FitSpec: Refining Property Sets for Functional Testing

Rudy Braquehais

University of York, UK
rmb532@york.ac.uk

Colin Runciman

University of York, UK
colin.runciman@york.ac.uk

Abstract

This paper 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.

Categories and Subject Descriptors D.2.5 [*Software Engineering*]: Testing and Debugging — Testing tools

Keywords property-based testing, mutation testing, systematic testing, formal specification, Haskell.

1. Introduction

Property-based testing tools automatically test a set of properties describing a set of functions. QuickCheck (Claessen and Hughes 2000) and SmallCheck (Runciman et al. 2008) are well-known examples of such tools for Haskell. 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 paper 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 (§3).

FitSpec enumerates small finite black-box mutations of functions under test (§4.1 and §4.3). It tests those mutants against the property set, recording the ones that *survive* by passing all the tests (§4.2 and §4.4). It presents the *number of surviving mutants* along with

any *smallest surviving mutant* (§4.5). 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 (§4.6).

Example 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))
```

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 paper 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 paper is organized as follows. §2 defines minimality, completeness, equivalence and implication of property sets; §3 describes how to use FitSpec; §4 describes how FitSpec works internally; §5 presents example applications and results; §6 discusses related work; §7 draws conclusions and suggests future work.

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 2   The following property set describing the standard 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 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   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 counter-example. □

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 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.   □

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). This section details this process.

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` (§4.1) and `Mutable` (§4.3) 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). Writing

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

is enough to create the necessary instances. In §4 we show how to define such instances manually, and why that is desirable in some cases.

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.

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.

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 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. How FitSpec Works

This section presents details of how FitSpec works. We explore how data values and mutants are enumerated (§4.1 and §4.3), how properties are tested (§4.2), how mutants are tested against properties (§4.4) in searches for surviving mutants (§4.5), how conjectures are made based on test results (§4.6), how we control the extent of testing (§4.7), and how we show mutants (§4.7).

4.1 Enumerating Test Data

To mitigate the combinatorial explosion when enumerating data values, FitSpec uses a size-bounded enumeration technique. The enumeration works in a similar way to Feat (Duregård et al. 2012). However, the ranking and ordering of values are defined differently to align better with our needs when enumerating functional mutants. Parallel to QuickCheck's `Arbitrary`, SmallCheck's `Serial` and Feat's `Enumerable` typeclasses, we define `Listable`:

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

A `Listable` instance's `tiers` value is a possibly infinite list of finite sublists (or tiers): 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 integers, it is the absolute value; 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.

Given `tiers`, it is easy to compute a list of all values in non-decreasing order of size:

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

`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:

```
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.

The sum and product of two tier-lists are defined by:

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

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

3

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

Table 1. Numbers of data values in successive tiers for several example data types.

So, when both tier-lists are infinite:

```
[t0,t1,t2,...] \/ [u0,u1,u2,...] =
  [ t0 ++ u0,  t1 ++ u1,  t2 ++ u2,  ... ]

[t0,t1,t2,...] >< [u0,u1,u2,...] =
  [ t0**u0
  , t0**u1 ++ t1**u0
  , t0**u2 ++ t1**u1 ++ t2**u0
  , ...
  ]
```

Each `cons<N>` is defined in terms of `><`.

Example 6   Here is a `Listable` instance for `Bool`:

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

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

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

Example 7   For the following natural-number type, defined as a wrapper over `Int`s,

```
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.   □

Example 8   Here is a `Listable` instance for lists:

```
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.   □

Example 9   As a final example, for the 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, for example:

```
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 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).

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] ]
  ]
```

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 `setsOf` and `bagsOf` functions will be useful when defining tiers of mutants (cf. §4.3) and tiers of values satisfying a data invariant (cf. §5.3, §5.4, §5.6).

4.2   Testing Properties

Using the enumeration described in §4.1, FitSpec provides several functions to check whether properties hold. Consider first:

```
holds :: Testable a => Int -> a -> Bool
```

The function `holds` takes as arguments a number of tests `n` and a `Testable` property; its result is `True` if the property is found to hold for `n` tests (or in all cases if there are fewer than `n` possibilities) and `False` otherwise. For example, to check the ordered-result property of `sort` for the first 1000 lists of naturals we may evaluate:

```
holds 1000 (\xs -> ordered (sort (xs::[Nat])))
```

The type annotation of `xs` is necessary to determine the instance of `Listable` used when enumerating values for testing.

4.3   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 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 `><`. 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 7 (revisited)   Recalling the natural-number type `Nat`, 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. 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 = if x == x' then fx' else 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)
```

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

Table 2. Numbers of inequivalent mutants in successive tiers for several original functions.

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
```

We omit details of the functions `concatMapT` and `products`, but they are straightforward.

Example 10   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 11   The first four inequivalent mutants for the identity 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 12   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 2 shows, for a few example functions, the number of inequivalent mutants in successive tiers. In the worst case, this number increases by around $3\times$ as size increases by one.

4.4   Testing Mutants against Properties

As we saw in §3, in order to collect property functions of different types into a single list, we apply FitSpec's `property` function to

5

each of them. The `property` function is polymorphic over the class of `Testable` types:

```
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
```

Example 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 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.   □

4.5   Searching for Survivors

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

Example 1 (revisited)   Recall the *incomplete* property set describing `sort` given in §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.

4.6   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.   □

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 1.

Weak	Mild	Strong	Mild	Weak

```
0%   11%        33%        66%     88% 100%
```
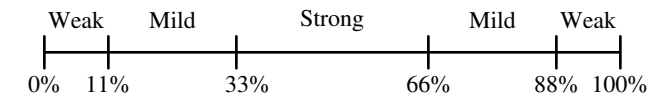
Figure 1. Conjecture strengths by % of surviving mutants.

4.7   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 §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 (§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 §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. We omit details of the `ShowMutable` class: it has a method to show a mutated value given also the original value; instances for user-defined datatypes can be automatically derived.

Example 13 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` (§3).

5. Applications and Results

In this section, we use FitSpec to refine properties of: boolean negation and conjunction operators (§5.1); sorting (§5.2); merge on min-heaps (§5.3); set membership, insertion, deletion, intersection, union (§5.4), powersets and partitions (§5.5); path and subgraph on digraphs (§5.6).

    In §5.1 and §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 §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 for the `Word2` type.

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. 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
              {2,7} ==> {5}   88% killed
              {2,4} ==> {5}   88% killed
              ... 6 conjectures omitted ...
```

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

Table 3. How enlarging the sorted element-type increases the time required for convergence. In practice, `Word2` is sufficient to obtain good results.

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, but it is correct — all idempotent binary boolean operators are associative.

5.2 Sorting

Consider the following properties of `sort`, which are similar to those given in the introduction. This set of properties is a complete but not minimal specification of `sort`.

```
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: the reported minimal property-subsets and conjectures stay the same. The smaller the type, the lower the values of test-control parameters, and the less run-time, we need to obtain convergence (see Table 3). 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 3, 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] ]
```

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 counter-example 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 §7.

5.3 Binary Heaps

In this section, we 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 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`). It might seem strange that property 3 specify `merge`, but looking at its definition, we can see why:

```
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. In order to check that property 3 alone should be excluded as a complete specification, one option would be to add `insert` to the tuple of functions to be mutated.

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

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.

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 §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. (My fault! CR) 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 `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!

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. (The reader is invited to write down a few properties they expect these functions to satisfy.)

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 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`:

Example	#-mutants	#-tests	time	space
Bool (§5.1)	63	8	< 1s	18MB
Sorting (§5.2)	4000	4000	12s	62MB
Heaps (§5.3)	4000	2000	42s	102MB
Basic Sets (§5.4)	750	1024	5s	22MB
Sets of Sets (§5.5)	17441	256	5s	58MB
Digraphs (§5.6)	750	1500	12s	1853MB

Table 4. 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`).

```
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`:

```
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)
```

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.

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 §5.2, for polymorphic functions, the element 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.

6.   Related Work

Since the introduction of QuickCheck (Claessen and Hughes 2000), several other property-based testing libraries and techniques have been developed, such as Smallcheck, Lazy SmallCheck (Runciman et al. 2008; Reich et al. 2013) and Feat (Duregård et al. 2012).

QuickSpec   Claessen et al. (2010) present the QuickSpec tool, which is able to generate algebraic specifications automatically. 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 (§5.1) and Heaps (§5.3)   As we show in §5.1 and §5.3, FitSpec can assist in the refinement of specifications generated by QuickSpec.

Basic Sets (§5.4)   For comparison, consider again the basic functions of the set library (§5.4), an example where we did *not* start with QuickSpec-generated properties. We can compare our final specification with QuickSpec's output. For comparison, in the `Set` library example, QuickSpec 1 (Claessen et al. 2010) generates a complete specification with 70 properties. QuickSpec 2 (Smallbone and Johansson 2016) generates a complete specification with 43 properties, not including any of ours.

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. 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. 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 §5.

Sorting (§5.2)   Consider the following explicit definition of `sort`, which is used as an example in 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 §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. 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 (§5.4)   MuCheck derives no mutants for any of `insertS`, `deleteS`, `subS`, `\/` or `/\` (cf. §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.

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.

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.

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 §3 and §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 §5 FitSpec's applicability for a range of small examples, and we have briefly compared in §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 §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 §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. We shall return to this issue shortly, when considering future work.

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 §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.

Ease of use   Arguably, a tool is easier to use if it requires less work from the programmer. As we illustrated in §3, writing a minimal program to apply 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; Lindblad 2007).

Future Work   Finally 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 §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`

Acknowledgements

References

L. Bulwahn. Smart testing of functional programs in Isabelle. In *LPAR 2012*, LNCS 7180, pages 153–167. Springer, 2012.

K. Claessen and J. Hughes. QuickCheck: A lightweight tool for random testing of Haskell programs. In *ICFP'00*, pages 268–279. ACM, 2000.

K. Claessen, N. Smallbone, and J. Hughes. QuickSpec: Guessing Formal Specifications Using Testing. In *TAP 2010*, LNCS 6143, pages 6–21. Springer, 2010.

R. Demillo, R. Lipton, and F. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, April 1978.

J. Duregård, P. Jansson, and M. Wang. Feat: functional enumeration of algebraic types. In *Haskell'12*, pages 61–72. ACM, 2012.

J. Duregård. Ultra-Lightweight Black Box Mutation Testing. `https://youtu.be/ROKxri62WYQ`, 2014. Accessed 1 April 2016.

A. Gill and C. Runciman. Haskell Program Coverage. In *Haskell'07*, pages 1–12. ACM, 2007.

Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering*, 37(5): 649–678, Sept 2011.

D. Le, M. A. Alipour, R. Gopinath, and A. Groce. MuCheck: An extensible tool for mutation testing of Haskell programs. In *ISSTA 2014*, pages 429–432. ACM, 2014.

F. Lindblad. Property directed generation of first-order test data. In *TFP'07*, pages 105–123, 2007.

B. Marick. How to misuse code coverage. In *International Conference on Testing Computer Software*, pages 16–18, 1999.

J. S. Reich, M. Naylor, and C. Runciman. Advances in Lazy SmallCheck. In *IFL'13*, LNCS 8241, pages 53–70. Springer, 2013.

C. Runciman, M. Naylor, and F. Lindblad. SmallCheck and Lazy SmallCheck: Automatic exhaustive testing for small values. In *Haskell'08*, pages 37–48. ACM, 2008.

T. Sheard and S. P. Jones. Template meta-programming for Haskell. In *Haskell'02*, pages 1–16. ACM, 2002.

N. Smallbone and M. Johansson. Quick specifications for the lazy programmer. Submitted for publication, 2016. URL `http://www.cse.chalmers.se/~nicsma/papers/quickspec2.pdf`.