitzelmann, 2007, Hofmann et al., 2010, Hofmann and Kitzelmann, 2010] are systems for program synthesis using inductive functional programming techniques [Kitzelmann, 2010]. They are able to produce programs based on a limited list of input-output bindings and a set of background functions. Similarly, there are PROGOL [Muggleton, 1995], FOIL [Quinlan and Cameron-Jones, 1993] and GOLEM [Cameron-Jones and Quinlan, 1994] for

Table 6.2: Extrapolate contrasted with Lazy SmallCheck and SmartCheck: ●=yes; ○=no.

	SmartCheck	Extrapolate	Lazy SC
Random testing	●	○	○
Enumerative testing	○	●	●
Demand-driven testing	○	○	●
Automatic generator instance derivation	●	●	●
Generalized counterexamples	●	●	●
strict	●	●	○
partial (w/ undefined values)	○	○	●
functional generalizations	○	○	●
repeated variables	○	●	○
side conditions	○	●	○

logic programs. There is a potential for the application of these systems and their techniques to generalize counterexamples: based on which test inputs pass or fail, generate a program to describe a set of counterexamples.

Lazy SmallCheck Lazy SmallCheck [Runciman et al., 2008, Reich et al., 2013] is a property-based testing tool that uses laziness to prune the search space. Before testing fully defined test values, it tries partially defined test values: if a property fails or passes for a partially defined value, more defined variations of that value need not be tested. As a side-effect of this test strategy, Lazy SmallCheck is able to return partially defined values as counterexamples (see Table 6.4). These can be read as generalized counterexamples.

SmartCheck Because QuickCheck tests values randomly, it does not always return small counterexamples, but relies on shrinking [Claessen, 2012] to derive smaller counterexamples from larger ones. SmartCheck [Pike, 2014] is an extension to QuickCheck that provides two improvements: a better algorithm for shrinking and generalization of counterexamples. SmartCheck's generalization algorithm performs both universal and existential quantification but does not allow repeated variables. SmartCheck is perhaps the most closely related tool to Extrapolate, and Pike's paper inspired the work reported here.

Table 6.2 summarizes differences between three tools for Haskell able to report generalized counterexamples: Lazy SmallCheck, SmartCheck and Extrapolate. The key contribution of Extrapolate is allowing for repeated variables and side conditions in generalized counterexamples.

We now revisit examples from §6.4 comparing results of Extrapolate with results of SmartCheck and Lazy SmallCheck.

Table 6.3: Counterexamples for the `count` property of `sort` (Example 6.1 from §6.1) reported by Extrapolate, SmartCheck and Lazy SmallCheck.

Tool	Counterexample
Ungeneralized	`0 [0,0]`
Lazy SmallCheck	`0 [0,0]`
SmartCheck (min)	`4 (4:4:x0)`
SmartCheck	`9 (8:17:9:9:x0)`
Extrapolate	`x (x:x:_)`
Extrapolate (side)	`x xs  when  count x xs > 1`

Criteria The following paragraphs define some of the criteria used in evaluating results of different tools: *generality* and *degenerate counterexamples*.

Generality The more general the counterexample, the better it is. We consider a generalized test-case description more general than another if:

- it strictly subsumes another;
- it includes a larger set of failing test cases.

We say that a generalization is maximal when no more general description exists.

Degenerate counterexamples When a reported generalized counterexample is *too general* and includes inputs that are *not* counterexamples, we say it is *degenerate*. Later in this section, we shall see examples of this.

Representatives and Median values SmartCheck randomly tests values and does not usually report the same counterexamples. In Tables 6.3–6.7 counterexamples for SmartCheck are median representatives only. In Table 6.1, the values for numbers of symbols, constructors and variables are the median of 1000 sample runs.

Summary Table Table 6.1 summarizes all results. For *most* examples, Extrapolate gives *more general* results than either Lazy SmallCheck or SmartCheck. For *all* examples, Extrapolate gives results *at least as general* as Lazy SmallCheck and SmartCheck.

All tools usually report their results within a second, a reasonable time when testing a property. Extrapolate is slower to produce conditional generalizations on the `sort` and `parser` examples, taking respectively 5s and 9s. This increased runtime is still reasonable as generalization is not performed for every property under test but only for those that fail. There are no significant differences in memory use.

Faulty sort (§6.1) See Table 6.3. The counterexample reported by Extrapolate has the same number of symbols as the one reported by Lazy SmallCheck. Lazy SmallCheck is not able to report a generalization as the property being tested is not lazy. Extrapolate is able

Table 6.4: Counterexamples for the property involving `noDiv0` (§6.4.2) reported by Extrapolate, SmartCheck and Lazy SmallCheck. Most generalized counterexamples reported by SmartCheck are degenerate.

Tool	Counterexample
Ungeneralized	`Div (C 0) (Add (C 0) (C 0))`
Lazy SmallCheck	`Div (C _) (Add (C 0) (C 0))`
SmartCheck	`Div x0 (Add (C (-5)) (C 5))`
Extrapolate	`Div (C _) (Add (C 0) (C 0))`
Extrapolate (side)	`Div e1 (Add (C 0) (C 0))  when  noDiv0 e1`

to generalize the tail and the initial elements of the list whereas SmartCheck[2] is only able to generalize the tail. With the function `count` in the background, Extrapolate reports a *maximal* generalization – there is no more general description of the failing cases.

The reported runtime of 5 seconds for this example, only applies if we try to reach the condition involving `count` by increasing the maximum explored condition size to 6. With the default settings, Extrapolate reports the condition involving `elem` in two seconds (§6.1).

Calculator and faulty `noDiv0` (§6.4.2) See Table 6.4. Extrapolate and Lazy SmallCheck report the same generalization. The generalizations SmartCheck reports in 96% of runs are *too* general — as they include values that are *not* counterexamples if we read `==>` as a logical implication. Indeed they are not failing test cases. Concerning the counterexample reported in Table 6.4:

```
> let prop e = noDiv0 e ==> eval e /= Nothing
> let x0 = (Div (C 0) (C 0))
> prop (Div x0 (Div (C (-5)) (C 5)))
True
```

With the precondition falsified, the property holds.

Integer overflow (§6.4.3) See Table 6.5. Lazy SmallCheck is not able to find a counterexample even after running for an hour, replicating the result reported by Pike [2014]. We did not use a customized integer enumeration (§6.4.3). Although Extrapolate fails to report an unconditional generalization, it reports a conditional generalization. SmartCheck reports a degenerate generalization that includes inputs that are not counterexamples. Concerning the proposed counterexample reported in Table 6.5:

```
> let x0 = [21874]; x1 = []; x2 = []; x3 = []
> prop (T x3 (-21874:x0) x2 (-12585:[]) x1)
True
```

[2]Since SmartCheck only tries to generalize the first argument of properties (a design choice), the property had to be uncurried for it to report a generalization.

Table 6.5: Counterexamples for the property about integer overflows (§6.4.3) reported by Extrapolate, SmartCheck and Lazy SmallCheck. The generalized counterexample reported by SmartCheck is degenerate.

Tool	Counterexample
Ungeneralized	`T [] [] [] [-1] [-32768]`
Lazy SmallCheck	— timeout: c.e. not found —
SmartCheck	`T x3 (-21874:x0) x2 (-12585:[]) x1`
Extrapolate	`T [] [] [] [-1] [-32768]`
Extrapolate (side)	`T xs xs xs ((-1):xs) ((-32768):xs)  when  0 == sum xs`

Table 6.6: Counterexamples for the parser property (§6.4.4) reported by Extrapolate, SmartCheck and Lazy SmallCheck.

Tool	Counterexample
Ungeneralized (full record syntax)	`Lang {modules = [], funcs = [Func {fnName = Var "a",` `args = [And (Int 0) (Bool False)], stmts = []}]}`
Ungeneralized	`Lang [] [Func (Var "a") [And (Int 0) (Bool False)] []]`
Lazy SmallCheck	`Lang [] [Func (Var "a") [And (Int 0) (Bool False)] []]`
SmartCheck	`Lang [] [Func (Var "U") []` `[Return (And (Bool False) (Int 0))]]`
Extrapolate	`Lang _ (Func _ (And (Int _) (Bool _):_) _:_)`
Extrapolate (side)	`Lang _ (Func _ (And e f:_) _:_)  when  e /= f`

Faulty parser (§6.4.4) See Table 6.6. Neither Lazy SmallCheck nor SmartCheck is able to report a generalization. Extrapolate is able to report both an unconditional generalization and to improve it in a further conditional generalization. Extrapolate reports counterexamples that are smaller than counterexamples reported by other tools.

Faulty XMonad (§6.4.5) See Table 6.7. SmartCheck[3] and Extrapolate give counterexamples of almost the same number of symbols, but the Extrapolate counterexample has fewer constant constructors and more variables. SmartCheck always assigns variable names whereas Extrapolate uses "`_`" for unrepeated variables, making it more immediately apparent where component values do not matter. Extrapolate's conditional and unconditional generalizations are equivalent.

Versions used We have used the following versions for each tool:

- Lazy SmallCheck: version of 2014-07-07 — compiled with GHC 7.8.4;
- SmartCheck: version 0.2.4 — compiled with GHC 8.0.2;

[3]Due to time constraints we have not tested Lazy SmallCheck or SmartCheck on this example. The SmartCheck counterexample is as reported by Pike [2014] in the original paper about SmartCheck.

Table 6.7: Counterexamples for `prop_delete` from XMonad (§6.4.5) reported by Extrapolate and SmartCheck. The SmartCheck counterexample is the one reported by Pike [2014] in the original SmartCheck paper.

Tool	Counterexample
Ungeneralized	`StackSet (Screen (Workspace 0 0 (Just (Stack 'a' "" ""))) 0 0) [] [] (fromList [])`
SmartCheck	`StackSet (Screen (Workspace x0 (-1) (Just x1)) 1 1) x2 x3 (fromList [])`
Extrapolate	`StackSet (Screen (Workspace _ _ (Just _)) _ _) _ _ _`
Extrapolate (side)	`StackSet (Screen (Workspace _ _ ms) _ _) _ _ _  when  Nothing /= ms`

- Extrapolate: version 0.3.0 — compiled with GHC 8.2.1.

6.6 Conclusions and Future Work

Conclusions In summary, we have presented a tool that is able to generalize counterexamples of functional test properties. As set out in §6.2 and §6.3, Extrapolate enumerates and tests generalizations reporting the one that fails all tests. We have demonstrated in §6.4 Extrapolate's applicability to a range of examples. And after reviewing some related work §6.5, we have compared Extrapolate results with those reported by other tools.

Value of reported laws The conjectured generalizations reported by Extrapolate are surprisingly accurate in practice, despite their inherent uncertainty in principle. They can provide helpful insights into the source of faults. Allowing repeated variables and side-conditions makes possible the discovery of generalizations previously unreachable by other tools that provide similar functionality such as Lazy SmallCheck [Reich et al., 2013] and SmartCheck [Pike, 2014].

Ease of use Extrapolate requires no more programming effort than a regular property-based testing tool such as QuickCheck [Claessen and Hughes, 2000] or SmallCheck [Runciman et al., 2008]. If only standard Haskell datatypes are involved, no extra `Listable` or `Generalizable` instances are needed. If user-defined data types can be freely enumerated without a constraining data invariant, instances can be automatically derived.

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

Type-by-type generalization Although sufficiently fast for the examples we have tried, the current algorithm to find counterexamples is very naïve. It considers generalizations replacing subexpressions of several different types by variables, all in one step. We believe runtime could be reduced by switching to an iterative process where generalization happens

one type at a time. First at outer types, then at inner types. To see why, consider the following example.

Recall Example 6.2. Extrapolate lazily generates 12 generalizations of the list [0,0]: `xs`, `x:xs`, `x:y:xs`, `x:x:xs`, `x:y:[]`, `x:x:[]`, `x:0:xs`, `x:0:[]`, `0:xs`, `0:x:xs`, `0:x:[]` and `0:0:xs`. These generalizations include replacement of sub-expressions by variables of types `Int` and `[Int]`. To reduce the number of generated expressions, we might first consider generalizations with replacements of sub-expressions of `[Int]` type: `xs`, `0:xs` and `0:0:xs` (the first correct generalization). We then consider further generalizations with replacements of sub-expressions of `Int` type: `x:y:xs`, `x:x:xs`, `0:x:xs` and `x:0:xs` with `x:x:xs` being the correct generalization. In this example, the total number of considered generalizations is reduced from 12 to 7.

First generalizing with one variable per type An interesting observation used in Speculate (§5.4.2) could be used to speed-up Extrapolate. For a property with several variables per type to be true, its one-variable-per-type instance should be true as well, for example:

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

The same is true for generalized counterexamples: if lists of `x:y:xs` always falsify a property, then lists of the form `x:x:xs` should as well. Based on this observation, we can test one-variable-per-type generalizations first, potentially reducing the generalization search space.

Generalizing from several counterexamples The current version of Extrapolate bases its generalizations on a single counterexample. As mentioned in §6.5, it may be possible to use techniques from inductive functional programming [Kitzelmann, 2010] to base its generalizations on several counterexamples. This could potentially reduce the time needed to produce generalizations.

Parallelism As a way to improve performance, particularly when dealing with costly test functions, we could parallelize the testing of different enumerated generalizations among multiple processors.

Multiple generalizations Reported generalizations are derived from initial counterexamples. After finding a generalization, Extrapolate could search for other counterexamples and report additional generalizations. These could hopefully be of additional help in finding the source of faults.

Automatically include functions occurring in properties in the background In several examples (§6.4), we provided functions present in the property as background for side conditions improving generalized counterexamples. This could be done automatically to improve out-of-the-box results.

Improved control of configuration parameters Future versions of Extrapolate could offer improved control of configuration parameters. As briefly mentioned in §6.2, the size limit on side conditions is a very fragile parameter: just incrementing it may result in a runtime increase of a few orders of magnitude. To make it easier to configure, future versions of Extrapolate could offer a time limit parameter for when exploring conditions.

Custom generic derivation hierarchy The improvement of counterexamples in favour of more general ones can be compared with the improvement of counterexamples from QuickCheck by shrinking. In QuickCheck, though there are some proposed default generic derivation rules for shrinking, the shrinking methods are also exposed and custom shrinking functions can be declared explicitly. In Extrapolate, the background is a bit like that: there are defaults, and options to declare more. But the method of defining and searching the hierarchy of generalizations is a fixed default. Allowing it to be overridden could be one way to solve the problem of generalizations that break invariant conditions.

Availability

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

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

This chapter describes Extrapolate as of version 0.3.0.

Chapter 7

Conclusions & Future Work

This thesis has presented techniques and tools for discovery, refinement and generalization of properties of functional programs by reasoning from test results. We used the Haskell programming language as a setting, but we expect these techniques to be applicable in other functional programming languages. Chapter 2 reviewed past work in the area of property-based testing. Chapter 3 described the *LeanCheck* tool and a simple technique for enumerative property-based testing. Chapter 4 described the *FitSpec* tool and a technique for refinement of properties using mutation testing with regards to minimality and completeness. Chapter 5 described the *Speculate* tool and a technique for discovery of properties involving conditional equations and inequalities. Chapter 6 described the *Extrapolate* tool and a technique for generalization of counterexamples of test properties.

7.1 Summary of Contributions

In the following paragraphs, we summarize the contributions of each chapter.

Chapter 2, Literature Review Chapter 2 provided an updated literature review on the area of property-based testing.

Chapter 3, LeanCheck Enumerative size-bounded property-based testing already existed in the literature [Runciman et al., 2008, Reich et al., 2013, Duregård et al., 2012].

Chapter 3 contributed a size-bounded enumeration of functions and the framework on which we built other tools presented in this thesis.

Chapter 4, FitSpec Mutation testing is not new [DeMillo et al., 1978, Le et al., 2014, Jia and Harman, 2011, Offutt and Untch, 2001]. Previous work in this area was mostly concerned with completeness of tests and how strongly they restrict the functions under test. It was also based mostly on *syntactic* mutations.

Chapter 4 contributed a technique for enumerative black-box mutation testing avoiding the problem of repeated mutants and a way to use the results of mutation testing to

guide refinement of property sets. These refinements regard minimality and completeness. Towards completion, to kill a mutant, we either add a property or make an existing property stronger. Towards minimality, we remove a property because it is redundant. Moving away from syntactic mutations to semantic or black-box mutations, was one of the keys for the refinement guidance to work as we avoid the problem of repeated mutants.

Chapter 5, Speculate Property discovery techniques were already described in the literature [Claessen et al., 2010, Smallbone, 2011, 2013, Smallbone et al., 2017] but were limited to discovery of equations.

Chapter 5 contributed techniques for *direct* discovery of inequalities and conditional equations. Now, users can get more information with the same amount of programming effort as before. Reported laws can be used for understanding, documentation, and testing.

Chapter 6, Extrapolate Techniques for generalization of counterexamples were already described in the literature [Reich et al., 2013, Pike, 2014].

Chapter 6 contributed a more general technique which allows for generalized counterexamples with repeated variables and side-conditions. Often, Extrapolate's results are more general than those reported by other tools, informing the programmer more fully and more immediately what characterizes failures. When properties are incorrect, Extrapolate's results can be used to guide property refinement.

7.2 Conclusions

The following three paragraphs, recall each of the goals set out in the motivation (§1.1).

Both FitSpec and Speculate can *reduce* the *human effort* needed for property-based testing. FitSpec can be used to guide the creation of properties (Chapter 4). Speculate can discover properties altogether (Chapter 5).

FitSpec can *reduce* the *computational effort* needed for property-based testing. Given a property set, it conjectures which properties may be redundant and how. The user is prompted to consider removing these properties reducing the computational effort of further regression tests (Chapter 4).

Extrapolate can *increase* the *benefit* obtained from property-based testing. It not only reports a failing counterexample but also generalized counterexamples informing the programmer more fully and more immediately what characterizes failures (Chapter 6).

Value of Results Results of lightweight reasoning based on testing are surprisingly accurate in practice, despite their inherent uncertainty in principle. These results often lead to helpful insights into functions under test.

If the above statement seems familiar, it may be because variations of it are found in the conclusions of Chapters 4, 5 and 6, where the "results of lightweight reasoning" are respectively: refinements of properties, discovered laws and generalized counterexamples.

Ease of Use Arguably, tools are easier to use if they require less work from the programmer. As illustrated in Chapters 3 and 6, using LeanCheck or Extrapolate requires no more programming effort than other property-based testing tools such as QuickCheck [Claessen and Hughes, 2000] or SmallCheck [Runciman et al., 2008]. Even in Chapters 4 and 5 where some extra work is required of the programmer, writing a program to apply FitSpec and Speculate takes only a few lines of code. In particular, using Speculate requires no more programming effort than using QuickSpec.

All tools described in this thesis require that data types under test be instances of the `Listable` typeclass. If only standard Haskell data types 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 the tools may sometimes 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 [Bulwahn, 2012, Lindblad, 2007].

Speculate and FitSpec both require the user to select functions to be reasoned upon manually. They do not currently incorporate methods to automatically select functions based on simply the module name, like in the EasySpec interface for QuickSpec [Kerckhove, 2017].

Applicability and Portability To use the tools and techniques described in this thesis, the only restriction placed on properties and functions under test is that they should be *purely functional*, without side effects. In the context of Haskell, this means no `IO` computations. Because these tools are based on black-box testing, that means they work on programs that use language extensions. In contrast, some other techniques, like Reach [Naylor and Runciman, 2007] or MuCheck [Le et al., 2014], perform syntactic evaluation of programs, and require that they conform to a *core language* or a subset of Haskell.

We have avoided undue reliance on GHC compiler extensions, so it is easy to port the tools to other Haskell compilers: we stuck very close to the Haskell 98 and 2010 standards [Jones, 2003, Marlow, 2010]. This also means that the tools are easier to maintain as they mostly rely on established and stable language features.

The tools we developed are not research tools only but work in practice. They are easy to install and run using the current standard Haskell environments. Early users have included computing professionals and students engaged in practical work for taught courses.

7.3 Future Work

As a result of work carried out for this thesis, we hope to have improved the tool ecosystem for property-based testing in Haskell. We can reasonably claim that property-based testing

is a little bit easier and more useful now than it was before we started. Nevertheless, there is still pretty room for improvement, as detailed in the next paragraphs.

Properties about evaluation We could explore properties about evaluation, specifically, about *strictness*, *performance* and *sharing*.

Properties about strictness In this thesis, we have focused on properties about fully defined values. Even in Chapter 6, where we explored generalizations of counterexamples, they represent sets of fully defined counterexamples. We could extend the work of this thesis to include properties about *strictness* and the demands on argument values. For example, consider the following crude property about the strictness of take:

```
prop_takeLazy xs = take (length xs) (xs ++ undefined) == xs
```

Maybe one could devise a Speculate-like technique that conjectures these kind of properties? To achieve this, one could incorporate ideas from the work of Runciman et al. [2008] and Reich et al. [2013] on Lazy SmallCheck and the work of Danielsson and Jansson [2004].

Properties about performance Smallbone and Gedell [2013–2017] implemented a tool that is able to infer the computational complexity of programs by testing. We can think of defining and testing properties about expected performance of programs, such as: the average case complexity of `sort` should be $O(n \log n)$.

Properties about sharing We could explore properties about *sharing* and how things are arranged in memory. This would be useful when implementing data structures that depend on sharing for efficient use of memory and to avoid repeated computations. Take for example, the following imaginary data structure:

```
data DataSuffix = DataSuffix xs ys
```

Suppose there is a data invariant telling `ys` is always a suffix of `xs`. With the current tools, it is easy to define a property that checks this. However, for the sake of efficiency, we may wish to extend the invariant to mean that `ys` is a suffix of `xs` from the memory point of view: when traversing `xs`, at some point, one of the conses of `xs` will point at the same memory location of `ys`. We might investigate how to define and test such properties.

Scalability FitSpec and Speculate do not scale well for some examples (cf. §5.5.5). Improved versions of FitSpec might avoid computing the entire table of properties × mutants. Improved versions of Speculate might avoid computing all possible inequalities between class representatives by exploring the transitivity and anti-symmetry properties of orderings. However, it is unclear if these changes will have a huge impact on performance as Haskell's lazyness already helps.

Parallelism As a way to improve performance, particularly when dealing with costly test functions (§5.5.5), we could parallelise the testing of properties among multiple processors. We imagine this would be just laborious, but will not provide many challenges.

Signature inference for property discovery and refinement Although it is easy to write a program to apply FitSpec or Speculate, it could be a chore in a project applying both tools to several modules.

Kerckhove [2017] describes EasySpec. The user provides only a module name from the command line and EasySpec automatically select functions to describe using QuickSpec. EasySpec or an alternative implementation could be integrated with either FitSpec or Speculate, making it easier to use those tools.

Deriving efficient generators for data types with a data invariant One could incorporate methods to derive efficient generators of values satisfying given preconditions [Bulwahn, 2012, Lindblad, 2007, Runciman et al., 2008, Reich et al., 2013, Duregård, 2016]. Speculate itself may be helpful in doing this. On §5.5.5 we manually use the laws reported by Speculate to discard non-canonical regular expressions from their enumeration. The possibility of automating this process could be investigated.

Program generation We believe there is potential to use Speculate in enumerative program generation [Katayama, 2004, Reich, 2013]. Discovered laws can be used to prune away semantically equivalent programs.

Application in other programming languages We could explore *porting* the tools developed in this thesis to other *functional programming languages*. Even more interestingly, we could also explore porting to other *non-functional programming languages* and the challenges of dealing with side effects.

Availability

LeanCheck, FitSpec, Speculate and Extrapolate are freely available with a BSD3-style licence from:

- `https://hackage.haskell.org/package/leancheck`
- `https://hackage.haskell.org/package/fitspec`
- `https://hackage.haskell.org/package/speculate`
- `https://hackage.haskell.org/package/extrapolate`

or alternatively from:

- `https://github.com/rudymatela/leancheck`
- `https://github.com/rudymatela/fitspec`
- `https://github.com/rudymatela/speculate`
- `https://github.com/rudymatela/extrapolate`

All example applications reported here are present in the tools' source packages making results easily replicable.

Bibliography

Haskell's Data.Dynamic library documentation. `https://hackage.haskell.org/package/base/docs/Data-Dynamic.html`, 2017.

Bernhard K Aichernig. Contract-Based Testing. In *Formal Methods at the Crossroads. From Panacea to Foundational Support*, pages 34–48. Springer, 2003.

Tristan Oliver Richard Allwood. *Finding The Lazy Programmer's Bugs*. PhD thesis, Imperial College London (University of London), 2011.

Thomas Arts, John Hughes, Joakim Johansson, and Ulf Wiger. Testing telecoms software with QuviQ QuickCheck. In *Erlang'06*, pages 2–10. ACM, 2006.

Franz Baader and Tobias Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1999.

Leo Bachmair, Nachum Dershowitz, and David A. Plaisted. Completion without failure. In *Resolution Of Equations In Algebraic Structures*, volume 2, pages 1–30. Academic Press, Boston, 1989.

Adrian Bachmann, Christian Bird, Foyzur Rahman, Premkumar Devanbu, and Abraham Bernstein. The Missing Links: Bugs and Bug-fix Commits. In *FSE'10*, pages 97–106. ACM, 2010.

Stefan Berghofer and Tobias Nipkow. Random testing in Isabelle/HOL. In *SEFM'04*. IEEE, 2004.

Richard S. Bird. *An Introduction to the Theory of Lists*. Technical monograph prg-56, Oxford University Computing Laboratory, October 1986.

Luc Bougé, Nicole Choquet, Laurent Fribourg, and M-C Gaudel. Test sets generation from algebraic specifications using logic programming. *Journal of Systems and Software*, 6 (4):343–360, 1986.

Rudy Braquehais and Colin Runciman. FitSpec: refining property sets for functional testing. In *Haskell'16*, pages 1–12. ACM, 2016.

Rudy Braquehais and Colin Runciman. Extrapolate: generalizing counter-examples of functional test properties. In *IFL 2017 (Draft Proceedings)*. ACM, 2017a.

Rudy Braquehais and Colin Runciman. Speculate: discovering conditional equations and inequalities about black-box functions by reasoning from test results. In *Haskell'17*, pages 40–51. ACM, 2017b.

Rudy Braquehais, Michael Walker, José Manuel Calderón Trilla, and Colin Runciman. A simple incremental development of a property-based testing tool (functional pearl). Unpublished draft, 2017.

Lukas Bulwahn. Smart testing of functional programs in Isabelle. In *LPAR 2012*, LNCS 7180, pages 153–167. Springer, 2012.

Lukas Bulwahn. *Counterexample Generation for Higher-Order Logic Using Functional and Logic Programming*. PhD thesis, 2013.

R. Mike Cameron-Jones and J. Ross Quinlan. Efficient top-down induction of logic programs. *SIGART Bulletin*, 5(1):33–42, Jan 1994.

Olaf Chitil, Colin Runciman, and Malcolm Wallace. *Freja, Hat and Hood — A Comparative Evaluation of Three Systems for Tracing and Debugging Lazy Functional Programs*, pages 176–193. Springer, 2001.

Olaf Chitil, Maarten Faddegon, and Colin Runciman. A lightweight Hat: Simple type-preserving instrumentation for self-tracing lazy functional programs. In *IFL 2016*, pages 1–14. ACM, 2016.

Jan Christiansen and Sebastian Fischer. EasyCheck – Test Data for Free. In *Functional and Logic Programming*, LNCS 4989, pages 322–336. Springer, 2008.

Koen Claessen. Shrinking and Showing Functions. In *Haskell'12*, pages 73–80. ACM, 2012.

Koen Claessen and John Hughes. QuickCheck: A lightweight tool for random testing of Haskell programs. In *ICFP'00*, pages 268–279. ACM, 2000.

Koen Claessen, Colin Runciman, Olaf Chitil, John Hughes, and Malcolm Wallace. Testing and tracing lazy functional programs using QuickCheck and Hat. In *AFP'03*, LNCS 2638, pages 59–99. Springer, 2003.

Koen Claessen, Nicholas Smallbone, and John Hughes. QuickSpec: Guessing Formal Specifications Using Testing. In *TAP 2010*, LNCS 6143, pages 6–21. Springer, 2010.

Koen Claessen, Moa Johansson, Dan Rosén, and Nicholas Smallbone. Hipspec: Automating inductive proofs of program properties. In *Workshop on Automated Theory eXploration: ATX 2012*, 2012.

John Horton Conway. *Regular algebra and finite machines*. Chapman and Hall, 1971.

Nils Anders Danielsson and Patrik Jansson. Chasing bottoms. *Mathematics of Program Construction*, page 85–109, 2004.

R.A. DeMillo, R.J. Lipton, and F.G. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, April 1978.

Jonas Duregård. The testing-feat package, v0.4.0.2, 2014. URL `http://hackage.haskell.org/package/testing-feat`.

Jonas Duregård, Patrik Jansson, and Meng Wang. Feat: functional enumeration of algebraic types. In *Haskell'12*, pages 61–72. ACM, 2012.

Jonas Duregård. *Enumerative Testing and Embedded Languages*. Licentiate thesis, Chalmers University of Technology, 2012.

Jonas Duregård. Ultra-Lightweight Black Box Mutation Testing. `https://youtu.be/ROKxri62WYQ`, 2014.

Jonas Duregård. *Automating Black-Box Property Based Testing*. PhD thesis, Chalmers University of Technology, 2016.

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

George Fink and Matt Bishop. Property-based testing: a new approach to testing for assurance. *ACM SIGSOFT Software Engineering Notes*, 22(4):74–80, 1997.

George Fink and Karl Levitt. Property-based testing of privileged programs. In *Computer Security Applications Conference, 1994. Proceedings., 10th Annual*, pages 154–163. IEEE, 1994.

George Fink, Calvin Ko, Myla Archer, and Karl Levitt. Towards a property-based testing environment with applications to security-critical software. In *4th Irvine Software Symposium*, pages 39–48, 1994.

Gordon Fraser and Andrea Arcuri. Evosuite: Automatic test suite generation for object-oriented software. In *ESEC/FSE '11*, pages 416–419. ACM, 2011.

John Gannon, Paul McMullin, and Richard Hamlet. Data abstraction, implementation, specification, and testing. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 3(3):211–223, 1981.

The GHC Team. The Glasgow Haskell Compiler. `https://www.haskell.org/ghc/`, 1992–2017.

Andy Gill. Debugging haskell by observing intermediate data structures. *Electronic Notes in Theoretical Computer Science*, 41(1):1, 2001. Also in Haskell'00.

Andy Gill and Colin Runciman. Haskell Program Coverage. In *Haskell'07*, pages 1–12. ACM, 2007.

Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: Directed automated random testing. In *PLDI '05*, pages 213–223. ACM, 2005.

Gordon J. Uszkay and Jacques Carette. GenCheck Tutorial. `https://github.com/JacquesCarette/GenCheck/blob/master/tutorial/GenCheck_Tutorial.pdf`, 2012.

Martin Hofmann and Emanuel Kitzelmann. I/O guided detection of list catamorphisms: Towards problem specific use of program templates in IP. In *PEPM'10*, pages 93–100. ACM, 2010.

Martin Hofmann, Emanuel Kitzelmann, and Ute Schmid. Porting Igor II from Maude to Haskell. In *AAIP 2009, Revised Papers*, pages 140–158. Springer, 2010.

John Hughes. *Software Testing with QuickCheck*, pages 183–223. Springer, 2010.

John Hughes, Ulf Norell, Nicholas Smallbone, and Thomas Arts. Find More Bugs with Quickcheck! In *AST '16*, pages 71–77. ACM, 2016.

Graham Hutton. *Programming in Haskell.* Cambridge University Press, 2nd edition, 2016.

Yue Jia and Mark Harman. An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering*, 37(5):649–678, Sept 2011.

Moa Johansson, Dan Rosén, Nicholas Smallbone, and Koen Claessen. Hipster: Integrating theory exploration in a proof assistant. In *CICM 2014*, pages 108–122. Springer, 2014.

Simon Peyton Jones, editor. *Haskell 98 Language and Libraries, The Revised Report.* Cambridge University Press, 2003.

Stefan Kahrs. The Primitive Recursive Functions are Recursively Enumerable, 2006. URL `http://www.cs.kent.ac.uk/people/staff/smk/primrec.pdf`.

Susumu Katayama. Power of brute-force search in strongly-typed inductive functional programming automation. In *PRICAI 2004: Trends in Artificial Intelligence*, LNAI 3157, pages 75–84. Springer, 2004.

Susumu Katayama. Systematic search for lambda expressions. In *TFP 2005*, 2005.

Susumu Katayama. Recent improvements of MagicHaskeller. In *AAIP 2009, Revised Papers*, LNCS 5812, pages 174–193. Springer, 2010.

Susumu Katayama. An analytical inductive functional programming system that avoids unintended programs. In *PEPM'12*, pages 43–52. ACM, 2012.

Tom Sydney Kerckhove. *Signature Inference for Functional Property Discovery*. Masters thesis, ETH Zürich, 2017.

Emanuel Kitzelmann. Data-driven induction of recursive functions from input/output-examples. In *AAIP 2007*, pages 15–26, 2007.

Emanuel Kitzelmann. Inductive programming: A survey of program synthesis techniques. In *AAIP 2009, Revised Papers*, pages 50–73. Springer, 2010.

Donald Knuth and Peter Bendix. Simple word problems in universal algebras. In *Automation of Reasoning*, pages 342–376. Springer, 1983.

Pieter Koopman, Artem Alimarine, Jan Tretmans, and Rinus Plasmeijer. GAST: Generic automated software testing. In *Implementation of Functional Languages*, LNCS 2670, pages 84–100. Springer, 2003.

Dexter Kozen. A completeness theorem for kleene algebras and the algebra of regular events. *Information and Computation*, 110(2):366–390, 1994.

Bob Kurtz, Paul Ammann, Marcio E. Delamaro, Jeff Offutt, and Lin Deng. Mutant subsumption graphs. In *IEEE 7th International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 176–185, 2014.

Duc Le, Mohammad Amin Alipour, Rahul Gopinath, and Alex Groce. MuCheck. `https://bitbucket.org/osu-testing/mucheck.git`.

Duc Le, Mohammad Amin Alipour, Rahul Gopinath, and Alex Groce. Mutation testing of functional programming languages. Technical report, Oregon State University, School of Software Engineering and Computer Science, 2013.

Duc Le, Mohammad Amin Alipour, Rahul Gopinath, and Alex Groce. MuCheck: An extensible tool for mutation testing of Haskell programs. In *ISSTA 2014*, pages 429–432. ACM, 2014.

Fredrik Lindblad. Property directed generation of first-order test data. In *TFP'07*, pages 105–123, 2007.

Brian Marick. How to misuse code coverage. In *International Conference on Testing Computer Software*, pages 16–18, 1999.

Simon Marlow, editor. *Haskell 98 Language and Libraries, The Revised Report*. 2010.

Stephen Muggleton. Inverse entailment and progol. *New Generation Computing*, 13(3): 245–286, Dec 1995.

Glenford J Myers, Corey Sandler, and Tom Badgett. *The art of software testing.* John Wiley & Sons, 3rd edition, 2011.

Matthew Naylor and Colin Runciman. Finding Inputs that Reach a Target Expression. In *Source Code Analysis and Manipulation, 2007*, pages 133–142. IEEE, 2007.

Max S. New, , Burke Fetscher, Robert Bruce Findler, and Jay McCarthy. Fair enumeration combinators. *Journal of Functional Programming*, 27, 2017.

Henrik Nilsson. *Declarative Debugging for Lazy Functional Languages*. PhD thesis, 1998.

Henrik Nilsson and Jan Sparud. The evaluation dependence tree as a basis for lazy functional debugging. *Automated Software Engineering*, 4(2):121–150, Apr 1997.

Rickard Nilsson. Scalacheck: the definitive guide. 2014.

A. Jefferson Offutt and Roland H. Untch. *Mutation Testing for the New Century*, chapter Mutation 2000: Uniting the Orthogonal, pages 34–44. Springer, 2001.

Bryan O'Sullivan, John Goerzen, and Donald Bruce Stewart. *Real World Haskell.* O'Reilly, 2008.

Carlos Pacheco and Michael D. Ernst. Randoop: Feedback-directed random testing for java. In *OOPSLA '07*, pages 815–816. ACM, 2007.

Manolis Papadakis and Konstantinos Sagonas. A proper integration of types and function specifications with property-based testing. In *Erlang'11*, pages 39–50. ACM, 2011.

Lee Pike. Smartcheck: Automatic and efficient counterexample reduction and generalization. In *Haskell'14*, pages 59–70. ACM, 2014.

J. R. Quinlan and R. M. Cameron-Jones. FOIL: A midterm report. In *ECML-93: European Conference on Machine Learning*, pages 1–20. Springer, 1993.

Jason S Reich. *Property-based Testing and Properties as Types: A hybrid approach to supercompiler verification.* PhD thesis, University of York, 2013.

Jason S. Reich, Matthew Naylor, and Colin Runciman. Advances in Lazy SmallCheck. In *IFL'13*, LNCS 8241, pages 53–70. Springer, 2013.

Rich Hickey and Reid Draper. test.check. `https://github.com/clojure/test.check`, 2013–2017.

Colin Runciman, Matthew Naylor, and Fredrik Lindblad. SmallCheck and Lazy SmallCheck: Automatic exhaustive testing for small values. In *Haskell'08*, pages 37–48. ACM, 2008.

Arto Salomaa. Two complete axiom systems for the algebra of regular events. *Journal of the ACM (JACM)*, 13(1):158–169, 1966.

Koushik Sen, Darko Marinov, and Gul Agha. *CUTE: a concolic unit testing engine for C*, volume 30. ACM, 2005.

Tim Sheard and Simon Peyton Jones. Template meta-programming for Haskell. In *Haskell'02*, pages 1–16. ACM, 2002.

Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. When do changes induce fixes? *ACM SIGSOFT Software Engineering Notes*, 30(4):1–5, 2005.

Nicholas Smallbone. *Property-based testing for functional programs*. Licentiate thesis, Chalmers University of Technology, 2011.

Nicholas Smallbone. *Lightweight verification of functional programs*. PhD thesis, Chalmers University of Technology, 2013.

Nicholas Smallbone and Tobias Gedell. Infer complexity of algorithms by testing. `https://github.com/nick8325/complexity`, 2013–2017.

Nicholas Smallbone, Moa Johansson, Koen Claessen, and Maximilian Algehed. Quick specifications for the busy programmer. *Journal of Functional Programming*, 27, 2017.

Calvin Smith, Gabriel Ferns, and Aws Albarghouthi. Discovering relational specifications. In *ESEC/FSE 2017*, pages 616–626. ACM, 2017.

Jacob Stanley. Hedgehog. `https://github.com/hedgehogqa/haskell-hedgehog`, 2017.

Don Stewart and Spencer Sjanssen. XMonad. In *Haskell '07*, pages 119–119. ACM, 2007.

Phil Stocks and David Carrington. A framework for specification-based testing. *IEEE Transactions on Software Engineering*, 22(11):777–793, 1996.

Nikolai Tillmann and Wolfram Schulte. Parameterized unit tests. *SIGSOFT Software Engineering Notes*, 30(5):253–262, September 2005.

Michael Walker and Colin Runciman. Cheap remarks about concurrent programs. Presented at TFP'17, 2017.

Malcolm Wallace, Olaf Chitil, Thorsten Brehm, and Colin Runciman. Multiple-view tracing for Haskell: a new Hat. In *Haskell'01*, pages 182–196. ACM, 2001.

Hong Zhu, Patrick A. V. Hall, and John H. R. May. Software unit test coverage and adequacy. *ACM Computing Surveys*, 29(4):366–427, December 1997.