Proving Type Class Laws for Haskell

Andreas Arvidsson, Moa Johansson, and Robin Touche

Department of Computer Science and Engineering, Chalmers University of Technology
andreas.arvidson@gmail.com, moa.johansson@chalmers.se,
robin_touche@hotmail.com[**]

Abstract. Type classes in Haskell are used to implement ad-hoc polymorphism, i.e. a way to ensure both to the programmer and the compiler that a set of functions are defined for a specific data type. All instances of such type classes are expected to behave in a certain way and satisfy laws associated with the respective class. These are however typically just stated in comments and as such, there is no real way to enforce that they hold. In this paper we describe an extension to the inductive theorem prover HipSpec, which allows the user to write down type class laws which are then automatically instantiated and sent to the theorem prover when declaring a new instance of a type class.

1 Introduction

Type classes in Haskell are used to implement ad-hoc polymorphism, or overloading [13]. They allow programmers to define functions which behave differently depending on the types of their arguments. A type-class declares a set of (abstract) functions and their types, and any datatype declared an instance of the class has to provide implementations for those functions. Below is the Haskell library type class `Monoid`, which declares the three abstract functions `mempty`, `mappend` and `mconcat`:

```
class Monoid a where
        mempty :: a
        mappend :: a -> a -> a
        mconcat :: [a] -> a
```

Usually, when we define a type-class we have some expectation on what properties instances of this type class should satisfy when implementing the functions specified. These are captured by *type class laws*. For monoids, any instance is supposed to satisfy the laws below:

[**] Andreas Arvidsson and Robin Touche are MSc students at Chalmers. This extended abstract describes work from their MSc thesis [1], supervised by Moa Johansson.

```
-- Left identity
mappend mempty x = x

-- Right identity
mappend x mempty = x

-- Associativity of mappend
mappend x (mappend y z) = mappend (mappend x y) z

-- Specification of mconcat
mconcat = foldr mappend mempty
```

The last law is in fact a specification for the *default implementation* of the `mconcat` function, which is commonly used (unless the user wants to provide their own, optimised, implementation). The last law then becomes a trivial identity.

Example 1. The most obvious instance of `Monoid` is probably `Lists`, but we may for instance also declare the natural numbers to be monoids, with `+` corresponding to `mappend`:

```
data Nat = Zero | Succ Nat

-- Natural number addition
(+) :: Nat -> Nat -> Nat
Zero + a = a
(Succ a) + b = Succ (a + b)

-- Natural number multiplication
(*) :: Nat -> Nat -> Nat
Zero * m = Zero
(Succ n) * m = m + (n * m)

-- Make Nat an instance of the Monoid type class
instance Monoid Nat where
        mempty = Zero
        mappend = (+)
```

We could also have declared `Nat` a monoid in a different manner, with with `*` corresponding to `mappend`:

```
instance Monoid Nat where
        mempty = Succ Zero
        mappend = (*)
```

As these instance of the `Monoid` class are quite simple. We might, by just looking at these instances, convince ourselves that they behave in accordance with the type class laws for monoids, and settle for that. But what if we had a more complicated instance or had made a mistake? Unfortunately, in Haskell, type class laws are typically only stated in comments or documentation, if at all, and there is no support for checking that an instance of a type class actually behave in accordance with the laws.

To address this problem, Jeuring et al. developed a framework for expressing and testing type class laws in Haskell using the QuickCheck tool [8, 3]. As further work, they identify the need to also provide stronger guarantees by also *proving* type class laws using an automated theorem prover. However, the type class laws typically present in Haskell programs often involve recursive functions and datatypes, which means that we might need induction to prove them. While there has been much success using SMT-solvers and first-order theorem provers for reasoning about programs, these provers typically do not support induction. Some of the difficulties with inductive proofs is that they often require auxiliary lemmas, which themselves require induction. A system built to handle these kind of problems is HipSpec [4], a state-of-the-art automated inductive theorem for Haskell. Our contribution is an extension to HipSpec which allows us to:

- Write down type class laws in Haskell as *abstract properties* (§3.1), including support for types with class constraints.
- Automatically instantiate these abstract properties when new instances of a type class is declared, and translate them into a intermediate language called TIP [9], suitable for passing on to automated theorem provers (§3.2).
- Send the generated conjectures to an automated theorem prover for proof, or output the result as a TIP-problem file (§4).

This allows us to state the type class laws abstractly only once, and automatically infer and prove the concrete properties that any new instance need to satisfy to comply with the type class laws.

2 Background: HipSpec and TIP

HipSpec allows the user to write down properties to prove in the Haskell source code, in a similar manner to how Haskell programmers routinely write QuickCheck properties. HipSpec supports a subset of the Haskell language, with the caveat that functions are assumed to be terminating and values assumed to be finite. HipSpec can, if required, apply induction to the conjectures it is given, and then send the resulting proof obligations to an external theorem prover, such as Z3, E or Waldmeister [6, 11, 7]. The power of HipSpec comes from its *theory exploration* phase: when given a conjecture to prove, HipSpec first use its subsystem QuickSpec [5], which *explores* the functions occurring in the problem by automatically suggesting a set of potentially useful basic lemmas, which HipSpec then proves by induction. These can then be used in the proof of the main conjecture. However, as theory exploration happens first, HipSpec sometimes also proves some extra properties, perhaps not strictly needed. We consider this a small prize for the extra power theory exploration provides.

Example 2. As a small example, consider asking HipSpec to prove that our `natAdd` function from the introduction indeed is commutative. The Haskell file is annotated with the property we wish to prove:

```
prop_add_commute x y = x + y == y + x
```

Calling HipSpec on this file instantly produces the output:

```
Proved:
    (m + n) == (n + m)
    (m + (n + o)) == (n + (m + o))
    prop_add_commute  x + y == y + x
```

Notice that HipSpec has printed out two properties it discovered itself during its theory exploration phase (one of which is identical to the property we stated!). These were discovered automatically during HipSpec's theory exploration phase, and proved by induction. Proving `prop_add_commute` is then trivial.

The TIP language. The latest version of HipSpec also supports the TIP-language [9], as an intermediate language into which Haskell functions, types and properties are translated in order to facilitate passing them on to various external theorem provers. TIP is a general format for expressing problems for inductive theorem provers based on SMT-LIB [2], extended with support for datatypes and recursive functions. We will not give an in depth description of the syntax of TIP here, save for a small example of the property we saw above, namely that addition is commutative:

```
(assert-not (forall ((x Nat)) (= (plus x y) (plus y x))))
```

The keyword `assert-not` is used to tell the prover which properties to attempt to prove (this is inherited from SMT-LIB, where proofs are by refutation). Similarly, the keyword `assert` is used to tell the provers which properties are to be treated as axioms.

The ambition of TIP is to provide a shared format for many inductive theorem provers, thereby making it easier to share benchmarks and compare different provers. There exists a growing number of benchmarks for inductive theorem provers written in the TIP language, and a suite of tools for translating TIP into various common input formats for automated theorem provers such as SMT-LIB and TPTP [10, 12]. We will make use of the TIP language when instantiating type-class laws as described in the next section. The advantage is that this allows us to, in the future, experiment also with other inductive theorem proving backends, not only HipSpec. Furthermore, any inductive problems which the theorem prover cannot prove, can be added to the TIP benchmark suite to provide developers with additional challenge problems.

3 Type Class Laws in HipSpec

In the previous section, we showed how HipSpec has traditionally been used, with the user annotating the source file with properties to prove. In a sense, HipSpec has been used mainly as a theorem prover, which just happened to take (a subset of) Haskell as input. In this work, we want extend HipSpec to be able to handle more of the Haskell language, and thereby take a step towards making it a more useful tool also for programmers, and not just as a theorem prover with Haskell as input.

3.1 Expressing Type Class Laws

Type class law are expressed almost as normal HipSpec properties, using much of the same syntax as in Example 2. The difference is that the type class laws use abstract functions, i.e. functions which need to be supplied concrete definitions for the class instances. This is reflected in the type signature of the law, which contains type variables with class constraints. These abstract laws will give rise to multiple different versions specialised for each individual instance of that class.

Example 3. Consider one of the laws for monoids which we have encountered before.

```
plusAssoc :: Monoid a => a -> a -> a -> Equality a
mappend x (mappend y z) === mappend (mappend x y) z
```

Notice that the type variable `a` is required to be a `Monoid` in the type signature. `Equality` is HipSpec's type for equational properties to prove, and is represented using `===`.

3.2 Instantiating Laws

HipSpec does currently only supports fully polymorphic type variables, i.e. type variables which have no constraints on the values they can assume. Since type class laws contains type variables with such constraints (e.g. the constraint that `a` must be a monoid in Example 3), they must be converted into specialised versions. This is rather straight-forward, and done by manipulating the GHC Core expressions resulting from compilation. For each type variable constrained by a type class, we first simply replace with the type of all possible instances of that type class defined in the current file (if there are any).

The type class law `plusAssoc` from Example 3 will for the `Nat` instance we saw before becomes: [1]

```
plusAssoc1 :: Nat -> Nat -> Nat -> Equality a
mappend x (mappend y z) === mappend (mappend x y) z
```

Notice that the type constraint has disappeared.

GHC Core implements type classes by dictionary passing, i.e. when an overloaded function is called, a lookup in dictionary provides the appropriate concrete instance. For increased readability of the generated TIP-code, we inline dictionary lookups and rename them with more informative new names.

```
plusAssoc2 :: Nat -> Nat -> Nat -> Equality a
MonoidNatmappend x (MonoidNatmappend y z) ===
 MonoidNatmappend (MonoidNatmappend x y) z
```

The new function `MonoidNatmappend` is included in the resulting TIP file, defined as the corresponding function from Example 1 (i.e. `+` or `*`, depending on which `Monoid` instance we are considering).

3.3 Superclasses and Instances with Constraints

Sometimes, it is appropriate to include some additional information in the generated TIP problem file, other than the function definitions and instances of the type class laws.

[1] For readability, we give these in a Haskell like syntax, rather than in GHC Core

Superclass Laws. Haskell type classes may be defined as subclasses of already existing type classes. For instance, the type class `Ord`, used for totally ordered datatypes, is an extension of the type class for equality, `Eq`:

```
class Eq a => Ord a where
...
```

This means that instances of `Ord` must also be instances of `Eq`, otherwise the compiler will complain. When generating instances of type class laws, we have therefore made the design decision to assume that all the superclass laws hold when attempting to prove the corresponding subclass laws. For example, this means assuming that the laws for `Eq` holds while proving the laws for a new instance of `Ord`. In other words, we assume that the `Eq` laws were already proved before, when `a` was declared an instance of `Eq`. This allows us to handle our proofs modularly, treating each type class in isolation and not having to worry about long chains of dependencies between type classes. The generated TIP file which we pass to the theorem prover must therefore include the super class laws as axioms holding for the type `a`. To achieve this, the type variable `a` is substituted by an arbitrary dummy datatype, upon which we may include such assertions.

Constrained Class Instances. Another similar scenario is when the class instance itself has a constrained type variable in its declaration. This is often the case for polymorphic data types, for example `Maybe a` if declaring it an instance of `Monoid` like in the example below:

```
instance Monoid a => Monoid (Maybe a) where
          ...
```

In this case we need to take a little more care when instantiating the type class laws; we cannot simply replace the type variable `a` in the `Monoid a` constraint by all its instances, as this includes `Maybe a` itself, as well as `Maybe(Maybe(...))` and so on. Clearly this is not what's intended. Instead, we will interpret this to mean that: *Assuming* that the type `a` is a `Monoid`, satisfying the associated type class laws, then prove that `Maybe a` does too. As for superclasses, we substitute `a` for a new concrete dummy type, about which we can assert the laws for monoids, and use them when proving the laws for `Maybe`.

4 Proving Type Class Laws

Once the TIP files has been generated we pipe them through to HipSpec (or, if available, some other inductive theorem prover) for proof. We have focused the experiments given here to a number of type classes where many of the laws are a little bit harder to prove than for basic type classes (such as `Eq`) and may require induction.

The experiments presented here use a development version of the HipSpec system, which includes improvements such as native support for the TIP-format as input, and options for more informative proof output which explains the steps of the proofs. However, unlike the previous version of HipSpec (described in [4]), the new version does not, at the time of writing, fully support higher-order functions. We have therefore not yet been able to prove all laws of common type classes such as `Monoid`, `Functor` and `Monad`. Addressing this issue is ongoing work. Our tool can however output problem files in TIP-format also for type-classes containing higher-order functions.

4.1 Experimental Results

The timings in the experiments were obtained on a regular laptop computer. The source files for all the experiments are available online from:
`https://github.com/chip2n/hipspec-typeclasses`.

Semiring. Our first experiment comes from algebra, and is about semirings. A semiring is a set coupled with operators for addition (`+`) and multiplication (`*`), as well as identity values for both of them. As a Haskell typeclass, this can be expressed as:

```
class Semiring a where
    zero :: a
    one  :: a
    (+)  :: a -> a -> a
    (*)  :: a -> a -> a
```

The following eight laws are supposed to hold for semirings:

1. $(a + b) + c = a + (b + c)$
2. $0 + a = a + 0 = a$
3. $a + b = b + a$
4. $(a * b) * c = a * (b * c)$
5. $1 * a = a * 1 = a$
6. $a * (b + c) = (a * b) + (a * c)$
7. $(a + b) * c = (a * c) + (b * c)$
8. $0 * a = a * 0 = 0$

We tested the following four instances of semirings:

Nat: Natural numbers with addition and multiplication and identities 0 and 1.

Bool (1): Disjunction for addition and conjunction for multiplication with identities False and True

Bool (2): Conjunction for addition and disjunction for multiplication with identities True and False

Matrix2: A 2-by-2 matrix with matrix addition and multiplication and identities empty matrix and identity matrix respectively.

Results: Semiring

Instance	**Law**								**Total time**
	1	2	3	4	5	6	7	8	
Nat	✓	✓	✓	✓	✓	✓	✓	✓	28.2 s
Bool (1)	✓	✓	✓	✓	✓	✓	✓	✓	2.9 s
Bool (2)	✓	✓	✓	✓	✓	✓	✓	✓	2.9 s
Matrix2	✓	✓	✓	✓	✓	✗	✗	✓	29.3 s

Most of these proofs are rather easy for HipSpec, except the distributivity properties for matrices. We conjecture that this is due to the fact that HipSpec has not been told anything about auxiliary functions for summations of matrix elements row/column-wise, which is required here. Notice that in the case of the natural numbers and matrices, HipSpec spends more time doing theory exploration, and inventing lemmas. For the booleans, the proofs are straightforward and do not even need induction.

CommutativeSemiring. To demonstrate a typeclass which depend on a superclass, we also included a commutative semiring in our evaluation.

```
class Semiring a => CommutativeSemiring a
```

As the name suggests, a commutative semiring is a semiring with one additional law stating that `*` is commutative:

1. $a * b = b * a$

We tested the same instances as for semirings:

Results: CommutativeSemiring

Instance	**Law**	**Total time**
	1	
Nat	✓	21.0 s
Bool (1)	✓	2.8 s
Bool (2)	✓	2.5 s
Matrix2	!	11.9 s

As expected, natural numbers and booleans can easily be shown to also be commutative semirings. Note that the matrix instance fails as expected; matrix multiplication is not commutative.

Reversible. The next type class characterise data structures that can be reversed:

```
class Reversible a where
    reverse :: a -> a
```

It has one law, stating that `reverse` is idempotent:

1. $\mathit{reverse}\ (\mathit{reverse}\ \mathit{xs}) = \mathit{xs}$

We tested the following three instances:

List (rev): Naive recursive list reverse.
List (qrev): List reverse using accumulator.
Tree: A binary tree with a mirror operation, flipping the tree left to right.

Results: Reversible

Instance	**Law**	**Total time**
	1	
List (*rev*)	✓	5.9 s
List (*qrev*)	✓	5.7 s
Tree	✓	0.9 s

All these proofs require induction, and the ones about lists also need auxiliary lemmas which HipSpec must discover and prove first. However, as there is only very few functions and laws present (unlike for semirings), this is rather fast.

Monoid. We have already encountered the Monoid type class as a running example.

1. $mappend\ mempty\ x = x$
2. $mappend\ x\ mempty = x$
3. $mappend\ x\ (mappend\ y\ z) = mappend\ (mappend\ x\ y)\ z$
4. $mconcat\ xss = foldr\ mappend\ mempty\ xss$

Note that the 4th law is higher-order and, as mentioned above, the version of HipSpec we are using does not yet support higher-order functions. We can therefore only give the results for the remaining three laws, and do so for the datatype instances we have seen before. The `Matrix2` instance has the additional constraint of requiring its elements to also be monoids, and similarly, `Maybe a` has a constraint requiring `a` to be a monoid .

Results: Monoid

Instance	**Law**			**Total time**
	1	2	3	
Nat (*add*)	✓	✓	✓	4.3 s
Nat (*mul*)	✓	✓	✓	27.3 s
Matrix2 (*mul*)	✓	✓	✓	2.3 s
List	✓	✓	✓	1.5 s
Maybe	✓	✓	✓	2.1 s

Laws 1 and 2 for monoids are rather trivial. The only instance needing slightly longer is when *mappend* is instantiated to natural number multiplication, in which law 3 needs some lemmas to be discovered and proved.

4.2 Summary of Results

The proof obligations resulting from the type classes above are mainly within the comfort zone of an inductive theorem prover such as HipSpec. Usually, the prover just needs a few seconds to complete the proofs, unless it has discovered many candidate lemmas during the theory exploration phase. This is the downside of using HipSpec: it sometimes eagerly spends some time proving lemmas that turn out not to be necessary for the final proof.

It is disappointing that we have not yet been able to automatically prove any higher-order laws, but hope that it is merely a matter of time until support has been added also to the new version of HipSpec. We could have opted for using an old version of HipSpec, but it does not support TIP as an input language, and we would have had to instead produce modified copies of the original Haskell source files to give to the prover. This would likely have meant less automation of the whole proving pipeline. We also think it is valuable to output the instantiated laws and proof obligations in a format like TIP which can more easily be shared between different theorem proving systems. This means that we are not actually bound to just a single theorem prover (HipSpec), but can quite easily connect our work also to other provers, should a better one become available.

5 Conclusion

This work demonstrates how automated theorem proving and functional programming can be integrated to provide benefit to programmers. We showed how type class laws can be expressed as abstract properties, then instantiated and proved when we declare a new instance of a type class. Moving type class laws from informal descriptions in the documentation to something that is actually enforced would both help programmers to a shared understanding of the purpose of the type class as well as helping them implement instances correctly.

Many Haskell programmers are today routinely writing QuickCheck properties which are automatically tested on many randomly generated values. In the future, we envisage programmers writing similar properties and laws, and using a tool like HipSpec to not only test, but also *prove* their programs correct, with little more overhead than using QuickCheck. We believe our work is at least a small step on the way towards this ambitious goal.

References

1. Andreas Arvidsson and Robin Touche. Proving type class laws in Haskell. Master's thesis, Chalmers University of Technology, 2016.
2. Clark Barrett, Aaron Stump, and Cesare Tinelli. The SMT-LIB standard – version 2.0. In *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories*, July 2010. Edinburgh, Scotland.
3. Koen Claessen and John Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proceedings of ICFP*, pages 268–279, 2000.

4. Koen Claessen, Moa Johansson, Dan Rosén, and Nicholas Smallbone. Automating inductive proofs using theory exploration. In *Proceedings of the Conference on Auomtated Deduction (CADE)*, volume 7898 of *LNCS*, pages 392–406. Springer, 2013.
5. Koen Claessen, Nicholas Smallbone, and John Hughes. QuickSpec: Guessing formal specifications using testing. In *Proceedings of TAP*, pages 6–21, 2010.
6. Leonardo De Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In *Proceedings of TACAS*, TACAS'08/ETAPS'08, pages 337–340. Springer-Verlag, 2008.
7. Thomas Hillenbrand, Arnim Buch, Roland Vogt, and Bernd Löchner. Waldmeister - high-performance equational deduction. *Journal of Automated Reasoning*, 18(2):265–270, 1997.
8. Johan Jeuring, Patrik Jansson, and Claudio Amaral. Testing type class laws. In *2012 ACM SIGPLAN Haskell Symposium, Copenhagen*, pages 49–60, 2012.
9. Moa Johansson, Dan Rosén, Nicholas Smallbone, and Koen Claessen. TIP: Tons of inductive problems. In *Proceedings of the Conference on Intelligent Computer Mathematics*, 2015.
10. Dan Rosén and Nicholas Smallbone. TIP: Tools for inductive provers. In *Logic for Programming, Artificial Intelligence, and Reasoning: 20th International Conference, LPAR-20 2015, Suva, Fiji, November 24-28, 2015, Proceedings*, pages 219–232. Springer Berlin Heidelberg, 2015.
11. Stephan Schulz. System Description: E 1.8. In Ken McMillan, Aart Middeldorp, and Andrei Voronkov, editors, *Proc. of the 19th LPAR, Stellenbosch*, volume 8312 of *LNCS*. Springer, 2013.
12. G. Sutcliffe. The TPTP Problem Library and Associated Infrastructure: The FOF and CNF Parts, v3.5.0. *Journal of Automated Reasoning*, 43(4):337–362, 2009.
13. P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '89, pages 60–76, New York, NY, USA, 1989. ACM.