Translating Type Class Laws from AGDA2HS to QuickCheck

Giovanni Grandi[1]
Supervisor(s): Jesper Cockx[1]**, Nathaniel Burke**[1]
[1]EEMCS, Delft University of Technology, The Netherlands

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering
June 21 2026

Name of the student: Giovanni Grandi
Final project course: CSE3000 Research Project
Thesis committee: Jesper Cockx, Nathaniel Burke, Annibale Panichella

Abstract

This research aims to explore using property based testing to find counterexamples for proving statements about programs in AGDA2HS. I have created an extension to AGDA2HS to translate type class laws into QuickCheck properties and evaluated on real-world Haskell code to evaluate its effectiveness. I found that having relatively cheap to run tests compared to formalizing helped save time ensuring definitions were correct before formalizing statements.

1 Introduction

When programming there are some cases when the code you write needs to verify that certain properties hold. For example, if you are writing a library that then gets used by other people. In this case users of your library expect that it should work as advertised. One way you could do this is proving the correctness of your implementation.

In Haskell, this is doable on paper with equational reasoning with the method described by Hutton [1, chapter 16]. With this approach, you replace functions with their definitions and previously proven lemmas about them. This however is error-prone. When writing out equational reasoning steps, it is easy to make transcription mistakes by writing either definitions or lemmas wrong. Proving on paper also means that when the program is updated it is easy to forget to update the proofs that go along with it.

Luckily, there is a better solution by using interactive theorem provers such as Agda [2]. In Agda, you can write the program, the properties of the program and the proof in the same language. Agda then determines the correctness of proofs when running the type checking.

To use this capability in Haskell, there is a tool called AGDA2HS presented by Cockx et al. [3]. It is a backend for Agda that produces readable Haskell code as its output. This allows combining the verification capabilities of Agda while still producing code that can be used and audited by Haskell developers.

One of the most essential features of Haskell are its type classes. These are the mechanism by which Haskell implements ad-hoc polymorphism. This means that type classes provide overloaded functions for different types. One example of an operation that is useful to implement is a way to print a type as a `String`. In Haskell, this is provided by the `Show` type class. A different example of a useful type class is the `Ord` type class which provides the functions to compare two values such as the `<=` function.

Type classes have an implicit series of laws that programmers expect to hold [4]. For example, by implementing the `Ord` type class you should ensure that if `x <= y` and `y <= z` then `x <= z`. These laws are often used to make valid instances of polymorphic methods for all types. If there is a type `a` which implements `Ord` without following this law, a binary search tree with values of type `a` would not work since in some cases the left branch would contain values bigger than the right branch.

One use case for AGDA2HS is proving the correctness of type class laws. However this can be a difficult process depending on the definition of the type class. In my experience, writing the proof for the laws of a given type class can take twice to three times as much time as writing the instances. What would be ideal is if before proving there was a way to find counterexamples of the laws based on the definitions to prevent spending time trying to prove false statements.

Property based testing with QuickCheck [5] can be used for generating these counterexamples. QuickCheck allows testing a property of a Haskell program by generating a series of random inputs and checking that the property holds for each one.

This paper explores how can type class laws in Agda be translated to QuickCheck properties in Haskell with AGDA2HS. It presents the following contributions:

- I present an implementation of a translation from type class laws in AGDA2HS to QuickCheck properties described in Section 3 using the `Dec` type to check whether the proofs could be inhabited.
- An evaluation of the translation on 3 examples of instances of type classes from Haskell packages and the Haskell `base` package in Section 4. These were picked to showcase interesting examples of the algorithm succeeding and its limitations.

I then discuss some background for understanding the work in Section 2, the related work in Section 5 and the conclusions in Section 7.

2 Background

2.1 AGDA2HS

AGDA2HS is a custom backend for Agda that allows compiling a subset of Agda to readable Haskell code. It allows writing code very similarly to writing in Haskell but also supporting proving the correctness. Proving functional correctness from Haskell directly is only possible on paper which is both error prone and brittle. With agda2hs both the program and proof are written in the same language and the proofs are checked automatically [3]. The generated code is then reasonable Haskell code that can be reasoned about by developers who do not know Agda.

AGDA2HS comes with a `base` library that mirrors the `base` package in Haskell. This has a reimplementation of many functions and type classes that enable writing Agda code with exactly the same functions as Haskell. These have been reimplemented in Agda so that they can be used in proofs of theorems in AGDA2HS.

2.2 Predicates in Agda

In Agda, a statement is "true" if it is inhabited and "false" if it is empty. For example, the unit type `⊤` is inhabited since it has one constructor that can always be used and the empty type `⊥` is empty since it has no constructors:

```
record ⊤ : Type where
  instance constructor tt

data ⊥ : Type where
```

This same idea can be used to create other types to check if a statement is inhabited. For example, I can create a type for checking whether a list is non-empty or that two values are equal:

```
data NonEmpty {a : Type} : List a → Type where
  instance itsNonEmpty : ∀ {x xs} → NonEmpty (x ∷ xs)

data _≡_ {a : Type} (x : a) : a → Type where
  instance refl : x ≡ x
```

The `_≡_` type (also known as the identity type) allows you to express equality since when if two values `x` and `y` are equal you can construct an instance of it using the `refl` constructor and if they are not equal it acts as the `⊥` with no constructor.

There are also more complicated facts that can be modeled. For example, I can model the statement that a list is ascending with the following type:

```
data IsAscending {a : Type} ⦃ iOrdA : Ord a ⦄ : List a → Type where
  instance
    Empty : IsAscending []
    OneElem : {x : a} →  IsAscending (x ∷ [])
    ManyElem : {x y : a} {xs : List a}
        → ⦃ IsTrue (x <= y) ⦄
        → ⦃ IsAscending (y ∷ xs) ⦄
        → IsAscending (x ∷ y ∷ xs)
```

Either the list has to be an empty list, in which case it is ascending. Alternatively it can be a list with a single element which makes it ascending. Finally, it can contain two or more elements in which case the first element needs to be less than or equal to the second element and the rest of the list needs to also be in ascending order.

For all of these types, you might have noticed the `instance` keyword appear. This is to make the constructors available to a feature of Agda called instance resolution. Instance resolution allows for the automatic search of arguments to your function [6]. An example where this is useful when trying to define a function with a precondition since it allows Agda to automatically find a proof of the precondition.

With predicates like this, you can sometimes create a decision procedure which enables finding whether a type is inhabited. To enable this, AGDA2HS has the `Dec` type which is encoded isomorphically[1] to the following:

```
record Dec {a} (A : Type a) : Type a where
  constructor _⟨⟩
  field
    inhabited  : Bool
    @0 ⦃ witness ⦄ : if inhabited then A else (A → ⊥)
```

This says that each `Dec` instance is equipped with two things. The first is a `Bool` indicating whether it is inhabited or not. The second is a witness where if it is inhabited it must be an element of the type and if it is not inhabited it must be a proof that creating an element of the type would lead to a contradiction. The second argument is also marked as erased with `@0` this guarantees that it is available for type checking but will not get compiled into the final Haskell output.

This enables writing a decision procedure for the `NonEmpty` predicate:

```
decNonEmpty : {xs : List a} → Dec (NonEmpty xs)
decNonEmpty {xs = x ∷ xs} = True ⟨⟩
decNonEmpty {xs = []} = (False ⟨⟩) ⦃ λ { () } ⦄
```

This says that if `xs` is a non empty list, the statement is true and instance search can fill in the proof using the `itsNonEmpty` case. If `xs` is the empty list, since there is no constructor, we must provide a proof that it would lead to a contradiction. In this case, the proof is specified

[1]The proof of which is in Appendix A

by `λ { () }` since Agda can figure out that there are no constructors matching an empty list. A procedure can also be constructed for `IsAscending`, the code for it is in Appendix A.

2.3 Type classes

Type classes are the way Haskell implements ad-hoc polymorphism. For example, In Haskell there is a type class which holds the `_==_` operation for checking if two things are equal. Implicitly, this type class has the law that if `a == b` then `a` and `b` are the same object for all intents and purposes. In Agda, type classes are implemented as Agda records [6]. Records in Agda are types for grouping values together and giving them names [7]. These can then be placed inside of `instance` blocks to create an instance of them for a given type. There is also an implicit form of argument like `{{ y }}` which specifies that there must be an instance of type `y`. See Listing 1 for how they are used from Agda.

```
-- Pair type
record Pair (A B : Type) : Type where
  field
    fst : A
    snd : B

-- Eq Type class
record Eq (a : Type) : Type where
  infix 4 _==_
  field
    _==_ : a → a → Bool

open Eq {{...}} public
```

```
instance
  open Pair
  -- Create an instance of Eq for pairs
  -- stating that two pairs are equal
  -- iff they have the same first and
  -- second element
  iEqPair : {{ Eq a }} → {{ Eq b }}
    → Eq (Pair a b)
  iEqPair ._==_ x y =
    fst x == fst y && snd x == snd y
```

Listing 1: Example of defining a record type, type class and creating an instance of a type class in Agda

In the base library of AGDA2HS alongside the `base` type classes of Haskell, there are `IsLawful` type classes which hold laws. An example is the `IsLawfulEq` type class which enriches the `Eq` type class with an `isEquality` field which states that `x ≡ y` if and only if `x == y`. These theorems can then be used to prove further theorems that use these data types.

2.4 Property Based Testing

Property based testing is a form of testing where you create a property and then check it against random inputs to see whether it holds. When a counterexample is found, it will then try shrinking it to be as simple as possible [8]. To show this, imagine the following example in Haskell[2]:

```
data Nat = Zero | Suc Nat deriving (Eq)
```

Here, Nat is the unary natural numbers data class. This shows. With this representation `1` can be modeled as `Suc Zero`, `2` can be modeled as `Suc (Suc Zero)` and so forth. A useful function on the natural numbers is addition:

```
add :: Nat -> Nat -> Nat
add Zero y = y
add (Suc x) y = add x y
```

However, this definition is incorrect. A property addition should satisfy is commutativity. We can use QuickCheck to validate if it holds. This can be modeled in QuickCheck as:

[2]For the full code see Appendix C

```
prop_add_commutative (x :: Nat) (y :: Nat) = add x y === add y x
```

When running this with QuickCheck, we see a generated counterexample:

```
ghci> quickCheck prop_add_commutative
*** Failed! Falsified (after 2 tests):
0
1
```

By then looking through the code with these values in minds, when the `1` is on the left-hand side it gets simplified down to `0` and then `add 0 y` returns `0`. To fix this function, we add a `Suc` to the right-hand side:

```
add :: Nat -> Nat -> Nat
add Zero y = y
add (Suc x) y = Suc (add x y)
```

```
ghci> quickCheck prop_add_commutative
+++ OK, passed 100 tests.
```

And when it is run using QuickCheck we get the output that it passed 100 tests. While this number may be fairly low compared to the infinite number of possible test cases, it does give some confidence that this property holds and that the function is correct.

Another use case for which QuickCheck works is useful is for higher-order functions. For example we may want to test that applying the `map` function twice is the same as composing the two transformations. Luckily QuickCheck supports generating arbitrary functions [5]. To show this property, we could write:

```
prop_map_compose (f :: Int -> Int) (g :: Int -> Int) (xs :: [Int]) =
  map g (map f xs) == map (f . g) xs
```

However when used in this way, it has no way to `show` them since by default there is no instance of `Show (a -> b)`. To get around this, it is possible to import `Text.Show.Functions` which provides an implementation of it. It will show the following output:

```
ghci> quickCheck prop_map_compose
*** Failed! Falsified (after 2 tests and 1 shrink):
<function>
<function>
[1]
```

This is far from ideal since while it lets you know that a counterexample exists it does not tell you what it is. This makes finding the error in the method or the property very difficult. Instead you can use the `Fun` type introduced by Claessen [8] which supports `show`ing and `shrink`ing functions. It can be used in the previous example like this:

```
prop_map_compose (Fun _ (f :: Int -> Int)) (Fun _ (g :: Int -> Int)) (xs ::
[Int]) =
  map g (map f xs) == map (f . g) xs
```

With it, we can now see a valid counterexample:

```
ghci> quickCheck prop_map_compose
*** Failed! Falsified (after 3 tests and 9 shrinks):
{_->0}
{_->1}
[0]
```

With this counterexample in mind, we can notice that the property was incorrect. It has the function composition switched the wrong way around. With this fixed, it can now pass all of the property based tests.

3 Implementation

I have implemented new functionality within AGDA2HS to transform type classes into property based tests. This section outlines the changes made. This extension is evaluated and the limitations are discussed in Section 4.

To explain how the translation works, I will walk through generating properties of a lawful `Functor` type class for the `Maybe` type. The `Maybe` type represents a computation that can fail. It is defined as follows:

```
data Maybe (a : Type) : Type where
  Nothing : Maybe a
  Just : a → Maybe a
```

In order to show that is a lawful `Functor`, it must first be a `Functor`:

```
instance
  iFunctorMaybe = record{DefaultFunctor iDefaultFunctorMaybe}
  iDefaultFunctorMaybe .DefaultFunctor.fmap f Nothing = Nothing
  iDefaultFunctorMaybe .DefaultFunctor.fmap f (Just x) = Just (f x)
  {-# COMPILE AGDA2HS iFunctorMaybe #-}
```

We can now start creating the properties for this type class based on the `IsLawfulFunctor` definition:

```
postulate instance iLawfulFunctorMaybe : IsLawfulFunctor Maybe
{-# COMPILE AGDA2HS iLawfulFunctorMaybe laws #-}
```

With this, the first change I made to AGDA2HS is visible. The laws pragma on the last line is a new pragma added as part of my functionality. Its goal is to signal to AGDA2HS that it can ignore the implementation of the type class and instead it should generate properties. This was done as a pragma since the compilation between a normal type class and one with laws is different. Additionally it encourages type class authors to follow the same philosophy as the `agda2hs-base` of being able to gradually add the proofs of instances following the laws of a type class.

Once AGDA2HS compiles the symbol `iLawfulFunctorMaybe`, the `laws` pragma calls a function I have added[3]. The first thing my code does is look at the type and finds the name of the final returned thing. In this case the name it finds is `IsLawfulFunctor`. Once it has it, it can look up the information of it. The thing we care about in this is the fields of the record. In the case of `IsLawfulFunctor`, these are the fields it finds:

```
record IsLawfulFunctor (F : Type → Type) ⦃ iFuncF : Functor F ⦄ : Type₁ where
  field
    identity : (fa : F a) → (fmap id) fa ≡ id fa
    composition : (fa : F a) (f : a → b) (g : b → c) →
      fmap (g ∘ f) fa ≡ (fmap g ∘ fmap f) fa
```

From these fields, the code only includes the ones that are not hidden. This was done since there are some type classes which have instance fields to refer to previous properties that should hold. This way, a user can choose whether they want to generate tests for all the

[3]The full code for this function is in Appendix B

properties. The reason why the check does not use erasure is that none of the `IsLawful` type classes in the AGDA2HS standard library use erasure for their fields.

It now iterates over each of the fields. For purposes of describing the algorithm, I will only consider the `composition` law however the extension will perform these steps on each field. I start by inferring the type of the field:

```
  fieldTy <- liftTCM $ infer (Def (defName def) [Proj ProjSystem $ unDom law])
```

This would be roughly equivalent to writing `iLawfulFunctorMaybe .composition`. In this specific case, the type will be

```
{a b c : Type} (fa : Maybe a) (f : a → b) (g : b → c) →
  fmap (g ∘ f) fa ≡ (fmap g ∘ fmap f) fa
```

This is where we hit a limitation in QuickCheck. If we were to translate this as is to a QuickCheck property, we would not be able to test it since QuickCheck cannot test polymorphic properties. QuickCheck provides two solutions for this: using a template Haskell extension that defaults every property to an `Integer` or using provided wrapper types. Similarly to the template Haskell solution, I replace each of the polymorphic types with `Integer`. While in the future I would like to be able to support using arbitrary types, this was not possible to implement due to time constraints. This limitation will be detailed later.

After I have performed this application, the type becomes:

```
(fa : Maybe Integer) (f : Integer → Integer) (g : Integer → Integer) →
  fmap (g ∘ f) fa ≡ (fmap g ∘ fmap f) fa
```

From this, I wrap the final return type with `Dec` and then use instance search to look for a matching instance. This has the upside of being able to using any type as the return of the law as long as it has a `Dec` instance. However it also requires that there to be an instance of `Dec (x ≡ y)`. To solve this, I added an instance of type `⦃ _ : IsLawfulEq a ⦄ {x y : a} → Dec (x ≡ y)`. This means that in order to generate the QuickCheck tests, the user must provide an equality and prove that it is lawful. This way when trying to prove two things are equal, it will use the same notion of equality as running the QuickCheck properties.

The final step in the translation is to take the arguments to this function and transform them into the arguments to a function. To do this, I transform each argument into a pattern match where I specify its type. I then create a function declaration with the arguments and body and emit it to Haskell. For this law, the emitted declaration is:

```
prop_composition (fa :: Maybe Integer)
  (Fun _ (f :: Integer -> Integer)) (Fun _ (g :: Integer -> Integer))
  = fmap (g . f) fa == (fmap g . fmap f) fa
```

The property can now be ran with QuickCheck.

4 Analysis

In order to evaluate the output of the presented algorithm, I have selected some examples to look at. The first one is a real world example from the `transformers` package. The second is a type class that I couldn't find as a type class but could find a corresponding function in both a priority queue and a sorted list. The last set of examples are based on functions from the `base` package in Haskell.

4.1 Finding Errors

One case that is interesting is with the ListT monad transformer. This is a monad transformer which is used to add non-determinism to the effects in your monadic context. There used to be an implementation of `ListT` in `Control.Monad.Trans.List`. However this implementation is only a lawful monad when the inner monad `m` is commutative [9] and so was later removed. This section will show the implementation failing.

In order for the monad inside to be commutative, the order of execution of the effects must be commutative. This can be explained through the following property:

```
do
  a <- actA
  b <- actB
  m a b
==
do
  b <- actB
  a <- actA
  m a b
```

Examples of monads that are commutative include `Reader` (since the environment passed to all the actions is the same) or `Maybe` (since the all computations fail the same way). Examples of monads that are non-commutative include `Writer` with a non-commutative monoid (For example the `List` monoid since `xs ++ ys` is not always equal to `ys ++ xs`) or the `Either` monad (since not all errors are the same)

To show this requirement, I will replicate its implementation using a non-commutative inner monad:

```
-- newtype ListT m a = ListT { runListT :: m [a] }
record ListT (m : Type → Type) (a : Type) : Type where
  no-eta-equality; pattern; constructor ListT'
  field runListT : m (List a)
```

This consists of a single field which stores a list inside of a monad. The next thing to be added is the implementations of `fmap`, `<*>`, `pure` and `>>=`:

```
iDefaultFunctorListT .DefaultFunctor.fmap f (ListT' x) = ListT' $ (f <$>_) <$> x

iDefaultApplicativeListT .DefaultApplicative.pure = ListT' ∘ pure ∘ pure
iDefaultApplicativeListT .DefaultApplicative._<*>_ (ListT' mf) (ListT' mx) =
  ListT' $ _<*>_ <$> mf <*> mx

iDefaultMonadListT .DefaultMonad._>>=_ m k = ListT' $ do
  a <- runListT m
  b <- mapM {List} (runListT ∘ k) a
  pure (concat b)
```

These are the concrete implementations of the type classes in the monad hierarchy that are necessary to have a monad instance. With this implemented, I can then generate monadic laws for the properties, picking an inner monad that is known to fail. In Haskell, `Strings` are lists of `Char` so the same reasoning for why `Writer` of a List is a non-commutative monad applies to `Writer String`.

```
postulate instance iPreLawfulMonadListT : PreLawfulMonad (ListT (Writer String))
{-# COMPILE AGDA2HS iPreLawfulMonadListT laws #-}
```

This now generates the Haskell properties that correspond to the monadic laws. Below are a subset of the properties:

```
prop_leftIdentity (x :: Integer)
  (Fun _ (k :: Integer -> ListT (Writer String) Integer))
  = (return x >>= k) == k x
prop_associativity (ma :: ListT (Writer String) Integer)
  (Fun _ (f :: Integer -> ListT (Writer String) Integer))
  (Fun _ (g :: Integer -> ListT (Writer String) Integer))
  = do x <- ma
       f x >>= g
     == (ma >>= f >>= g)
prop_def_zap_bind
  (mab :: ListT (Writer String) (Integer -> Integer))
  (ma :: ListT (Writer String) Integer)
  = (mab <*> ma) ==
      do f <- mab
         x <- ma
         return (f x)
```

When running these three properties, this is the output:

```
ghci> quickCheck prop_leftIdentity
+++ OK, passed 100 tests.
ghci> quickCheck prop_associativity
*** Failed! Falsified (after 5 tests and 19 shrinks):
ListT (Writer' {runWriter = ("",[0,0])})
{_->ListT (Writer' {runWriter = ("a",[0])})}
{_->ListT (Writer' {runWriter = ("b",[])})}
ghci> quickCheck prop_def_zap_bind
*** Failed! Falsified (after 3 tests and 1 shrink):
ListT (Writer' {runWriter = ("\"",[])})
ListT (Writer' {runWriter = ("a",[])})
```

As expected it shows counterexamples for the associativity property but I wasn't expecting it to also fail the equality of `<*>` and `Control.Monad.ap`. With the counterexample it makes sense since with the provided counter example, with `ap` it ignores all effects after the first one and with `<*>` it combines the two. To see both the source and generated code for `Example1.Writer` see Appendix C.

Through this example, I can see an instance of a non-trivial monad that can be shown to be not correct. Trying to write proofs for these statements is non-trivial since `ListT` has a list inside of a monad. Without having tried to prove it, I think I likely would have had to use the `cong-monad` helper with a pattern lambda. By being able to find a counterexample beforehand, I can avoid the work of proving it.

4.2 Using `Decs` for laws

Certain data structures have a function to transform them into a sorted list. While I could not find any examples of this being modeled as a type class, I could find an implementation of a priority queue[4] and a sorted list[5] that provide a function with the same signature. A list provides the same functionality by using `Data.List.sort`. It is thus reasonable to create a type signature that provides this shared functionality. It could be expressed as a type class and instance with the following:

[4]https://hackage.haskell.org/package/priority-queue
[5]https://hackage.haskell.org/package/sorted-list

```
record ExtractSorted (cont : Type → Type) : Type₁ where
  field
    extractSorted : ⦃ _ : Ord a ⦄ → cont a → List a

{-# COMPILE AGDA2HS ExtractSorted class #-}

instance
  iExtractSortedList : ExtractSorted List
  iExtractSortedList = record { extractSorted = sort }
```

In the current version of AGDA2HS, the `sort` function is postulated so it has no implementation to write proofs against. Having some idea of correctness is still useful however. For this I construct an `IsLawfulExtractSorted` type class to signify what the expectations around it are.

```
record IsLawfulExtractSorted (cont : Type → Type) ⦃ _ : ExtractSorted cont ⦄
  : Type₁ where field
    IsAscending-extractSorted : ∀{a} ⦃ _ : Ord a ⦄ ⦃ _ : IsLawfulOrd a ⦄ →
      (xs : cont a) → IsAscending (extractSorted xs)
```

I am using the `IsAscending` predicate from the Section 2.2, since it represents that a list is sorted. This works because a byproduct of using `Dec` to emit the return value of the laws, I can use any predicate with a deciding procedure. Using the `IsLawfulExtractSorted`, I can then generate the corresponding property for `Lists`:

```
iLawfulExtractSortedList .IsAscending-extractSorted = prf where postulate
  prf : ∀{a} ⦃ _ : Ord a ⦄ (xs : List a) → IsAscending (extractSorted xs)
{-# COMPILE AGDA2HS iLawfulExtractSortedList laws #-}
```

Which generates the property using the `Dec` procedure for `IsAscending`:

```
prop_IsAscending_ascendingList (xs :: [Integer])
  = decIsAscending (extractSorted xs)
```

Running this property gives:

```
ghci> quickCheck prop_IsAscending_ascendingList
+++ OK, passed 100 tests.
```

This is a useful test for a couple of reasons. The first one is that `IsLawfulExtractSorted` shows that it is possible to use any type with a `Dec` instance as the result of the law. This is useful for a situation like this with an extrinsic predicate that you still want to verify. The other interesting result with this example is that `sort` is postulated on the AGDA2HS side. This means that the only way that anything could be proven about it is through postulating more properties. By instead creating a property based test for it, I can still verify that it holds for some amount of cases.

4.3 Limitations

In order to validate that my approach worked for a wide variety of type classes in the Haskell standard library, I ran it on some of the postulated instances inside of the AGDA2HS base. These are interesting to try since they should be guaranteed to pass even if the instances are postulated.

4.3.1 Arbitrary types in the instance declaration

The first instance I ran it on was the lawful functor instance for the tuple:

```
postulate instance iLawfulFunctorTuple₂ : IsLawfulFunctor (a ×_)
{-# COMPILE AGDA2HS iLawfulFunctorTuple₂ laws #-}
```

When trying to generate properties, the first limitation of my approach shows up: it does not support arbitrary types in the signature. This means that the arbitrary `a` makes it fail to compile. This can be resolved by changing the definition to have a specific type:

```
postulate instance iLawfulFunctorTuple₂ : IsLawfulFunctor (Integer ×_)
```

Which would then allow it to generate properties successfully.

4.3.2 Preconditions

When trying to generate properties for `Ord Nat` as follows:

```
postulate instance iLawfulOrdNat : IsLawfulOrd Nat
{-# COMPILE AGDA2HS iLawfulOrdNat laws #-}
```

It does not succeed. The reason for this is that some of the laws of `Ord` require a precondition. Consider the following law:

```
transitivity :  ∀ (x y z : a) → ((x <= y) && (y <= z)) ≡ True → (x <= z) ≡ True
```

In this case, there is a precondition. Currently there is nothing in my code to handle this. QuickCheck supports filtering preconditions with the `==>` function, so this could be added as future work.

4.3.3 Functions

While both Haskell and Agda support creating type class instances with functions, there is no way to create an instance of `Eq` for them. This is because in order for two functions to be equal, they must be equal for every input. This would take $O(n)$ where n is the size of the domain. This is possible for functions with small domains. For large domains such as 128 bit integers or the infinitely many natural numbers, this becomes computationally infeasible. In QuickCheck this can be tested by using a random value and checking if the two functions are identical. This can be done with the following:

```
prop_equal_functions x = f x == g x
  where f = ...
        g = ...
```

In order for this approach to work with generated property based tests, there has to be a way to change out the equality and the result of the equality.

This also affects all types for which you cannot implement `Eq` such as the `Reader` monad. Making changes to allow this is future work that would enable generating and running properties for more types.

5 Related Work

There are three main approaches in the literature that achieve similar results to translating type class laws into property based tests: the first is property based testing directly within a theorem prover, the second is testing type class laws directly within Haskell and the third is proving them directly within Haskell.

5.1 Property Based Testing within Theorem Provers

There have been other attempts to port QuickCheck to theorem proving languages. The general conclusion is that it is helpful to combine theorem proving and property based testing.

One such example was QuickChick in the Rocq theorem prover [10]. While Rocq also supports type classes so it would be possible to combine the efforts in Rocq to produce tests directly from type classes. QuickChick however has a slightly different scope to what I am trying to achieve since the focus is on writing verified tests while I was generating known good tests.

Property Based Testing has also been attempted in Agda 1 / Alfa [11]. The interesting thing in this paper is the method of combining testing and proving on smaller lemmas that are encountered is useful for both removing unwanted cases from the domain of the lemma and finding counterexamples to find bugs in the program or specification.

5.2 Generating Properties for Type Class Laws within Haskell

Within Haskell, there has already been prior work trying to generate QuickCheck properties for type class laws. Jeuring et al. [12] put forwards the ClassLaws framework which can run property based tests based on specifications defined using `type family` declarations to provide arguments. For each new property, this requires setting up a series of new declarations. If a user has already opted into using AGDA2HS, generating the property based tests corresponding to their type classes only requires postulating an instance, compiling it with the laws pragma and creating the necessary QuickCheck instances. AGDA2HS allows replacing the postulated instance of the laws with a proof of correctness once you are reasonably satisfied they are correct. This allows treating the property based testing as a step in formalizing the laws rather than the goal.

5.3 Generating Properties for Type Class Laws within Haskell

Work has also been done in the other direction to allow proving things from within Haskell using LiquidHaskell. Using refinement types and annotations on functions, it is possible to prove certain type class laws such as monoid [13] or Functor, Applicative and Monad [14]. While this approach can prove certain type class laws, it is mostly focused on proving and not finding counterexamples. This approach can be combined with property based testing to find counterexamples before attempting to prove statements.

6 Responsible Research

In order to ensure the reproducibility of this research, both the code shown in the paper and my modifications to AGDA2HS are available in a public repository[6]. The repository includes a cabal freeze file which ensures that the versions of dependencies are identical across installations.

To ease recreating the Haskell ouputs, I have added a bash script that reruns my fork of AGDA2HS on each of the sources.

The code is licensed under the same MIT license as AGDA2HS. This allows high compatibility with other research or projects who would like to use my extension.

Finally, there was no use of generative AI when creating this paper or writing the code.

[6]Repository: https://github.com/ggrandi/agda2hs/tree/type-class-invariants/research-project-examples

7 Conclusions and Future Work

My research explored how can type class laws in Agda be translated to QuickCheck properties in Haskell with AGDA2HS. To do this, I implemented an extension to AGDA2HS that allows generating QuickCheck properties for type classes that contain only properties. This extension is shown to be useful for saving time trying to prove laws that can be shown false and it allows generating test cases to validate that `postulate`d functions behave as intended. This was overall helpful in preventing me from spending time trying to prove statements I knew were incorrect.

In order to integrate this functionality into AGDA2HS there are a couple additions that should be implemented. The first is a better way to apply types to laws. What I envision is a variation of the laws pragma which specifies a name and a value to apply where it defaults to `Integer`. I would imagine it looking like `{-# COMPILE AGDA2HS ... laws a=Type1 b=Type2 #-}`. The second change is more explicit support for preconditions. This would enable generating checks for all of the laws in the standard library of AGDA2HS. Thirdly, I would like to be able to support emitting equality check that is not `Eq` since it does not work for functions. Potentially using something like the `type family` approach of ClassLaws. The final change I would like to make is supporting type classes where the laws and functions are combined in the same type class. This is currently not supported since the `laws` pragma I added can only transform laws and the base agda2hs can only transform functions.

Finally, there is one recommendation I have for AGDA2HS:

- I would recommend adding quotient types to AGDA2HS. The only way to create a version of `LawfulEq` is if `Eq` checks that all of the fields are equal. However there are some cases where that might not be desired. I ran into two examples were this would have been nice. The first is fractional numbers represented as the ratio of two integers, whose representations are the same if and only if they represent the same ratio such as $\frac{1}{2}$ and $\frac{2}{4}$. The second is a variant of the `Either` type were all errors are considered the same to be able to collect a list of errors in multiple branches of computation and still be a lawful monad.

A Extra Code for explaining Dec

Here is the proof that the definition I have given is isomorphic to the one in AGDA2HS. In this snippet of code, `Dec` is the definition shown in this paper and `Dec'` is the definition in AGDA2HS.

```
isomorphism : ∀{P} → (Dec P → Dec' P) × (Dec' P → Dec P )
isomorphism {P} .fst ((inhabited ⟨⟩) ⦃ p ⦄) = inhabited ⟨ of {P} p ⟩
isomorphism {P} .snd (inhabited ⟨ p ⟩)      = (inhabited ⟨⟩) ⦃ invert {P} p ⦄
```

Here is the code that gives the `Dec (IsAscending xs)` procedure:

```
instance
  decIsAscending : {a : Type} ⦃ _ : Ord a ⦄ {xs : List a} → Dec (IsAscending xs)
  decIsAscending {xs = []} = True ⟨⟩
  decIsAscending {xs = _ ∷ []} = True ⟨⟩
  decIsAscending {xs = x ∷ y ∷ xs} = mapDec
    (λ { (h , h') → ManyElem ⦃ _ ⦄ ⦃ h ⦄ ⦃ h' ⦄ })
    (λ { (ManyElem ⦃ h ⦄ ⦃ h' ⦄) → h , h' })
    (iDecPair {IsTrue (x <= y)} {IsAscending (y ∷ xs)})

  {-# COMPILE AGDA2HS decIsAscending #-}
```

This uses the mapDec function provided by AGDA2HS to say that the `ManyElem` case is isomorphic to a pair containing `IsTrue (x <= y)` and `IsAscending (y ∷ xs)`.

B Code for the Extension

```haskell
compileLaws :: Definition -> C [Hs.Decl ()]
compileLaws def = do
  reportSDoc "rp" 10 $ text "def: " <+> prettyTCM (defType def)
  -- TODO: inspect _defTs for applying values to the definition
  TelV _defTs defCore <- def & defType & telView
  qname <- case unEl defCore of
    Def q _ -> pure q
    x -> agda2hsErrorM $ text "expected def but got: " $+$ prettyTCM x
  reportSDoc "rp" 10 $ text "q: " <+> prettyTCM qname
  defInfo <- getConstInfo qname
  reportSDoc "rp" 10 $ text "defInfo: " <+> pretty defInfo
  defFields <- case theDef defInfo of
    RecordDefn x -> pure . _recFields $ x
    _ ->
      agda2hsErrorM $
        text "expected to compile a record with the laws pragma but got: " $+$ pretty defInfo
  integerTy <- resolveStringName "Integer" <&> (`Def` [])
  -- TODO: Use a different check for whether a field should be included as a law
  let laws = filter (\field -> argInfoHiding (domInfo field) == NotHidden) defFields
  (catMaybes <$>) $ forM laws $ \law -> do
    reportSDoc "rp" 10 $ text "f: " <+> pretty law
    fieldTy <- liftTCM $ infer (Def (defName def) [Proj ProjSystem $ unDom law])
    lawTy <- liftTCM $ applyX integerTy fieldTy
    reportSDoc "rp" 10 $ text "lawTy: " <+> prettyTCM lawTy
    let law_name = law & unDom & qnameName & nameCanonical & nameNameParts & haskellifyName
    let prop_name = Hs.Ident () ("prop_" ++ law_name)
    TelV fieldTs fieldTy <- telView lawTy
    reportSDoc "rp" 10 $ text "telescope args: " <+> prettyTCM fieldTs
    decidedExp <- addContext fieldTs $ do
      decTy <- decify fieldTy
      -- Ignore the type since it can be inferred on the Haskell side and I expect it to be a Boolean
      liftTCM (findInstance' decTy) >>= \case
        Nothing -> do
          agda2hsErrorM $ "No Dec instance found for" $+$ prettyTCM fieldTy
        Just decInst -> do
          reportSDoc "rp" 10 $ text "decinst: " <+> prettyTCM decInst
          compileTerm decTy decInst
    pats <- pats fieldTs
    pure . Just . Hs.FunBind () . pure $
      Hs.Match () prop_name pats (Hs.UnGuardedRhs () decidedExp) Nothing
 where
  pats :: Tele (Dom Type) -> C [Hs.Pat ()]
  pats = foldrM aux [] . telToList

  aux :: Dom (ArgName, Type) -> [Hs.Pat ()] -> C [Hs.Pat ()]
  aux Dom{domInfo, unDom = (name, ty)} pats
    -- TODO: handle instance arguments from the patterns
    | argInfoHiding domInfo /= NotHidden = pure pats
    | otherwise = do
        compiledTy <- compileType (unEl ty)
        let typedVar = Hs.PatTypeSig () (Hs.PVar () (Hs.Ident () name)) compiledTy
        case unEl ty of
          Pi _ _ ->
            pure . (: pats) $
              Hs.PApp () (Hs.UnQual () $ Hs.Ident () "Fun") [Hs.PWildCard (), typedVar]
          _ -> pure $ typedVar : pats

  -- \| Applies @x@ to all pi types of the form @(x : Set) -> ...@
  applyX :: Term -> Type -> TCM Type
  applyX x ty = case unEl ty of
    Pi arg body
      -- currently it just checks `Type₀` since agda2hs expects most types to be there
      | unEl (unDom arg) == Sort (Univ UType (Max 0 [])) -> do
          appliedTy <- piApplyM ty x
          applyX x appliedTy
      | otherwise -> do
          reportSDoc "rp" 100 $ text "N applyX: " <+> prettyTCM arg
          newBody <- (body $>) <$> underAbstraction arg body (applyX x)
          pure $ mkPiSort arg newBody `El` Pi arg newBody
    _ -> pure ty
```

C Links to full code files

All code and resulting code are inside of my fork. They can be found at https://github.com/ggrandi/agda2hs/tree/type-class-invariants.

Other examples that I used to as motivation throughout the project can be found at https://github.com/ggrandi/agda2hs/tree/type-class-invariants/tutorials/generating-instance-properties

References

[1] G. Hutton, *Programming in haskell*. Cambridge University Press, 2016.

[2] U. Norell, "Towards a practical programming language based on dependent type theory," phdthesis, 2007. Accessed: Jun. 15, 2026. [Online]. Available: https://www.cse.chalmers.se/~ulfn/papers/thesis.pdf

[3] J. Cockx, O. Melkonian, L. Escot, J. Chapman, and U. Norell, "Reasonable Agda is correct Haskell: writing verified Haskell using agda2hs," in *Proceedings of the 15th ACM SIGPLAN International Haskell Symposium*, in Haskell 2022. New York, NY, USA: Association for Computing Machinery, Sep. 2022, pp. 108–122. doi: 10.1145/3546189.3549920.

[4] B. Yorgey, "The Typeclassopedia," *The Monad.Reader*, vol. 13, pp. 17–68, 2009.

[5] K. Claessen and J. Hughes, "QuickCheck: a lightweight tool for random testing of Haskell programs," in *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, in ICFP '00. New York, NY, USA: Association for Computing Machinery, Sep. 2000, pp. 268–279. doi: 10.1145/351240.351266.

[6] D. Devriese and F. Piessens, "On the bright side of type classes: instance arguments in Agda," in *Proceedings of the 16th ACM SIGPLAN international conference on Functional programming*, in ICFP '11. New York, NY, USA: Association for Computing Machinery, Sep. 2011, pp. 143–155. doi: 10.1145/2034773.2034796.

[7] Agda Development Team, "Agda 2.8.0 Documentation." Accessed: Apr. 30, 2026. [Online]. Available: https://agda.readthedocs.io/en/v2.8.0

[8] K. Claessen, "Shrinking and showing functions: (functional pearl)," in *Proceedings of the 2012 Haskell Symposium*, in Haskell '12. New York, NY, USA: Association for Computing Machinery, Sep. 2012, pp. 73–80. doi: 10.1145/2364506.2364516.

[9] HaskellWiki contributors, "ListT done right." Accessed: Jun. 02, 2026. [Online]. Available: https://wiki.haskell.org/index.php?title=ListT_done_right&oldid=66355

[10] Z. Paraskevopoulou, C. Hriţcu, M. Dénès, L. Lampropoulos, and B. C. Pierce, "Foundational Property-Based Testing," in *Interactive Theorem Proving*, C. Urban and X. Zhang, Eds., Cham: Springer International Publishing, 2015, pp. 325–343. doi: 10.1007/978-3-319-22102-1_22.

[11] P. Dybjer, Q. Haiyan, and M. Takeyama, "Combining Testing and Proving in Dependent Type Theory," in *Theorem Proving in Higher Order Logics*, D. Basin and B. Wolff, Eds., Berlin, Heidelberg: Springer, 2003, pp. 188–203. doi: 10.1007/10930755_12.

[12] J. Jeuring, P. Jansson, and C. Amaral, "Testing type class laws," in *Proceedings of the 2012 Haskell Symposium*, in Haskell '12. New York, NY, USA: Association for Computing Machinery, Sep. 2012, pp. 49–60. doi: 10.1145/2364506.2364514.

[13] N. Vazou, L. Lampropoulos, and J. Polakow, "A tale of two provers: verifying monoidal string matching in liquid Haskell and Coq," in *Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell*, in Haskell 2017. New York, NY, USA: Association for Computing Machinery, Sep. 2017, pp. 63–74. doi: 10.1145/3122955.3122963.

[14] N. Vazou *et al.*, "Refinement reflection: complete verification with SMT," *Proceedings of the ACM on Programming Languages*, vol. 2, no. POPL, p. 53:1–53:31, Dec. 2017, doi: 10.1145/3158141.