Testing Type Class Laws

Johan Jeuring

Patrik Jansson

Cláudio Amaral

Technical Report UU-CS-2012-008

Department of Information and Computing Sciences
Utrecht University, Utrecht, The Netherlands
www.cs.uu.nl

ISSN: 0924-3275

Department of Information and Computing Sciences
Utrecht University
P.O. Box 80.089
3508 TB Utrecht
The Netherlands

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.

Haskell'12, September 13, 2012, Copenhagen, Denmark.

class *Functor f* **where**
  $fmap :: (a \to b) \to f\ a \to f\ b$

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

$fmap\ id \quad == id$
$fmap\ (f \circ g) == fmap\ f \circ fmap\ g$

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

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

Instances of *Monad* should satisfy the following laws:

$return\ a \gg\!= k \quad == k\ a$
$m \quad \gg\!= return \quad == m$
$m \quad \gg\!= (\lambda x \to k\ x \gg\!= h) == (m \gg\!= k) \gg\!= h$

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

$fmap\ f\ xs == xs \gg\!= return \circ f$

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')

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:

$$\begin{aligned} &mempty &&:: a \\ &mappend &&:: a \to a \to a \end{aligned}$$

together with a method $mconcat :: [a] \to a$ which we won't use in this paper. We will write infix $\mathbin{+\!\!+}$ for *mappend*. Implementations of these methods in an instance of *Monoid* should satisfy the following three laws:

$$\begin{aligned} &mempty \mathbin{+\!\!+} m = m \\ &m \mathbin{+\!\!+} mempty = m \\ &l \mathbin{+\!\!+} (m \mathbin{+\!\!+} r) = (l \mathbin{+\!\!+} m) \mathbin{+\!\!+} r \end{aligned}$$

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

$$\begin{aligned} &monoidLaw_1\ m &&= mempty \mathbin{+\!\!+} m == m \\ &monoidLaw_2\ m &&= m \mathbin{+\!\!+} mempty == m \\ &monoidLaw_3\ l\ m\ r &&= l \mathbin{+\!\!+} (m \mathbin{+\!\!+} r) == (l \mathbin{+\!\!+} m) \mathbin{+\!\!+} r \end{aligned}$$

and can be tested as follows for the *Monoid* instance for lists

$$\begin{aligned} &main = \mathbf{do} \\ &\quad quickCheck\ (monoidLaw_1 :: [Int] \to Bool) \\ &\quad quickCheck\ (monoidLaw_2 :: [Int] \to Bool) \\ &\quad quickCheck\ (monoidLaw_3 \\ &\qquad\qquad :: [Int] \to [Int] \to [Int] \to Bool) \end{aligned}$$

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*:

$$\mathbf{newtype}\ Endo\ a = Endo\ \{appEndo :: a \to a\}$$

satisfies the *Monoid* laws? Adding the line

$$quickCheck\ (monoidLaw_1 :: Endo\ Int \to Bool)$$

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:

$$\mathbf{newtype}\ State\ s\ a = State\ \{runState :: s \to (a, s)\}$$

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 $a \to a \to Bool$. 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*:

$$\mathbf{type\ family}\ Param\ b$$

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

$$\begin{aligned} &\mathbf{type\ instance}\ Param\ [a] &&= () \\ &\mathbf{type\ instance}\ Param\ (Endo\ a) &&= a \\ &\mathbf{type\ instance}\ Param\ (State\ s\ a) &&= s \end{aligned}$$

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*

$$\begin{aligned} &\mathbf{class}\ TestEqual\ a\ \mathbf{where} \\ &\quad testEqual :: a \to a \to Param\ a \to Bool \end{aligned}$$

together with the instances:

$$\begin{aligned} &\mathbf{instance}\ Eq\ a \Rightarrow TestEqual\ [a]\ \mathbf{where} \\ &\quad testEqual\ l\ r\ \_ = l == r \\ &\mathbf{instance}\ Eq\ a \Rightarrow TestEqual\ (Endo\ a)\ \mathbf{where} \\ &\quad testEqual\ l\ r\ p = appEndo\ l\ p == appEndo\ r\ p \\ &\mathbf{instance}\ (Eq\ a, Eq\ s) \Rightarrow TestEqual\ (State\ s\ a)\ \mathbf{where} \\ &\quad testEqual\ l\ r\ s = runState\ l\ s == runState\ r\ s \end{aligned}$$

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".

```
type Equal a = (a, a)
infixr 0 =.=
(=.=) = (,)
monoidLaw1 m     = mempty ⊕ m =.= m
monoidLaw2 m     = m ⊕ mempty =.= m
monoidLaw3 l m r = l ⊕ (m ⊕ r)  =.= (l ⊕ m) ⊕ r
```

We can use this new formulation of the laws to test whether or not the *Monoid*-instance of *Endo a* satisfies the *Monoid*-laws.

```
tooVerboseVersionOfMain = do
  quickCheck
    (uncurry testEqual ∘ monoidLaw1
    :: Endo Int → Param (Endo Int) → Bool)
  quickCheck
    (uncurry testEqual ∘ monoidLaw2
    :: Endo Int → Param (Endo Int) → Bool)
  quickCheck
    ((λl m r → uncurry testEqual (monoidLaw3 l m r))
    :: Endo Int → Endo Int → Endo Int →
      Param (Endo Int) → Bool)
```

***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 :: MonoidLaw1 (Endo Int))
  quickLawCheck (un :: MonoidLaw2 (Endo Int))
  quickLawCheck (un :: MonoidLaw3 (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
  monoidLaw1 :: m → Equal m
  monoidLaw2 :: m → Equal m
  monoidLaw3 :: m → m → m → Equal m
  monoidLaw1 m     = mempty ⊕ m =.= m
  monoidLaw2 m     = m ⊕ mempty =.= m
  monoidLaw3 l m r = l ⊕ (m ⊕ r)  =.= (l ⊕ m) ⊕ r
```

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 MonoidLaw1 m
data MonoidLaw2 m
data MonoidLaw3 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 (MonoidLaw1 m) = m
type instance LawArgs (MonoidLaw2 m) = m
type instance LawArgs (MonoidLaw3 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 (MonoidLaw1 m) = m
type instance LawBody (MonoidLaw2 m) = m
type instance LawBody (MonoidLaw3 m) = m
```

The instances for functor laws, which we will give later, show more variety. Using these newly introduced type families, we can reformulate the type of the monoid laws as follows:

```
type Law t = LawArgs t → Equal (LawBody t)
class Monoid m ⇒ MonoidLaws m where
  monoidLaw1 :: Law (MonoidLaw1 m)
  monoidLaw2 :: Law (MonoidLaw2 m)
  monoidLaw3 :: Law (MonoidLaw3 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
          → Param (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 (MonoidLaw1 m) where
  lawtest _ = testEqual ∘ monoidLaw1
instance (MonoidLaws m, TestEqual m) ⇒
         LawTest (MonoidLaw2 m) where
  lawtest _ = testEqual ∘ monoidLaw2
instance (MonoidLaws m, TestEqual m) ⇒
         LawTest (MonoidLaw3 m) where
  lawtest _ = testEqual ∘ monoidLaw3
```

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 :: MonoidLaw1 (Endo Int))
  quickLawCheck (un :: MonoidLaw2 (Endo Int))
  quickLawCheck (un :: MonoidLaw3 (Endo Int))
```

to test the monoid laws for *Endo a*. As expected, QuickCheck does not find any counterexamples. In Section 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 $id \circ \bot = const\ \bot \not\equiv \bot$ and $\bot \circ id = const\ \bot \not\equiv \bot$, 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*.

```
defaultFunLaw1 x       = fmap id x =.= id x
defaultFunLaw2 (f,g,x) =
  (fmap f ∘ fmap g) x =.= fmap (f ∘ 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 $f\ a$ for some $f :: * \to *$ and some a. We define the instance of *LawArgs* for the datatype $FunLaw_1$ corresponding to this law as follows:

```
data FunLaw1 a (f :: * → *)
type instance LawArgs (FunLaw1 a f) = f a
```

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

```
data FunLaw2 a b c (f :: * → *)
type instance LawArgs (FunLaw2 a b c f) =
  (b → c, a → 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 (FunLaw1 a     f) = f a
type instance LawBody (FunLaw2 a b c f) = f c
```

Now we define the class *FunctorLaws*:

```
class Functor f ⇒ FunctorLaws f where
  funLaw1 :: Law (FunLaw1 a     f)
  funLaw2 :: Law (FunLaw2 a b c f)
  funLaw1 = defaultFunLaw1
  funLaw2 = defaultFunLaw2
```

We make these datatypes instances of *LawTest* as follows:

```
instance (FunctorLaws f, TestEqual (f a)) ⇒
         LawTest (FunLaw1 a f) where
  lawtest _ = testEqual ∘ funLaw1
instance (FunctorLaws f, TestEqual (f c)) ⇒
         LawTest (FunLaw2 a b c f) where
  lawtest _ = testEqual ∘ funLaw2
```

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 $lhs = rhs$ in a scientific paper, we typically write

```
  lhs
= {good reason}
  lhs'
...
  rhs'
= {another good reason}
  rhs
```

In this section we show how we express this proof as a list of expressions $[lhs, lhs', ..., rhs', rhs]$, 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 [Jansson 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)
```

on which we want to have a function *fmap* that not only applies a function to all elements in the list, but also reverses the list at the same time.

```
instance Functor List where
  fmap f Nil         = Nil
  fmap f (Cons x xs) = snoc (f x) (fmap f xs)
```

Here *snoc* takes an element and a list, and adds the element to the end of the list:

```
snoc y Nil         = Cons y Nil
snoc y (Cons x xs) = Cons x (snoc y xs)
```

We omit the more efficient implementation that uses an accumulating parameter. Suppose we also want to use functionality from the *Monad* and *Applicative* classes on our lists. For the *Monad* instance of our lists we take the predefined standard instance. An instance of *Applicative* requires a proper instance of *Functor*. To make sure that our list instance of *Functor* satisfies the *Functor* laws, we use our framework to test class laws.

```
import Control.Monad.Laws

instance FunctorLaws       List
instance MonadLaws         List
instance FunctorMonadLaws List
```

With the three instance declarations we declare that our instances should satisfy the laws of the *Functor* and *Monad* (represented by $MonLaw_1$, $MonLaw_2$, and $MonLaw_3$, see Figure 2) class, respectively, and that it should also satisfy the law that requires an instance of both *Functor* and *Monad* (represented by *FunMonLaw*, see the last law in Figure 2): $fmap\ f\ xs == xs >\!\!>\!\!= return \circ f$. We use *quickLawCheck* and *quickFLawCheck* to test the laws:

```
main = do
  quickLawCheck  (un :: FunLaw1 Char List)
  quickFLawCheck (un :: FunLaw2 Int Char Bool List)
  quickFLawCheck (un :: MonLaw1 Char Int List)
  quickLawCheck  (un :: MonLaw2 Int List)
  quickFLawCheck (un :: MonLaw3 Int Bool Char List)
  quickFLawCheck (un :: FunMonLaw Char Int List)
```

If we run *main*, we find that the *Functor* laws are not satisfied for our instance. For these two laws we get the counterexamples:

```
(Cons 0 (Cons 2 Nil), Cons 2 (Cons 0 Nil))
(Cons (-4) (Cons (-3) Nil), Cons (-3) (Cons (-4) Nil))
```

respectively. Clearly, lists of length two are sufficient to show that *fmap* changes the order of the elements. The *Monad* laws do not lead to any counterexamples, but for the *FunMonLaw* we get the counterexample:

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

Suppose we are (erroneously) convinced that our implementation of lists satisfies the first functor law. To find out where our reasoning fails, we provide a detailed sequence of steps which we think proves the law. The first functor law serves as an example:

```
instance FunctorLaws List where
  funLaw1 xs = addSteps (defaultFunLaw1 xs)
                        (case xs of
                           Nil          -> nilCase
                           xs@(Cons _ _) -> consCase xs)
nilCase = [fmap id Nil
          ,    -- definition of fmap on Nil
           Nil
          ]

consCase (Cons y ys) =
  [fmap id (Cons y ys)
  ,    -- definition of fmap for Cons
   snoc (id y) (fmap id ys)
  ,    -- definition of id
   snoc y (fmap id ys)
  ,    -- induction hypothesis
   snoc y ys
  ,    -- definition of id
   id (Cons y ys)
  ]
```

In the *FunctorLaws List* instance, we specify that we think that the left-hand side of the first functor law ($defaultFunLaw_1$) 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' =.= into a list of values instead of a pair of values:

```
type Equal    = []
type Theorem = Equal
(=.=)         :: a -> a -> Theorem a
(=.=) a1 a2 = [a1,a2]
addSteps ::   Theorem a -> Equal a -> Equal a
addSteps [lhs,rhs] steps = lhs : steps ++ [rhs]
addSteps _         _     = error "addSteps ..."
```

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

```
(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.

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 *List*s. We derive the *Show* instance for *List*s, and define the following instance of *TestEqual*:

```
instance (Eq a, Show a) => TestEqual (List a) where
  testEqual p _ = testEq (==) p
```

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 :: Show a ⇒
          (a → a → Bool) → Equal a → Property
testEq (==) steps =
  whenFail (print  $failingPair (==) steps)
  $property $liftBool $pairwiseEq (==) steps

pairwiseEq :: (a → a → Bool) → Equal a → Bool
pairwiseEq (==) (x : y : ys) = x == y ∧ pairwiseEq (==) (y : ys)
pairwiseEq (==) _            = True

type Pos = Int

failingPair :: (a → a → Bool) → [a] → (Pos, a, a)
failingPair = failingPair' 1

failingPair' pos (==) (x : y : ys) =
  if   ¬ (x == y)
  then (pos, x, y)
  else failingPair' (1 + pos) (==) (y : ys)
```

The functions *property* and *liftBool* are QuickCheck utilities which turn a boolean into a property. Function *whenFail* shows its first argument whenever the test of the property fails.

As explained in Section 2 types that abstract over functions, such as the the types *State* and *Endo* are harder to test. For these types we define:

```
instance (Eq a, Show a, Eq s, Show s) ⇒
         TestEqual (State s a) where
  testEqual = testRunEq runState (==)

instance (Eq a, Show a) ⇒ TestEqual (Endo a) where
  testEqual = testRunEq appEndo (==)
```

Here we use the function *testRunEq*, which takes a run function, a list of values to be tested for equality, and a start value for the run function, and returns a property, which tests consecutive elements in the list to be equal by means of the function *pairwiseEq*.

```
testRunEq ::    Show r ⇒
                (t → p → r)
             → (r → r → Bool)
             → Equal t → p → Property
testRunEq run (==) steps p =
  testEq (==) (map (`run` p) steps)
```

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, CoArbitrary a) ⇒
         Arbitrary (Endo a) where
  arbitrary = liftM Endo arbitrary
```

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 l_1, l_2, ... for a class C in our framework, we need to specify:

- The laws themselves as functions $defaultl_1$, $defaultl_2$, ...
- Datatypes L_1, L_2, ..., which take the type arguments used in the types of the laws as argument, and have no right-hand sides.
- Type family instances for the datatypes L_1, L_2, ..., 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 l_1, l_2, ..., 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 $l_1 = defaultl_1$ etc.
- Instances of the class *LawTest* for the datatypes L_1, L_2, ..., in which *lawtest* is defined by $lawtest\ \_ = testEqual \circ l_1$, 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*.

It follows that we have to perform a little amount of work per law and per datatype, to get functionality for testing laws for free. Per class for which we want to test laws, we need to specify one declaration, per law seven declarations, and per datatype on which we want to test the laws of a class five declarations. Twelve of these thirteen declarations need only be given once, and can be reused for testing laws on different datatypes, or testing different laws on a datatype.

We released version 0.3.0.1 of our code in June 2012 on Hackage under the name ClassLaws[2].

7. Testing Laws with Bottoms

The previous sections show how to test class laws in the standard QuickCheck environment, in which random generated values are total. Testing properties with total values is often sufficient, but sometimes we also want to know if a law holds in the presence of partial values. For functions that make essential use of laziness, it is necessary to also test with partial values. Every datatype has undefined or partial values, and we should adapt random generation for all datatypes to also test properties for partially-defined values. QuickCheck comes with predefined random generators in instances of *Arbitrary* for many types, and it is hard to use QuickCheck without importing these predefined random generators. The standard approach to change random generation is to introduce a *type modifier* and specify random generation for the type modifier. To randomly generate partially-defined values, we introduce a type modifier *Partial*

```
newtype Partial a = Partial {unPartial :: a}
```

We use this modifier to generate and test laws for partially-defined values. We show how to generate random values that are possibly partial, how to compare partial values for equality, and how to support partial predicates (QuickCheck properties) with *ClassLaws*.

What do we need to change? Suppose we want to test the first monoid law on *Endo Int* for partially-defined values. This law is tested in the *ClassLaws* framework using the expression

[2] `http://hackage.haskell.org/package/ClassLaws`

quickLawCheck $(un :: MonoidLaw_1\ (Endo\ Int))$. 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 *testEqual* 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* $(==)$, 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

$$quickLawCheckPartial = quickCheck \circ Partial \circ lawtest$$

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 $a \to prop$ 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
  propertyPartial :: prop → 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 ⇒
         Testable (Partial prop)
  where 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 $f =$ *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
  propertyPartial = property
```

The instance of *TestablePartial* on function types is more interesting:

```
instance (ArbitraryPartial a
         ,Show (Partial a)
         ,TestablePartial prop
         ) ⇒ TestablePartial (a → prop) where
  propertyPartial f = forAllShrink arb shr prop where
    arb          = fmap Partial arbitraryPartial
    shr (Partial x) = map Partial (shrinkPartial x)
    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
  arbitraryPartial :: Gen a
  shrinkPartial    :: a → [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][3]. 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 $isBottom :: a \to Bool$ 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 ($x \,/\backslash!\, y$) 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*:

```
tLess = ⊥ <=! (const ⊥ :: Bool → Bool)
tMeet = (⊥,'b','c') /\! ('a',⊥,'c') ==! (⊥,⊥,'c')
```

To work around some problems with ChasingBottoms we use our own classes *SemEq*, *SemOrd* and *SemMeet* 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

[3] See `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 $\perp$-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 $\perp$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 $e \to s$ 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 $\perp$ 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 :: ∀ e a.
  (Enum e, Bounded e, SemEq e, SemMeet a) ⇒
  Gen a → Gen (e → a)
arbitraryPartialFun ag = do
  funtab ← forM (⊥ : enumElems :: [e]) (\_ → ag)
  genPartial 1 6 (return (table2fun funtab))
```

Function *table2fun* returns a monotonous function by ensuring that the image of bottom is the meet of all possible images.

```
type FunTab e s = [s]

table2fun :: (Enum e, Bounded e, SemEq a, SemMeet a) ⇒
             FunTab e a → (e → a)
table2fun tab@(_ : tottab) = fun
  where meet = foldr1 (/\!) 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 ++ "_"
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 (Partial e), Show (Partial b)
         ) ⇒ Show (Partial (e → b)) where
  show (Partial f) = showPartialFun (⊥ : elements) f
```

Function *showPartialFun* shows for each value in the domain, the value in the codomain to which it is bound.

```
showPartialFun p f =
  if   isBottom f
  then "<_bot_fun_>"
  else "<(" ++
       (concat $ intersperse "; "
          [ show (Partial x) ++ "->" ++
            show (Partial (f x))
          | x ← p])
       ++ ")>"
```

Comparing partial values. Function *lawtest* uses the function *testEqual* to test whether or not the left-hand side and right-hand side of a law are equal. Now that laws are tested with partially-defined values, equality needs to deal with partial values as well. For this purpose we use the class *SemEq* inspired by *ChasingBottoms*.

```
class SemEq t where
  (==!) :: t → t → Bool
```

The "bottom-aware" equality test (==!) uses the standard equality (==) for total values, and deals with $\perp$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 $e \to b$ by means of:

```
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
  testEqual l _ = testEqPartial (==!) l
```

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 counterexamples show that if we instantiate these laws with a $\bot$ value, we get $\bot$ at the left-hand side of the first law and *const* $\bot$ 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 counterexamples 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 in the Gibbons and Hinze [2011] reference.

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

```
data MSPutPut s   (m :: * → *)
data MSPutGet s   (m :: * → *)
data MSGetPut     (m :: * → *)
data MSGetGet s a (m :: * → *)

class MonadState s m ⇒ MonadStateLaws s m where
  mSPutPut :: Law (MSPutPut s   m)
  mSPutGet :: Law (MSPutGet s   m)
  mSGetPut :: Law (MSGetPut     m)
  mSGetGet :: Law (MSGetGet s a 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*.

```
class Monad m ⇒ MonadState s m|m → s where
  get :: m s
  put :: s → 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)  = s
type instance LawBody  (MSPutGet s m)  = m s
type instance LawArgs  (MSGetPut m)    = ()
type instance LawBody  (MSGetPut m)    = m ()
type instance LawArgs  (MSGetGet s a m) =
                         s → s → m a
type instance LawBody  (MSGetGet s a 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 () s)
```

If we take $(,)$ instead of *Pair* we get the datatype *State s a* as defined under *Control.Monad.State* (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.Monad.State.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.Monad.State.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 $ λs → case runS m s 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 l _ = testEqPartial (==!) l
```

The instances of *Bounded* and *Enum* are used for testing equality of arbitrary functions defined on bounded enumeration 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 *run* 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 $s \to (a,s)$ using the approach to generating such functions on bounded enumeration domains introduced in Section 7.

```
instance (ArbitraryPartial a, SemMeet a
         ,ArbitraryPartial s, 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.

	Lazy		Strict	
Law	*run*	*exact*	*run*	*exact*
MSPutPut	.	.	.	.
MSPutGet	.	.	.	.
MSGetPut	.	.	.	.
MSGetGet	.	.	.	.
$FunLaw_1$	F	F	.	F
$FunLaw_2$	.	.	.	.
$MonLaw_1$	.	F	.	F
$MonLaw_2$	F	F	.	F
$MonLaw_3$	.	.	.	.
FunMonLaw	.	.	.	.

Table 1. Summary of 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`.

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 (S f) = "(S " ++ show f ++ ")"
instance (Enum s, Bounded s
         ,Show (Partial a), Show (Partial s)) ⇒
         Show (Partial (State s a)) where
  show (Partial s) | isBottom s = "_|_St_"
  show (Partial (S f)) =
    "(S " ++ show (Partial f) ++ ")"
```

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 *FunctorMonad* laws for our instance, so we also declare:

```
instance MonadLaws           (State s)
instance FunctorLaws         (State s)
instance FunctorMonadLaws    (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:

```
MSPutPut Bool (State Bool)
MSPutGet Bool (State Bool)
MSGetPut (State Bool)
MSGetGet Bool 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 third 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 values in the semantic domain, thus $\bot \not\equiv const\ \bot$. Run equality of f and g is checked after applying *runS* to both sides. The first functor law and the first and second monad law are not satisfied in many cases.

The first functor law fails for the value $\bot$ of type *State Bool* (). In the left hand side of the law we have

$$
\begin{array}{ll}
 & fmap\ id\ (\bot :: State\ Bool\ ()) \\
\equiv & \text{-- definition of } fmap \\
 & S\ \$\ \lambda s \to \textbf{let}\ Pair\ a\ s' = runS\ (\bot :: State\ Bool\ ())\ s \\
 & \qquad \textbf{in}\ \ Pair\ (id\ a)\ s' \\
\equiv & \text{-- apply } runS \\
 & S\ \$\ \lambda s \to \textbf{let}\ Pair\ a\ s' = \bot :: Pair\ ()\ Bool \\
 & \qquad \textbf{in}\ \ Pair\ a\ s' \\
\equiv & \text{-- } \textbf{let}\text{-reduction} \\
 & S\ \$\ \lambda s \to Pair\ (\bot :: ())\ (\bot :: Bool)
\end{array}
$$

which differs from the right hand side

$$
\begin{array}{ll}
 & id\ (\bot :: State\ Bool\ ()) \\
\equiv & \text{-- apply } id \\
 & \bot :: State\ Bool\ () \\
\equiv & \text{-- } \textbf{newtype} \text{ constructor } S \text{ is strict} \\
 & S\ (\bot :: Bool \to Pair\ ()\ Bool)
\end{array}
$$

It is important to notice that patterns in **let**-expressions are lazy. These terms are different with respect to exact equality, and their final states, with *True* as the first state parameter, are also different.

$$
\begin{array}{ll}
 & (\bot :: Bool \to Pair\ ()\ Bool)\ True \\
\equiv & \text{-- apply } \bot \\
 & \bot :: Pair\ ()\ Bool \\
\not\equiv & \\
 & Pair\ (\bot :: ())\ (\bot :: Bool) \\
\equiv & \text{-- beta-reduction} \\
 & (\lambda s \to Pair\ (\bot :: ())\ (\bot :: Bool))\ True
\end{array}
$$

This is an interesting counterexample because it works for both kinds of equality in the lazy implementation.

Another case where the first functor law fails is in the strict version with exact equality.

$$
\begin{array}{ll}
 & fmap\ id\ (\bot :: State\ Bool\ ()) \\
\equiv & \text{-- definition of } fmap \\
 & S\ \$\ \lambda s \to \textbf{case}\ runS\ (\bot :: State\ Bool\ ())\ s\ \textbf{of} \\
 & \qquad Pair\ a\ s' \to Pair\ (id\ a)\ s' \\
\equiv & \text{-- apply } runS \text{ and } id \\
 & S\ \$\ \lambda s \to \textbf{case}\ (\bot :: Pair\ ()\ Bool)\ \textbf{of} \\
 & \qquad Pair\ a\ s' \to Pair\ a\ s' \\
\equiv & \text{-- } \textbf{case}\text{-reduction} \\
 & S\ \$\ \lambda s \to (\bot :: Pair\ ()\ Bool) \\
\not\equiv & \\
 & \bot :: State\ Bool\ () \\
\equiv & \text{-- apply } id \\
 & id\ (\bot :: State\ Bool\ ())
\end{array}
$$

The first monad law only fails tests that compare monadic terms. This suggests the law only has problems with different function terms that map their arguments to equal images. This pattern can be explained by the objects $\bot$ and $const\ \bot$ of the function space $a \to b$, that map any $(x :: a)$ into $(\bot :: b)$. The counterexamples support this claim. For the strict version (with $k = const\ \bot$):

$$
\begin{array}{ll}
 & return\ False \gg\!= k \\
\equiv & \text{-- definition of } (\gg\!=) \\
 & S\ \$\ \lambda s \to \textbf{case}\ runS\ (return\ False)\ s\ \textbf{of} \\
 & \qquad Pair\ a\ s' \to runS\ (k\ a)\ s' \\
\equiv & \text{-- definition of } return \\
 & S\ \$\ \lambda s \to \textbf{case}\ runS\ (S\ \$\ \lambda s \to Pair\ False\ s)\ s\ \textbf{of} \\
 & \qquad Pair\ a\ s' \to runS\ (k\ a)\ s' \\
\equiv & \text{-- apply } runS \\
 & S\ \$\ \lambda s \to \textbf{case}\ (\lambda s \to Pair\ False\ s)\ s\ \textbf{of} \\
 & \qquad Pair\ a\ s' \to runS\ (k\ a)\ s' \\
\equiv & \text{-- beta-reduction} \\
 & S\ \$\ \lambda s \to \textbf{case}\ Pair\ False\ s\ \textbf{of} \\
 & \qquad Pair\ a\ s' \to runS\ (k\ a)\ s' \\
\equiv & \text{-- } \textbf{case}\text{-reduction} \\
 & S\ \$\ \lambda s \to runS\ (k\ False)\ s \\
\equiv & \text{-- apply } k \\
 & S\ \$\ \lambda s \to runS\ (\bot :: State\ Bool\ ())\ s \\
\equiv & \text{-- apply } runS \\
 & S\ \$\ \lambda s \to \bot :: Pair\ ()\ Bool \\
\not\equiv & \\
 & \bot :: State\ Bool\ () \\
\equiv & \text{-- apply } k \\
 & k\ False
\end{array}
$$

And for the lazy version (with $k = \bot :: Bool \to State\ Bool\ ()$):

$$
\begin{array}{ll}
 & return\ False \gg\!= k \\
\equiv & \text{-- definition of } (\gg\!=) \\
 & S\ \$\ \lambda s \to \textbf{let}\ Pair\ a\ s' = runS\ (return\ False)\ s \\
 & \qquad \textbf{in}\ \ runS\ (k\ a)\ s' \\
\equiv & \text{-- definition of } return \\
 & S\ \$\ \lambda s \to \textbf{let}\ Pair\ a\ s' = runS\ (S\ \$\ \lambda s \to Pair\ False\ s)\ s \\
 & \qquad \textbf{in}\ \ runS\ (k\ a)\ s' \\
\equiv & \text{-- apply } runS \\
 & S\ \$\ \lambda s \to \textbf{let}\ Pair\ a\ s' = (\lambda s \to Pair\ False\ s)\ s \\
 & \qquad \textbf{in}\ \ runS\ (k\ a)\ s' \\
\equiv & \text{-- beta-reduction} \\
 & S\ \$\ \lambda s \to \textbf{let}\ Pair\ a\ s' = Pair\ False\ s \\
 & \qquad \textbf{in}\ \ runS\ (k\ a)\ s' \\
\equiv & \text{-- } \textbf{let}\text{-reduction} \\
 & S\ \$\ \lambda s \to runS\ (k\ False)\ s \\
\equiv & \text{-- apply } k \\
 & S\ \$\ \lambda s \to runS\ (\bot :: State\ Bool\ ())\ s \\
\equiv & \text{-- apply runS} \\
 & S\ \$\ \lambda s \to \bot :: Pair\ ()\ Bool \\
\not\equiv & \\
 & \bot :: State\ Bool\ () \\
\equiv & \text{-- apply } k \\
 & k\ False
\end{array}
$$

The second monad law fails for cases similar to the first functor law. The lazy version of the state monad does not satisfy the second monad law either. From $\bot :: State\ Bool\ Ordering$ the law evaluates to

$$
\begin{array}{ll}
 & Pair\ (\bot :: Ordering)\ (\bot :: Bool) \\
\not\equiv & \bot :: Pair\ Ordering\ Bool
\end{array}
$$

when we run this monadic computation (in any first state) and to

$$
\begin{array}{ll}
 & const\ (Pair\ (\bot :: Ordering)\ (\bot :: Bool)) \\
\not\equiv & \bot :: State\ Bool\ Ordering
\end{array}
$$

when we check exact equality.

When we change to the strict version of the state monad we have fewer failing behaviours. Most failing behaviours that disappeared are due to $\mathit{Pair}\ \bot\ \bot \not\equiv \bot :: \mathit{Pair\ a\ b}$. But the issue remains when we use exact equality.

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

$$\mathit{const}\ (\bot :: \mathit{Pair\ Ordering\ Bool}) \not\equiv \bot :: \mathit{State\ Bool\ Ordering}$$

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 *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 *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 *quickLawCheckPartial*, which tests laws in the same format with potentially partially-defined values. To make this work we use a type modifier *Partial* and the *ChasingBottoms* library, and we introduce classes for generating and comparing potentially partial values. We use the framework and function *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.

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 *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 *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 *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 Lazy Smallcheck [Runciman et al. 2008] instead. Although Lazy SmallCheck generates partially-defined values, it does not generate functions, so also when using Lazy Smallcheck 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 *Num*, *Integral*, and *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 η-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 `https://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/65371/2009 and partially funded by LIACC through Programa de Financiamento Plurianual, Fundação para a Ciência e Tecnologia, Portugal. Nick Smallbone suggested the *TestablePartial* and *ArbitraryPartial* classes. The Haskell symposium reviewers suggested many improvements to the paper.

References

J.-P. Bernardy, P. Jansson, and K. Claessen. Testing polymorphic properties. In A. Gordon, editor, *ESOP'10*, volume 6012 of *Lecture Notes in Computer Science*, pages 125–144. Springer, 2010.

M. M. T. Chakravarty, G. Keller, S. Peyton Jones, and S. Marlow. Associated types with class. In *POPL'05*, pages 1–13. ACM, 2005.

K. Claessen and J. Hughes. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *ICFP'00*, pages 286–279. ACM, 2000.

N. A. Danielsson and P. Jansson. Chasing bottoms: A case study in program verification in the presence of partial and infinite values. In D. Kozen and C. Shankland, editors, *MPC'04*, volume 3125 of *Lecture Notes in Computer Science*, pages 85–109. Springer, 2004.

N. A. Danielsson, J. Hughes, P. Jansson, and J. Gibbons. Fast and loose reasoning is morally correct. In *POPL'06*, pages 206–217. ACM Press, 2006.

C. Elliott. Checkers. A Haskell package available on Hackage, 2012. URL `http://hackage.haskell.org/package/checkers-0.2.9`.

J. Gibbons and R. Hinze. Just do it: simple monadic equational reasoning. In *ICFP'11*, pages 2–14. ACM, 2011.

P. Jansson and J. Jeuring. Polytypic data conversion programs. *Science of Computer Programming*, 43(1):35–75, 2002.

S. Marlow, editor. *Haskell 2010 Language Report*, 2010. `http://www.haskell.org/onlinereport/haskell2010/`.

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.

W. Taysom. Quickchecking type class laws. Blog post, 2011. URL `http://www.cubiclemuses.com/cm/articles/2011/07/14/quickchecking-type-class-laws/`.