Testing Type Class Laws

Johan Jeuring
Utrecht University and Open Universiteit, the Netherlands
J.T.Jeuring@uu.nl

Patrik Jansson
Chalmers University of Technology, Sweden
patrikj@chalmers.se

Cláudio Amaral
Chalmers University of Technology, Sweden
LIACC - University of Porto, Portugal
amaral@chalmers.se
coa@ncc.up.pt

Abstract

The specification of a class in Haskell often starts with stating, in comments, the laws that should be satisfied by methods defined in instances of the class, followed by the type of the methods of the class. This paper develops a framework that supports testing such class laws using QuickCheck. Our framework is a light-weight class law testing framework, which requires a limited amount of work per class law, and per datatype for which the class law is tested. We also show how to test class laws with partially-defined values. Using partially-defined values, we show that the standard lazy and strict implementations of the state monad do not satisfy the expected laws.

Categories and Subject Descriptors D.1.1 [Programming Techniques]: Applicative (Functional) Programming

General Terms design, languages, verification

Keywords laws, classes, testing, state monad

1. Introduction

The specification of a class in Haskell starts with specifying the class methods, with their type signatures and often also the laws that should be satisfied. The signatures are part of the Haskell code and instances are checked for conformance by the compiler, but the class laws are normally just comments, leaving the laws unchecked. For example, Figure 1 gives the Haskell 2010 Language Report [Marlow 2010] specification of the Functor class, and Figure 2 gives parts of the specification of the Monad class.

A class law typically takes a number of arguments, and then formulates an equality between expressions in which both the arguments and values of the class type variable are used. The arguments of a law are universally quantified, as are the values of the class type variable. For example, the second functor law takes two arguments $f$ and $g$, and compares expressions obtained by mapping $f$ and $g$ in different ways to a value of the class type. The laws for class methods are central to the definition of classes but, unfortunately, Haskell provides no language support for stating or checking such laws.

class Functor f where
  fmap :: (a -> b) -> f a -> f b

The Functor class is used for types that can be mapped over. Instances of Functor should satisfy the following laws:

- $\text{fmap id} :: \text{id}$
- $\text{fmap (f \circ g)} :: \text{fmap f \circ fmap g}$

The instances of the class Functor for lists, Data.Maybe.Maybe and System.IO.IO satisfy these laws.

Figure 1. Specification of the Functor class in the Haskell report [Marlow 2010].

Instances of Monad should satisfy the following laws:

- $\text{return a} :: \text{a}$
- $\text{m >>= return} :: \text{m}$
- $\text{(\lambda x \rightarrow k x >>= h) (m >>= k)} :: \text{h}$

Instances of both Monad and Functor should additionally satisfy the law:

- $\text{fmap f xs} :: \text{xs} >>= \text{return} \circ f$

The instances of the class Monad for lists, Data.Maybe.Maybe and System.IO.IO defined in the Prelude satisfy these laws.

Figure 2. The Monad laws from the Haskell report [Marlow 2010].

Since class laws are central to the definition of some classes, we would like some guarantees that the laws indeed hold for instances of the class. There are several ways in which such guarantees can be obtained. To show that the laws are satisfied for a particular class instance, we can construct a proof by hand, use a theorem prover to construct a proof for us, or test the law with the QuickCheck [Claessen and Hughes 2000] library. In this paper we develop a framework for specifying class laws such that we can easily use QuickCheck to test a law for a class instance. In our framework we define a single function quickLawCheck to test any class law (of a certain form) on any datatype. This requires a small amount of work for each class law, and for each datatype. The main technology that makes this possible is type families [Chakravarty et al. 2005].

Default QuickCheck generators do not test properties for partially-defined values and the standard equality check cannot test partial values for equality. Since some classes make essential use of laziness, we want to be able to test class laws on partially-defined values too. The ChasingBottoms library developed by Danielsson and Jansson [2004] allows us to distinguish exceptional (‘bottom’)

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Haskell’12. September 13, 2012, Copenhagen, Denmark. Copyright © 2012 ACM 978-1-4503-1574-6/12/09... $10.00
values from other values. We use this library, and provide generators and equality tests suitable for testing class laws on partially-defined values. As an example we show that neither the lazy nor the strict state monad implementations satisfy the laws expected for such instances if values may be partially defined.

In this paper we make the following contributions:

- We develop a framework that supports specifying testable laws for a class.
- We make it easy to test a class law for a class instance.
- The framework supports stating and checking “poor man’s proofs” (representing equality reasoning) for the laws.
- We show that the standard strict and lazy implementations fail to satisfy the monad laws for partially-defined values.

This paper is organised as follows. Section 2 introduces our framework by showing how a user can test the monoid laws for an instance of the `Monoid` class. Section 3 shows how a user can specify laws in our framework in such a way that they can be easily tested. Section 4 shows how a user can add evidence (“poor man’s proofs”) to a class law. Section 5 describes what a user needs to do to test a class law on a datatype. Section 6 summarises the previous sections by describing the various components of the framework. Section 7 shows how to use the framework for testing with partial values. Section 8 explores different state monad implementations and explains their (non-)conformance with the laws. Section 9 gives related and future work and concludes.

2. Testing the monoid laws

This section uses common instances of the `Monoid` class to introduce our class-laws testing framework.

**The Monoid class.** The `Monoid` class, defined in the module `Data.Monoid` in Haskell’s base libraries, has the methods:

```haskell
mempty :: a
mappend :: a -> a -> a
```

together with a method `mconcat :: [a] -> a` which we won’t use in this paper. We will write infix `★` for `mappend`. Implementations of these methods in an instance of `Monoid` should satisfy the following three laws:

- `mempty ★ m = m`
- `m ★ mempty = m`
- `l ★ (m ★ r) = (l ★ m) ★ r`

**Testing Monoid laws using QuickCheck.** The `Monoid` laws are easily formulated as polymorphic QuickCheck properties:

```haskell
monoidLaw1 m = mempty ★ m ★ m
monoidLaw2 m = m ★ mempty ★ m
monoidLaw3 l m r = l ★ (m ★ r) ★ (l ★ m) ★ r
```

can be tested as follows for the `Monoid` instance for lists

```haskell
main = do
quickCheck (monoidLaw1 :: [Int] -> Bool)
quickCheck (monoidLaw2 :: [Int] -> Bool)
quickCheck (monoidLaw3 :: [Int] -> [Int] -> Bool)
```

Running `main` doesn’t lead to any counterexamples, as expected.

Throughout this paper we just pick monomorphic types (like `Int` here) by hand, but in general we should use the schema from Testing Polymorphic Properties [Bernardy et al. 2010] to find the best type.

**Testing laws for datatypes with functions.** What if we want to test whether or not the `Monoid` instance of the type `Endo a`:

```haskell
newtype Endo a = Endo { appEndo :: a -> a }
```

satisfies the `Monoid` laws? Adding the line

```haskell
quickCheck (monoidLaw1 :: Endo Int -> Bool)
```

to `main` gives, amongst others, the error message that we have no instance of `Eq (Endo Int)`. This is a reasonable error message, since indeed we have no equality for functions. How can we test two `Endo a`-values `l` and `r` for equality? If `a` is finite we can test equality of `appEndo l x` and `appEndo r x` for all possible inputs `x :: a`. But for big or infinite types, complete coverage is infeasible or impossible. Instead we add a parameter to generate random `a`-values. So to test equality of two `Endo a`-values `l` and `r`, we generate arbitrary values of type `a`, and test equality of `appEndo l` and `appEndo r` when applied to these random values.

Later in this paper we will also discuss laws for the `State` monad, where `State` is defined by:

```haskell
newtype State s a = State { runState :: s -> (a, s) }
```

To test equality of two `State s a`-values `l` and `r`, we need to generate an `s`-value, and compare `runState l x` with `runState r x`.

Since we also want to test laws for datatypes like `Endo a` and `State s a`, we replace the standard equality in testing by a method `testEqual`. Function `testEqual` also returns a boolean, but what arguments does it take? Function `testEqual` is a generalisation of `(★)`, so a first approximation for its type is `a → a → Bool`. This would be fine for a type such as `Int`, but is not appropriate for testing `Endo a` and `State s a`. For testing these types, `testEqual` needs an extra parameter, which depends on the type to be tested. To represent the parameter, we introduce a type family `Param`:

```haskell
type family Param b
```

The `Param` type family is defined for each datatype on which we want to test a law. For example, to determine the equality of values of `[a]`, `Endo a` and `State s a`, we define

```haskell
type instance Param [a] = ()
type instance Param (Endo a) = a
type instance Param (State s a) = s
```

We do not need an extra parameter to test list values, so the `Param` instance for lists is the empty tuple type. Now we can define the class `TestEqual`

```haskell
class TestEqual a where
  testEqual :: a → a → Param a → Bool
```

together with the instances:

```haskell
instance Eq a ⇒ TestEqual [a] where
  testEqual l r = l ★ r
instance Eq a ⇒ TestEqual (Endo a) where
  testEqual l r p = appEndo l p == appEndo r p
instance (Eq a, Eq s) ⇒ TestEqual (State s a) where
  testEqual l r s = runState l s ★ runState r s
```

Using `testEqual` for the `Monoid` laws. We could now replace `★` with `testEqual` in the `Monoid` laws, but for greater flexibility we first factor out the testing part by introducing an intermediate type `Equal a` for equality tests. Instead of a boolean, a law now returns a pair of values\(^1\). This choice makes it possible to easily experiment with different notions of equality without changing the “law” part.

\(^1\) In Sec. 4 we generalise this pair to a list of steps in a “poor man’s proof”.
the arguments of the law. Furthermore, to use function with the steps of a poor man’s proof in Section 4. To turn a law in the
We can use this new formulation of the laws to test whether or not the Monoid-instance of the law satisfies the Monoid-laws.

From quickCheck to quickLawCheck. The expressions that test the laws become quite verbose when we use testEqual. A first step towards making testing laws easier is to redefine the type of the method testEqual of the class TestEqual.

class TestEqual a where
testEqual :: Equal a → Param a → Property

The method testEqual now takes an Equal a-value as argument, instead of two a-values, and it returns a property instead of a boolean. Using Equal a-values as arguments, we get rid of the occurrences of uncurry in the arguments to quickCheck, and returning a property gives us more flexibility in the definition of testEqual. Furthermore, we will abstract from the common structure to arrive at the following form of the above tests (where un is just the dummy value undefined):

main = do
  quickLawCheck (un :: MonoidLaws (Endo Int))
  quickLawCheck (un :: MonoidLaws (Endo Int))
  quickLawCheck (un :: MonoidLaws (Endo Int))

In the rest of this section we will introduce the machinery to make this possible.

Function quickLawCheck is just quickCheck ∘ lawtest where lawtest turns a “law” into a testable property. Our next step is to explain how laws are represented.

Representing laws. Since monoids are specified as a class, and the laws are specified in (comments) in the class, we define a class MonoidLaws in which we specify the laws for monoids, together with their default instances.

class Monoid m ⇒ MonoidLaws m where

monoidLaws1 :: m → Equal m
monoidLaws2 :: m → m → Equal m
monoidLaws3 :: m → m → m ⋆ m

Note that instances can override the default instances for laws given in the MonoidLaws class. We will use this feature to extend a law with the steps of a poor man’s proof in Section 4. To turn a law into a testable property, we need to generate arbitrary values for the arguments of the law. Furthermore, to use function testEqual to test equality on the datatype on which the law is tested, we need to generate values of the parameter type. Since different laws take different numbers and types of arguments, we introduce another type family to represent the arguments of a law:

type family LawArgs t

We cannot make class methods instances of a type family, so for each law we introduce a datatype without values:

data MonoidLaws1 m
data MonoidLaws2 m
data MonoidLaws3 m

Now we can create instances of the type family LawArgs, which we will later connect to the class methods for the laws.

type instance LawArgs (MonoidLaws1 m) = m
type instance LawArgs (MonoidLaws2 m) = m
type instance LawArgs (MonoidLaws3 m) = (m, m, m)

In the body of the monoid laws, we compare two monoid values. To compare these two values, we use function testEqual. It follows that we need to detect the parameter type of the body of the law. We introduce yet another type family to describe the type appearing in the body of the law.

type family LawBody t

and for the three monoid laws we declare:

type instance LawBody (LawArgs t) = Equal (LawBody t)
class Monoid m ⇒ MonoidLaws m where

monoidLaws1 :: Law (MonoidLaws1 m)
monoidLaws2 :: Law (MonoidLaws2 m)
monoidLaws3 :: Law (MonoidLaws3 m)

Here we connect the datatypes for monoid laws to their respective class methods. This definition of the class MonoidLaws, together with the default instances, replaces the earlier definition given in this paragraph.

Testing laws. Using the type families LawArgs, LawBody, and Param, we can finally specify the type of the function lawtest. Since we use lawtest on values of different types, we let lawtest be the method of a class LawTest. Class methods have to refer to the type variable introduced by the class, so we add a dummy first argument to the lawtest method that steers its type.

class LawTest t where

lawtest :: t → LawArgs t → LawBody t → Property

In general, a type t cannot be recovered from a type family, such as LawArgs t. If we had used data families instead of type families we could have recovered the t, but using data families leads to many extra constructors, and we prefer to use type families. A law that is passed as argument to quickLawCheck is specified by an un-value of its corresponding type. The un-value is never used in function lawtest. The instances of LawTest for the monoid laws are easy:
instance (MonoidLaws m, TestEqual m) => LawTest (MonoidLaw_1 m) where
lawtest _ = testEqual ∘ monoidLaw_1
instance (MonoidLaws m, TestEqual m) => LawTest (MonoidLaw_2 m) where
lawtest _ = testEqual ∘ monoidLaw_2
instance (MonoidLaws m, TestEqual m) => LawTest (MonoidLaw_3 m) where
lawtest _ = testEqual ∘ monoidLaw_3

Testing laws with functional arguments. Some laws take functions as arguments. For example, the second functor law in Figure 1 takes two functions as arguments. Using quickLawCheck to test this law gives the error message that there is no instance of Show for functions. To test this law, and other laws that take functions as arguments, we introduce quickFLawCheck, a variant of quickLawCheck that doesn't require the types of all arguments of a law to be instances of the Show class. Using quickFLawCheck leads to rather incomprehensible error reports when a counterexample is found. To obtain a comprehensible counterexample, we have to introduce a Show instance for the function type that is used, for example by showing the function results on a few arguments.

Putting it all together. Using the definitions introduced in this section, we can make Endo a an instance of MonoidLaws:

instance MonoidLaws (Endo a)
and then we can write

main = do
  quickLawCheck (un :: MonoidLaw_1 (Endo Int))
  quickLawCheck (un :: MonoidLaw_2 (Endo Int))
  quickLawCheck (un :: MonoidLaw_3 (Endo Int))

to test the monoid laws for Endo a. As expected, QuickCheck does not find any counterexamples. In Sections 7 we will show how to define a function quickLawCheckPartial, which also tests laws for partially-defined values. If we replace quickLawCheck by quickLawCheckPartial in main, QuickCheck gives counterexamples for the first two monoid laws. The counterexamples represent the inequalities \( id \circ \bot = \text{const} \neq \bot \) and \( \bot \circ id = \text{const} \neq \bot \), where \( \bot \) (pronounced “bottom”) is the least defined value of any domain. Note that we use un (short for undefined) for a dummy value used essentially as a type argument, and \( \bot \) to build a partial value used in testing.

3. Specifying class laws

This section shows how a user can add laws to a class using our framework, by showing how the laws for functors are specified.

The module Control.Monad.Laws from our framework contains all the laws specified in comments in the Haskell 2010 Control.Monad module. But what if you define your own class, instances of which should satisfy a particular set of laws? This section shows how you can specify laws for a class, by showing how we specify the laws for the functor class.

The functor laws are specified in the Functor class in Figure 1. Here we define them in our framework, giving them names starting with default because we will use these definitions as defaults for instances of the class FunctorLaws.

- defaultFunLaw_1 \( x \) = fmap id x, defaultFunLaw_2 \( f, g, x \) = (fmap f ∘ fmap g) x

At the moment we still have to explicitly provide the arguments to the laws. It is future work to lift this restriction. The first functor law takes an argument \( x \) of type \( a \) for some \( f :: x \rightarrow a \) and some \( a \). We define the instance of LawArgs for the datatype FunLaw_1 corresponding to this law as follows:

- data FunLaw_1 a (f :: x \rightarrow a) = Law (FunLaw_1 a f)

The second functor law takes a triple of arguments: two functions, and a value on which the composition of these functions is mapped.

- data FunLaw_2 a b c f = (b \rightarrow c, a \rightarrow b, f a)

For the type of the body of the laws, we have to make explicit which of the argument type variables appear in the body.

- type instance LawBody (FunLaw_1 a f) = f a
- type instance LawBody (FunLaw_2 a b c f) = (b \rightarrow c, a \rightarrow b, f a)

Now we define the class FunctorLaws:

- class Functor f => FunctorLaws f where
- funLaw_1 :: Law (FunLaw_1 a f)
- funLaw_2 :: Law (FunLaw_2 a b c f)
- funLaw_1 = defaultFunLaw_1
- funLaw_2 = defaultFunLaw_2

We make these datatypes instances of LawTest as follows:

- instance (FunctorLaws f, TestEqual (f a)) => LawTest (FunLaw_1 a f) where
- lawtest _ = testEqual ∘ funLaw_1
- instance (FunctorLaws f, TestEqual (f c)) => LawTest (FunLaw_2 a b c f) where
- lawtest _ = testEqual ∘ funLaw_2

To implement laws for a class \( C \) in our framework, we define one empty datatype per law, for which we define instances of two type families. We then define a class CLaws in which we specify the laws for \( C \). To test the laws, they are made instances of the class LawTest.

4. Adding evidence to a law

This section shows how we can add evidence to a law in the form of a "poor man’s proof”, and test the evidence. The “proof” is expressed as a list of steps in an equality reasoning argument for why the law holds. For example, if we prove a law \( \text{lhs} = \text{rhs} \) in a scientific paper, we typically write

\[
\begin{align*}
\text{lhs} &= \{ \text{good reason} \} \\
\text{lhs}' &= \{ \text{another good reason} \} \\
\text{rhs}
\end{align*}
\]

In this section we show how we express this proof as a list of expressions \([\text{lhs}, \text{lhs}', \ldots, \text{rhs'}\), \text{rhs}]\), which requires that the types of the expressions are the same, and makes it possible to test equality of adjacent pairs, and hence of all expressions. The basic idea of these “proofs” is independent of the type family machinery used for ClassLaws. We used an early version already in 2001 when preparing [Janssen and Jeuring 2002], resulting in over 5000 lines of poor man’s proofs.

Suppose we define our own kind of lists,

- data List a = Nil | Cons a (List a)
5. Testing class laws

This section shows what we need to do to test a class law on an instance of the class for a particular datatype.

To test a law on a datatype using our framework, we need three instances for the datatype:

- an `Arbitrary` instance to generate arbitrary values of the datatype. The `Arbitrary` instance is needed for the body of the law, which usually is a value of the datatype itself.
- a `Show` instance to present a counterexample if such an example is found.
- a `TestEqual` instance for testing equality of a list of values.

For example, for the `Arbitrary` instance for the type `List`, we translate the arbitrary values generated by the `Arbitrary` instance for standard lists `[]` provided by QuickCheck to `Lists`. We derive the `Show` instance for `List`s, and define the following instance of `TestEqual`:

```haskell
instance (Eq a, Show a) => TestEqual (List a) where
  testEqual p = testEq (\x -> p)
```

In the `FunctorLaws List` instance, we specify that we think that the left-hand side of the first functor law (defaultFunLaw1) equals the right-hand side, and that evidence is provided by the list of steps given in the second argument of `addSteps`. For this to work, we have to change the `Equal` type, and its `constructor` \( \cdot \cdot \cdot \) into a list of values instead of a pair of values:

```haskell
type Equal = []

type Theorem = Equal

addSteps :: Theorem a -> Equal a -> Equal a

addSteps \( \cdot \cdot \cdot \) = error "addSteps . . ."
```

Function `addSteps` returns a list of values, which are pairwise tested for equality. Testing gives a counterexample:

```haskell
(5, Cons 1 (Cons 0 Nil), Cons 0 (Cons 1 Nil))
```

The first component (5) of the triple denotes the first position in the evidence where it fails to be a chain of equal expressions. Here, the fifth and sixth expressions are unequal and thus break the evidence chain. Since function `addSteps` includes the evidence steps in between the left-hand side and right-hand side of the law, and since we have a non-empty example here, a `consCase`, this implies that there are counterexamples for the equality of `snoc y ys` and `id (Cons y ys)`. This is indeed true: `snoc y ys` appends \( y \) to the end of \( ys \), instead of to the front. Any list with at least two different elements provides a counterexample.
Function `testEq` takes an equality operator and a list of values to be tested for equality, and returns a property, which tests consecutive elements for equality with the function `pairwiseEq`.

```
  testEq :: [a] -> Eq [a] -> Property
  testEq [] = $ property $ liftBool $ pairwiseEq (s z)

  whenFail :: [a] -> Bool -> Property
  whenFail [x] y = x == y
  whenFail xs y = pairwiseEq (xs) (y : ys)
```

The function `pairwiseEq` takes a list of values to be tested for equality, and returns a property, which tests consecutive elements in the list to be equal by means of the function `pairwiseEq`.

```
  pairwiseEq :: [a] -> [a] -> Property
  pairwiseEq [] [] = $ property $ liftBool $ pairwiseEq (s z)

  pairwiseEq (x : xs) (y : ys) = whenFail x y $ pairwiseEq xs ys
```

The laws themselves as functions `default1`, `default2`, ..., which take the type arguments used in the types of the laws as argument, and have no right-hand sides.

- Datatypes `L1`, `L2`, ..., which take the type arguments used in the types of the laws as argument.
- Type family instances for the datatypes `L1`, `L2`, ..., in which the instance for `LawArgs` specifies the types of the universally quantified arguments for the law, and `LawBody` specifies the type of the elements tested for equality.

A class `CLaws` with methods `l1`, `l2`, ..., which take the `LawArgs` of the corresponding datatype as argument, and return a value of the `Equal`-type for the `LawBody`. The laws are given default instances `l1 = default1`, etc.

- Instances of the class `LawTest` for the datatypes `L1`, `L2`, ..., in which `lawtest` is defined by `lawtest = testEqual l1`, etc.

For testing any law on a datatype `D` in our framework, we have to provide:

- A `D` instance of the type family `Param`, specifying the extra information necessary for testing equality of values of `D`.
- A `D` instance of the class `TestEqual`, with a method `testEqual` specifying how we test equality of values of type `D`.
- `D` instances of the classes `Arbitrary` and `Show`.

To test class laws `CLaws` on a datatype `D` for a `D` instance of `C`, we have to provide:

- An empty `D` instance of `CLaws`.

```
  testRunEq :: Show s => [a] -> (Pos, a, a) -> TestEqual
  testRunEq (s:ys) = case y of
    True -> $ property $ liftBool $ pairwiseEq (s z) (y : ys)
    False -> $ property $ whenFail (y : ys)
```

Besides the `TestEqual` instance, we also need to provide `Arbitrary` and `Show` instances for these types. A possible `Arbitrary` instance for `Endo a` lifts the arbitrary instance for `a`:

```
  instance Arbitrary a => Arbitrary (Endo a)
```

Showing a function is slightly more challenging:

```
  instance Show (Endo Int) where
  show (Endo f) = concat $ map (show $ f) [0..10]
```

where the `Show` instance just shows a small sample of `f`-values.

6. The `ClassLaws` framework

This section summarises the previous sections by giving an overview of our framework.

To specify one or more laws `l1`, `l2`, ... for a class `C` in our framework, we need to specify:

- The laws themselves as functions `default1`, `default2`, ...
quickLawCheck \( \text{un} :: \text{MonoidLaw}_1 \ (\text{Endo Int}) \). We do not want to change the type of the monoid law itself to also include partial values, so we change the implementation of quickLawCheck instead. The implementation of quickLawCheck uses testEq on the monoid law to test the law on random values. The TestEqual instance used to test the first monoid law on Endo a, uses testEq \( (\triangleleft) \), thus “normal” equality. We have to replace this function to ensure that partial values are generated (by passing arguments of type Partial a, and declaring a special instance of Arbitrary for Partial a). Furthermore, the equality test used should take partiality of values into account.

**Function quickLawCheckPartial.** The change to the TestEqual instance to also take partial values into account requires changes at all intermediate levels in the code too, which makes the change rather laborious. To avoid users having to change their types at many places, we introduce function quickLawCheckPartial, which takes a law as introduced in the ClassLaws framework as argument, and tests the law also with partially-defined values. The next section gives an extensive example of how the adapted functionality is used to show that none of the standard implementations of the state monad satisfies the state monad laws.

Function quickLawCheckPartial is defined by

\[
\text{quickLawCheckPartial} = \text{quickCheck} \circ \text{Partial} \circ \text{lawtest}
\]

Note that Partial is wrapped around a predicate taking two arguments, namely the law arguments and the parameter of the body of the law.

**Making Partial prop testable.** Function quickCheck requires the type of its argument to be an instance of Testable. The Testable class contains types which can be tested—here is a somewhat simplified presentation:

- the types Bool and Property are Testable, corresponding to properties without parameters,
- a function type \( a \to \text{prop} \) is Testable if prop is Testable and a is an instance of Arbitrary.

We copy the QuickCheck class structure to handle “partial laws”: we define the class TestablePartial here and ArbitraryPartial later.

**class TestablePartial prop where**

\[
\text{propPartial} :: \text{prop} \to \text{Property}
\]

The function propertyPartial has the same type as QuickCheck’s property function, but also takes values that may be partial into account when testing. To make a “partial law” Testable, we make Partial prop an instance of Testable.

**instance TestablePartial prop \( \Rightarrow \) Testable (Partial prop)**

**where**

\[
\text{property (Partial x) = propertyPartial x}
\]

So if the type prop is testable in the partial setting, Partial prop is testable using QuickCheck.

The value sent to quickCheck is of the form Partial f with \( f = \text{lawtest} \) law of type LawArgs t \( \to \) Param (LawBody t) \( \to \) Property. We provide TestablePartial instances for Property and functions in the same style as QuickCheck so that we can test all our predicates with partially-defined values.

Property (and Bool) are made instances of TestablePartial by reusing their instances of Testable.

**instance TestablePartial Property where**

\[
\text{propertyPartial} = \text{property}
\]

The instance of TestablePartial on function types is more interesting:

**instance (ArbitraryPartial a \( \Rightarrow \) \text{Show} (\text{Partial a})) \Rightarrow \text{TestablePartial prop}**

\[
\text{where}
\]

\[
\text{propertyPartial f = forAllShrink arb shr prop where}
\]

\[
\text{arb} = \text{fmap ArbitraryPartial arb}
\]

\[
\text{shr (Partial x) = map Partial (shrinkPartial x)}
\]

\[
\text{prop (Partial x) = propertyPartial (f x)}
\]

The instance of TestablePartial on function types turns a function \( f \) into a property using the QuickCheck function forAllShrink. Function forAllShrink takes a generator, a shrinking function, and a property as argument. The generator generates values using arbitraryPartial. The shrinking function, which is used whenever a counterexample is found, shrinks counterexamples using the ArbitraryPartial method shrinkPartial, defined below. The property applies function \( f \) to the generated value, and calls propertyPartial again. The instance of TestablePartial on function types requires a testable co-domain and the possibility to generate and show possibly partial values of the domain. For the latter requirements we give an instance of Show for Partial a, and an instance of the class ArbitraryPartial for a, where the class ArbitraryPartial is defined by:

**class ArbitraryPartial a where**

\[
\text{arbitraryPartial :: Gen a}
\]

\[
\text{shrinkPartial :: a \to [a]}
\]

To check a property for Partial values, QuickCheck now generates values using the generator given in the ArbitraryPartial instance instead of the arbitrary instance.

**Working with partial values.** To show, detect and compare partial values we build on the ChasingBottoms library [Danielsson and Jansson 2004]². Every (boxed) type in Haskell has a least defined “bottom”-value. When generating partial values we use \( \perp \) (defined to be error "_|_" ) to represent this bottom. (Note that we write \( \perp \) instead of un, to distinguish generated bottom values from the un values passed to lawtest to steer the type.) The ChasingBottoms library provides an unsafe function isBottom \( :: a \to \text{Bool} \) that tries to determine whether or not a value is bottom. Note that we simplify matters here. In a precise semantics for Haskell there would be several different “bottoms”: non-termination, different exceptions, etc. But we lump these together in one bottom for this paper. The test isBottom a returns False if a is distinct from bottom, True for certain exceptions (see the ChasingBottoms documentation for the details) and fails to terminate if a fails to terminate.

The library also exports a SemanticEq class which lets us check equality (with \( ==! \)) and a SemanticOrd class that lets us check domain order (with \( <=! \)) and determine the most defined value \( (x \mid y) \) that is at most as defined as both \( x \) and \( y \), the meet of the two values. With these operations we can provide instances of classes such as Show and ArbitraryPartial that deal with potentially partially-defined values. For example, tLess and tMeet both terminate and evaluate to True:

\[
\text{tLess} = \perp <=! (\text{const } \perp :: \text{Bool } \to \text{Bool})
\]

\[
\text{tMeet} = (\perp, \perp', \perp') /\!\!/ (\perp', \perp, \perp') ==! (\perp, \perp, \perp')
\]

To work around some problems with ChasingBottoms we use our own classes SemanticEq, SemanticOrd and SemanticMeet below. (We aim to submit patches to the package soon.)

**Generating partial values.** A user of our library has to provide functions that also generate partially-defined values by providing ²See http://hackage.haskell.org/package/ChasingBottoms for the corresponding software package.
instances of ArbitraryPartial a for all types a for which partial values should be generated.

For finite types such as Int and Char it is easy to define instances of ArbitraryPartial, using their Arbitrary instances defined in QuickCheck. To generate possibly partial values, function genPartial introduces ⊥-values in the set of values generated by another generator. Function genPartial takes as arguments the ratio between bottom values and values from a given generator, represented as two integers, and a generator, and returns a new generator using the ratio. We pick ratios so that ⊥s appear reasonably often, since we are particularly interested in testing values that contain partial values.

genPartial :: Int → Int → Gen a → Gen a
genPartial rb ra ga = frequency [(rb,return ⊥),(ra,ga)]

instance ArbitraryPartial Int where
  arbitraryPartial = genPartial 1 20 arbitrary

instance ArbitraryPartial Char where
  arbitraryPartial = genPartial 1 20 arbitrary

To test laws with functions as arguments, such as the second functor law, we want to generate arbitrary continuous functions, not just totally defined ones. Generating partial functions requires some extra machinery. Haskell functions are monotonous (and continuous), that is, they preserve the order of the elements of the domain. We need to guarantee that the arbitrary functions we generate are monotonous. This is in general a complex problem but in the following instance of ArbitraryPartial on function types e → s we limit ourselves to bounded enumerations e and types with a SemMeet s instance. Bounded enumerations give us flat domains, which makes it relatively easy to preserve the order in the codomain.

instance (Enum e, Bounded e, Eq e, SemMeet s, ArbitraryPartial s) ⇒ ArbitraryPartial (e → s) where
  arbitraryPartial = arbitraryPartialFun arbitraryPartial

To obtain an arbitrary partial function, we first create a function table which binds an arbitrary value of the codomain to each domain value. Since our domain is a bounded enumeration, its values consist of ⊥ together with all elements of the domain: enumElems = [minBound..maxBound]. We then turn this table into a function by means of the function table2fun.

arbitraryPartialFun :: ∀ a e. (Enum e, Bounded e, Eq e, SemMeet a) ⇒ Gen a → Gen (e → a)
arbitraryPartialFun ag = do
  funtab ← forM (⊥:enumElems::[e]) (\x → ag)
  genPartial 1 6 (return (table2fun funtab))

Function table2fun returns a monotonic function by ensuring that the image of bottom is the meet of all possible values.

type FunTab e s = [s]
table2fun :: (Enum e, Bounded e, Eq e, SemMeet a) ⇒ FunTab e a → (e → a)
table2fun tab@(⊥::tottab) = fun
  where meet = foldr1 (\x! y) tab
        fun x | isBottom x = meet
              | otherwise = tottab!! (fromEnum x)

With this setup we generate arbitrary partial functions from bounded enumerations. We could extend this to more general function types, but these definitions are sufficient to find counterexamples for the laws in the next section.

Showing partial values. Just as we need to generate partial functions, we need to show partially-defined values, since the counterexamples found when testing might include partial values. We give an instance Show (Partial a) for all types a for which we want to show partial values. It is easy to show values of type Partial a if we have an instance of Show for a, by using isBottom to distinguish between partial and total values.

instance Show (Partial Int) where
  show (Partial i) = showPartial "Int" show i

showPartial :: String → (a → String) → a → String
  showPartial t ⊥p | isBottom p = "⊥" ++ t ++ "" ++ p
  showPartial ⊥ f p = f p

Showing functions is slightly more challenging. If a function appears in a counterexample, we want to inspect the map between the domain and the codomain. Since we only generate functions from bounded enumeration domains, we only need to show such functions.

instance (Enum e, Bounded e, Show e, SemEq Int) ⇒ Show (Partial e) where
  showPartialFun showPartialFun arbitraryPartialFun arbitraryPartialFun

To show a partially-defined value, we must show that the value is total. For this purpose we use the class SemEq inspired by ChasingBottoms.

class SemEq t where
  (=):=! :: t → Bool
  
  The "bottom-aware" equality test (=):=! uses the standard equality (==) for total values, and deals with ⊥s separately. For example, the instance on Int is given by:

instance SemEq Int where
  x :=!= y = eqPartial (x :=!= y) x y
  eqPartial :: Bool → a → a → Bool
  eqPartial b x y = case (isBottom x, isBottom y) of
    (False, False) → b
    (bx, by) → bx :=!= by

We only compare functions defined on bounded enumerations. We check (extensional) equality by testing that two functions return the same value for all values in their domain. If we know how to compare partially-defined values of type b, and we have a bounded enumeration type e, we can compare functions of type e → b by means of:

Comparing partial values.
instance (Bounded e, Enum e, SemEq b) ⇒ SemEq (e → b) where
f ==! g = eqPartial eqFun f g
where eqFun = all (λx → f x ==! g x)
(⊥ : elements)

We have adapted the TestEqual instance of Endo a to test
the monoid laws also for partial values.

instance (SemEq (Endo a), Show (Partial (Endo a))) ⇒
TestEqual (Endo a) where
testEqPartial l _ = testEqPartial (==! !_)

Where testEqPartial is the (trivially) adapted version of testEq that
also deals with partial values. We can now call
testMonoidEndoPartial = do
quickLawCheckPartial (un :: MonoidLaw1 (Endo Bool))
quickLawCheckPartial (un :: MonoidLaw2 (Endo Bool))
quickLawCheckPartial (un :: MonoidLaw3 (Endo Bool))
to find that these laws are not satisfied anymore. QuickCheck gives
counterexamples for the first and second monoid laws. The coun-
terexamples show that if we instantiate these laws with a ⊥ value,
we get ⊥ at the left-hand side of the first law and const ⊥ at the
right-hand side, and similarly for the second law.

8. State Monad – A Case Study

This section defines the laws for the MonadState class, discusses
various instances of the class, and shows some counterexamples
we found when testing with partial values. To find counterex-
amples for the laws for the implementations, we use the ClassLaws
framework, and follow the steps as outlined in Section 6.

MonadState and its laws. The MonadState class is specified in
Figure 3. The specification does not explicitly mention laws, but
the following combinations of the MonadState operations are often
given as the axioms for MonadState [Gibbons and Hinze 2011].

put s' >>= put s = put s
put s >>= get = put s >>= return s
get >>= put = skip
get >>= (λs → get >>= k s) = get >>= λs → k s s

We could give the GetGet law as
get >>= λs → get >>= λs' → return (s, s') =
get >>= λs → return (s, s)

which would remove the need for the k argument and simplify the
type instance later, but we want to stick to the law exactly as given

By replacing = with ≡ in these equalities, we obtain the default
implementations of these laws in the class MonadStateLaws.

data MSPutPut s m = (m ∷ s → s)
data MSPutGet s m = (m ∷ s → s)
data MSGetPut s m = (m ∷ s → s)

class MonadState s m ⇒ MonadStateLaws s m where
mSPutPut :: Law (MSPutPut s m)
mSPutGet :: Law (MSPutGet s m)
mMSGetPut :: Law (MSGetPut s m)
mMSGetGet :: Law (MSGetGet s m)

We omit the default declarations of these laws for brevity. Each
of the datatypes used to represent a law has instances of the type
families LawArgs and LawBody.

\[\text{class Monad } m \Rightarrow \text{MonadState } s \mid m \rightarrow s \text{ where}\]
\[\text{get } :: m \rightarrow s\]
\[\text{put } :: s \rightarrow m ()\]

Figure 3. The MonadState class in the monad transformer library.

type instance LawArgs (MSPutPut s m) = (s, s)
type instance LawBody (MSPutPut s m) = m ()
type instance LawArgs (MSPutGet s m) = m

type instance LawBody (MSPutGet s m) = m ()
type instance LawArgs (MSGetPut s m) =
s → m a

type instance LawBody (MSGetPut s m) = m a

Finally, we make the MonadState laws instances of the class
LawTest to allow for testing the laws. We only show a single
instance, the other three instances are similar.

instance (MonadStateLaws s m, TestEqual (m ())) ⇒
LawTest (MSPutPut s m) where
lawtest _ = testEqual ◦
(MSPutPut :: Law (MSPutPut s m))

Two instances of MonadState. We use the following datatype
State s a for an instance of MonadState. A value of type State s a is
a function which given a state returns a pair of a value and a new
state.

newtype State s a = S { runS :: s → Pair a s }
data Pair a b = Pair a b

instance MonadState s (State s) where
get = S$ λs → Pair s s
put = S$ const (Pair (λs) s)

If we take (, ) instead of Pair we get the datatype State s a as defined
under Control.MonadState (library versions mtl-1.x). We use an
older version of the standard because from mtl-2.x the state monad
is defined by a monad transformer. Using the more recent version
would complicate the presentation in a way we think unnecessary
for the purpose of this paper. We use Pair instead of (, ) to allow
better control when testing partial values. (It simplifies making one
or both components strict, for example).

Depending on the instances of State s on Monad and Functor we
call the MonadState instance lazy or strict. The lazy version of
the state monad can be found in the module Control.MonadState.Lazy.

instance Monad (State s) where
return a = S$ λs → Pair a s
m >>= k = S$ λs → let Pair a' s' = runS m s
in runS (k a) s'

instance Functor (State s) where
fmap f m = S$ λs → let Pair a' s' = runS m s
in Pair (f a) s'

Control.MonadState.Strict contains the instances resulting in a
strict version of the state monad.

instance Monad (State s) where
return a = S$ λs → Pair a s
m >>= k = S$ λs → case runS m s of
Pair a s' → runS (k a) s'
instance Functor (State s) where
    fmap f m = S $ \lambda s -> case runS m of
        Pair a s' -> Pair (f a) s'

In the rest of this section we will use the lazy instance of state monad, unless mentioned otherwise.

Making State s testable. We want to test, using the ClassLaws framework, whether or not our State s instance of MonadState satisfies the laws. For this purpose, we need to specify

- a State s a instance of the type family Param, providing the extra parameter(s) needed to compare the monadic values,
- a State s a instance of the class TestEqual, with a method testEqual showing how we test equality of the monadic values,
- and State s a instances of the classes Arbitrary and Show.

The parameter of the type used for testing equality on State s values depends on the equality check we use in the TestEqual instance. For general types s we can test functions as shown in Section 2 for State s a values, by requiring an initial s value. But for bounded enumerations (the approach in Section 7) no such argument is needed. In both cases, an s parameter, which is ignored for the second equality, is fine.

type instance Param (State s a) = s

Depending on the kind of equality we want to use on functions, the TestEqual instance of State s can either use the helper function testRunEq or testEq. To test partial values, we use (trivially) adapted versions testRunEqPartial and testEqPartial of these functions. Function testRunEqPartial checks whether running state monadic expressions on some initial state results in the same final state, and testEqPartial checks whether the expressions have the same State s a value.

instance (SemEq a, SemEq s, Show (Partial a), Show (Partial s), Bounded s, Enum s) TestEqual (State s a) where
    testEqual _ _ = testEqPartial (===)

The instances of Bounded and Enum are used for testing equality of arbitrary functions defined on enumerated domain domains. We will refer to this equality as exact equality. We can change the equality check to use runS by changing testEqPartial to testRunEqPartial. We will refer to this equality as exact equality. For run equality, the Bounded and Enum constraints are not needed.

Generating arbitrary, possibly partially defined, State s a values relies on generating arbitrary functions of type s → (a,s) using the approach to generating such functions on bounded enumeration domains introduced in Section 7.

instance ArbitraryPartial a, SemMeet s, Enum s, Bounded s, Eq s, ArbitraryPartial (State s a) where
    arbitraryPartial = genPartial 1 20 (liftM S arbitraryPartial)

We generate partially-defined continuous functions on bounded flat domains with the help of an operator to calculate the meet of two values, for which we need instances of class SemMeet on a and s. Since State s a makes use of the datatype Pair a b, we provide instances of Arbitrary, ArbitraryPartial, Show and SemEq on Pair a b, together with a Show instance for Partial (Pair a b). The definitions are omitted.

In the tests of the laws we will use small enumeration types as arguments to State, both to reduce complexity of counterexamples and to make it possible to show functions that appear in counterexamples. For this purpose we use the types (), Bool, and Ordering, with one, two, and three non-bottom values, respectively. Since the maximum number of different type variables appearing in the laws is three, it suffices to have three different types available for testing. () and Bool already have Arbitrary and CoArbitrary instances. ArbitraryPartial instances for these types are similar to the ArbitraryPartial instances for Int and Char given in Section 7. For Ordering we define similar instances.

To define the instances of Show for the types State s a and Partial (State s a), we use the instance of Show on partial functions given in Section 7

instance (Enum s, Bounded s, Show a, Show s) => Show (State s a) where
    show (State s a) = show (Run s a)

Table 1 summarises the results for the lazy and strict state monad with run = run equality, exact = exact equality, “F” = fails QuickCheck test, “−” = passes 100 QuickCheck tests. The tests were run with ghc version 7.4.2 and the results are same both with and without the flag -fpedantic-bottoms.

<table>
<thead>
<tr>
<th>Law</th>
<th>Lazy</th>
<th>Strict</th>
</tr>
</thead>
<tbody>
<tr>
<td>run</td>
<td>run</td>
<td>run</td>
</tr>
<tr>
<td>MSPutPut</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MSPutGet</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MSPutPut</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MSGetPut</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FunLaw1</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>FunLaw2</td>
<td></td>
<td>F</td>
</tr>
<tr>
<td>MonLaw1</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>MonLaw2</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>MonLaw3</td>
<td></td>
<td>F</td>
</tr>
<tr>
<td>FunMonLaw</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Testing the MonadState laws. To test the MonadState laws for our State s instance of MonadState we create the empty instance:

instance MonadStateLaws s (State s)

We also want to test the Functor, Monad, and FunctionMonad laws for our instance, so we also declare:

instance MonadLaws (State s)
instance FunctorLaws (State s)
instance FunctionMonadLaws (State s)

Examples. To test the laws for our State s instances, we apply quickLawCheck and quickLawCheckPartial to each law, testing with total and partial values, respectively. The inputs to these functions are dummy values of the following types:

- MSPutPutBool (State Bool)
- MSPutGetBool (State Bool)
- MSGetPut (State Bool)
- MSGetGetBool Ordering (State Bool)

and so on for the other laws.

Table 1 summarises the results for the lazy and strict state monads. First, when testing only with total values, both implementa-
tions pass all tests, thus we only show results for partial values. The tests also suggest that the four MonadState laws, the second functor law, the first monad law and the FunMonLaw always hold, even in the presence of partial values. The failing cases in the partial setting are the first functor law and the first and second monad laws.

For partial values we distinguish between “run equality” and “exact equality”. With exact equality functions are compared as laws.

The tests also suggest that the four MonadState laws, the second functor law, the first monad law and the FunMonLaw always hold, even in the presence of partial values. The failing cases in the partial setting are the first functor law and the first and second monad laws.

For partial values we distinguish between “run equality” and “exact equality”. With exact equality functions are compared as laws.

Partial setting are the first functor law and the first and second monad partitions pass all tests, thus we only show results for partial values. The failing cases in the partial setting are the first functor law and the first and second monad laws.

For partial values we distinguish between “run equality” and “exact equality”. With exact equality functions are compared as laws.

The tests also suggest that the four MonadState laws, the second functor law, the first monad law and the FunMonLaw always hold, even in the presence of partial values. The failing cases in the partial setting are the first functor law and the first and second monad laws.

For partial values we distinguish between “run equality” and “exact equality”. With exact equality functions are compared as laws.

Partial setting are the first functor law and the first and second monad partitions pass all tests, thus we only show results for partial values. The failing cases in the partial setting are the first functor law and the first and second monad laws.

For partial values we distinguish between “run equality” and “exact equality”. With exact equality functions are compared as laws.

The tests also suggest that the four MonadState laws, the second functor law, the first monad law and the FunMonLaw always hold, even in the presence of partial values. The failing cases in the partial setting are the first functor law and the first and second monad laws.

For partial values we distinguish between “run equality” and “exact equality”. With exact equality functions are compared as laws.

Partial setting are the first functor law and the first and second monad partitions pass all tests, thus we only show results for partial values. The failing cases in the partial setting are the first functor law and the first and second monad laws.

For partial values we distinguish between “run equality” and “exact equality”. With exact equality functions are compared as laws.

The tests also suggest that the four MonadState laws, the second functor law, the first monad law and the FunMonLaw always hold, even in the presence of partial values. The failing cases in the partial setting are the first functor law and the first and second monad laws.

For partial values we distinguish between “run equality” and “exact equality”. With exact equality functions are compared as laws.

Partial setting are the first functor law and the first and second monad partitions pass all tests, thus we only show results for partial values. The failing cases in the partial setting are the first functor law and the first and second monad laws.

For partial values we distinguish between “run equality” and “exact equality”. With exact equality functions are compared as laws.

The tests also suggest that the four MonadState laws, the second functor law, the first monad law and the FunMonLaw always hold, even in the presence of partial values. The failing cases in the partial setting are the first functor law and the first and second monad laws.

For partial values we distinguish between “run equality” and “exact equality”. With exact equality functions are compared as laws.

Partial setting are the first functor law and the first and second monad partitions pass all tests, thus we only show results for partial values. The failing cases in the partial setting are the first functor law and the first and second monad laws.

For partial values we distinguish between “run equality” and “exact equality”. With exact equality functions are compared as laws.

The tests also suggest that the four MonadState laws, the second functor law, the first monad law and the FunMonLaw always hold, even in the presence of partial values. The failing cases in the partial setting are the first functor law and the first and second monad laws.

For partial values we distinguish between “run equality” and “exact equality”. With exact equality functions are compared as laws.

Partial setting are the first functor law and the first and second monad partitions pass all tests, thus we only show results for partial values. The failing cases in the partial setting are the first functor law and the first and second monad laws.

For partial values we distinguish between “run equality” and “exact equality”. With exact equality functions are compared as laws.

The tests also suggest that the four MonadState laws, the second functor law, the first monad law and the FunMonLaw always hold, even in the presence of partial values. The failing cases in the partial setting are the first functor law and the first and second monad laws.

For partial values we distinguish between “run equality” and “exact equality”. With exact equality functions are compared as laws.

Partial setting are the first functor law and the first and second monad partitions pass all tests, thus we only show results for partial values. The failing cases in the partial setting are the first functor law and the first and second monad laws.

For partial values we distinguish between “run equality” and “exact equality”. With exact equality functions are compared as laws.

The tests also suggest that the four MonadState laws, the second functor law, the first monad law and the FunMonLaw always hold, even in the presence of partial values. The failing cases in the partial setting are the first functor law and the first and second monad laws.

For partial values we distinguish between “run equality” and “exact equality”. With exact equality functions are compared as laws.

Partial setting are the first functor law and the first and second monad partitions pass all tests, thus we only show results for partial values. The failing cases in the partial setting are the first functor law and the first and second monad laws.

For partial values we distinguish between “run equality” and “exact equality”. With exact equality functions are compared as laws.

The tests also suggest that the four MonadState laws, the second functor law, the first monad law and the FunMonLaw always hold, even in the presence of partial values. The failing cases in the partial setting are the first functor law and the first and second monad laws.

For partial values we distinguish between “run equality” and “exact equality”. With exact equality functions are compared as laws.

Partial setting are the first functor law and the first and second monad partitions pass all tests, thus we only show results for partial values. The failing cases in the partial setting are the first functor law and the first and second monad laws.

For partial values we distinguish between “run equality” and “exact equality”. With exact equality functions are compared as laws.

The tests also suggest that the four MonadState laws, the second functor law, the first monad law and the FunMonLaw always hold, even in the presence of partial values. The failing cases in the partial setting are the first functor law and the first and second monad laws.

For partial values we distinguish between “run equality” and “exact equality”. With exact equality functions are compared as laws.

Partial setting are the first functor law and the first and second monad partitions pass all tests, thus we only show results for partial values. The failing cases in the partial setting are the first functor law and the first and second monad laws.

For partial values we distinguish between “run equality” and “exact equality”. With exact equality functions are compared as laws.

The tests also suggest that the four MonadState laws, the second functor law, the first monad law and the FunMonLaw always hold, even in the presence of partial values. The failing cases in the partial setting are the first functor law and the first and second monad laws.

For partial values we distinguish between “run equality” and “exact equality”. With exact equality functions are compared as laws.

Partial setting are the first functor law and the first and second monad partitions pass all tests, thus we only show results for partial values. The failing cases in the partial setting are the first functor law and the first and second monad laws.

For partial values we distinguish between “run equality” and “exact equality”. With exact equality functions are compared as laws.

The tests also suggest that the four MonadState laws, the second functor law, the first monad law and the FunMonLaw always hold, even in the presence of partial values. The failing cases in the partial setting are the first functor law and the first and second monad laws.

For partial values we distinguish between “run equality” and “exact equality”. With exact equality functions are compared as laws.

Partial setting are the first functor law and the first and second monad partitions pass all tests, thus we only show results for partial values. The failing cases in the partial setting are the first functor law and the first and second monad laws.

For partial values we distinguish between “run equality” and “exact equality”. With exact equality functions are compared as laws.

The tests also suggest that the four MonadState laws, the second functor law, the first monad law and the FunMonLaw always hold, even in the presence of partial values. The failing cases in the partial setting are the first functor law and the first and second monad laws.

For partial values we distinguish between “run equality” and “exact equality”. With exact equality functions are compared as laws.

Partial setting are the first functor law and the first and second monad partitions pass all tests, thus we only show results for partial values. The failing cases in the partial setting are the first functor law and the first and second monad laws.

For partial values we distinguish between “run equality” and “exact equality”. With exact equality functions are compared as laws.

The tests also suggest that the four MonadState laws, the second functor law, the first monad law and the FunMonLaw always hold, even in the presence of partial values. The failing cases in the partial setting are the first functor law and the first and second monad laws.

For partial values we distinguish between “run equality” and “exact equality”. With exact equality functions are compared as laws.

Partial setting are the first functor law and the first and second monad partitions pass all tests, thus we only show results for partial values. The failing cases in the partial setting are the first functor law and the first and second monad laws.

For partial values we distinguish between “run equality” and “exact equality”. With exact equality functions are compared as laws.

The tests also suggest that the four MonadState laws, the second functor law, the first monad law and the FunMonLaw always hold, even in the presence of partial values. The failing cases in the partial setting are the first functor law and the first and second monad laws.

For partial values we distinguish between “run equality” and “exact equality”. With exact equality functions are compared as laws.

Partial setting are the first functor law and the first and second monad partitions pass all tests, thus we only show results for partial values. The failing cases in the partial setting are the first functor law and the first and second monad laws.

For partial values we distinguish between “run equality” and “exact equality”. With exact equality functions are compared as laws.

The tests also suggest that the four MonadState laws, the second functor law, the first monad law and the FunMonLaw always hold, even in the presence of partial values. The failing cases in the partial setting are the first functor law and the first and second monad laws.

For partial values we distinguish between “run equality” and “exact equality”. With exact equality functions are compared as laws.

Partial setting are the first functor law and the first and second monad partitions pass all tests, thus we only show results for partial values. The failing cases in the partial setting are the first functor law and the first and second monad laws.
When we change to the strict version of the state monad we have fewer failing behaviours. Most failing behaviours that disappeared are due to $\mathit{Pair} :: \bot \leftrightarrow \bot :: \mathit{Pair\ a\ b}$. But the issue remains when we use exact equality.

With $(\bot :: \mathit{State\ Bool\ Ordering})$, the strict version results in

\[
\begin{align*}
\mathit{const} (\bot :: \mathit{Pair\ Ordering\ Bool}) \\
\not\equiv \bot :: \mathit{State\ Bool\ Ordering}
\end{align*}
\]

We have tried a few other variations of state monad implementations, without finding a formulation that satisfies all the laws at the same time. We believe that there is in fact no implementation of a state monad in Haskell which satisfies all of the laws. It is future work to prove that this is the case (or show a counterexample). The fact that state monads seem to work out fine anyway indicates that the laws are most likely “wrong”, at least for partial values. Exploring alternative formulations of the laws is also future work, but can be helped by the $\text{ClassLaws}$ framework. Starting from the paper on “Fast and loose reasoning” [Danielsson et al. 2006] it should be possible to implement a library of combinators for “selectively ignoring” bottoms in parts of the laws.

9. Conclusions and related work

We have introduced a framework for testing class laws. Using a single $\text{quickLawCheck}$ function, we can test any class law on any instance of a class with laws. To make this work, we need to specify laws in a particular format, and we need to provide instances for generating, comparing, and showing values of the class instance that we want to test. The format for specifying laws allows us to provide further evidence for a law, so that we can check the steps in a ‘proof’ for a law. Furthermore, we introduce a function $\text{quickLawCheckPartial}$, which tests laws in the same format with potentially partially-defined values. To make this work we use a type modifier $\mathit{Partial}$ and the $\mathit{ChasingBottoms}$ library, and we introduce classes for generating and comparing potentially partial values. We use the framework and function $\text{quickLawCheckPartial}$ to check whether or not the standard implementations of the state monad satisfy the expected laws. It turns out that none of the implementations satisfies the expected laws if we also test with partially-defined values.

$\text{ClassLaws}$ is a light-weight framework, in which a user has to add a couple of declarations per law, and a couple of declarations per datatype on which laws are to be tested, to test class laws. A few of these declarations could be derived automatically, such as the instances of $\mathit{LawTest}$, and the definition of the law in terms of the law default. Deriving these declarations automatically is hardly worth the effort: it saves only a few, trivial, lines, and would make the framework less light-weight.

There is little related work on checking type class laws. In his blog post ‘QuickChecking Type Class Laws’, Taysom [2011] shows how to QuickCheck the laws for semirings. He more or less describes the first steps we take in Section 2 for QuickChecking laws, and does not deal with testing laws for types like $\mathit{Endo\ a}$ or providing evidence, nor with testing with partially-defined values. Elliott [2012] has developed a package that wraps up the expected properties associated with various standard type classes as QuickCheck properties. He does not deal with testing laws for types like $\mathit{Endo\ a}$ or providing evidence, nor with testing with partially-defined values. On the other hand, Checkers makes it easy to check all laws of a class using a single declaration, something we deferred to future work. We used QuickCheck and ChasingBottoms for all testing purposes, but we could have used LazySmallCheck [Runciman et al. 2008] instead. Although LazySmallCheck generates partially-defined values, it does not generate functions, so also when using LazySmallCheck we would have had to implement our own generators for partially-defined functions.

Besides the class laws given in this paper, we also implemented the laws for the Haskell standard classes $\mathit{Num}$, $\mathit{Integral}$, and $\mathit{Show}$. It is future work to express laws for all classes specified in the Haskell base library. Other future work consists of making the framework more convenient to use, by providing functionality for testing all laws of a class by means of a single declaration, and by allowing $\eta$-reduction when specifying laws. Finally, we do not only want to test laws and their evidence, but also to verify laws using a proof checker like the Haskell Inductive Prover http://github.com/danr/hip by Dan Rosén.

Acknowledgements. This research has been partially funded by the Global Systems Dynamics and Policy (GSDP) project (FP7, ICT-2009.8.0 FET Open), and the Swedish Research Council. Cláudio Amaral is funded by FCT grant SFRH/BD/63571/2009 and partially funded by LIACC through Programa de Financiamento Plurianual, Fundação para a Ciência e Tecnologia, Portugal. Nick Smallbone suggested the $\text{TestablePartial}$ and $\text{ArbitraryPartial}$ classes. The Haskell symposium reviewers suggested many improvements to the paper.

References


