Abstract

One of the advantages of a purely functional programming language such as Haskell is referential transparency. It is substantially easier to verify the correctness of a program in a purely functional programming language than in an imperative language. The fact that function definitions in Haskell are equalities allows us to reason about properties of a program by substituting expressions with equal expressions. This process is called equational reasoning.

Some type classes in Haskell exhibit properties, also called type class laws, that every corresponding type class instance is expected to obey. These properties ensure certain desirable behavior or they can be used to conclude additional properties. Type class laws are not enforced by the Haskell compiler, so we need to check them ourselves.

This report describes the technique of equational reasoning and how it can be used to prove that a type class implementation obeys the corresponding type class laws. Further, it gives a brief description of the concept of a type class and describes the type class `Monoid` and its laws in more detail. Finally, it walks through an application example. We use equational reasoning to prove a type class law for a simple plugin system written in Haskell.
1 Introduction

One of the advantages often attributed to purely functional programming languages is that they are great for [verification](#) and [equational reasoning](#) [15]. When I started learning to program in a functional language, I didn’t know precisely what reasoning about program is and how it can be used to verify certain [properties](#) of a program. I also didn’t know what kind of properties of a program can be verified with formal methods. In order to understand equational reasoning I wrote this article.

1.1 Why use equational reasoning?

Equational reasoning can be used as a formal method for software verification. It’s a way to prove the correctness of a program. Correctness means that specified properties of the program hold for an arbitrary input.

A property of a program can be specified in the form of an equation. For example, the following equation states a property of the function [fmap](#).

\[ \text{fmap \ id} = \text{id} \]  

1

The function [id](#) is the identity function. The property represented by equation 1 says that applying the function [fmap](#) with the arguments [id](#) and an arbitrary container is the same as applying the function [id](#) on a container. Simply put, mapping the identity function over every item in a container has no effect. Given a function definition of [fmap](#), we can check if this property holds, using equational reasoning.

The goal of equational reasoning is to provide certainty that a program has particular properties. Proof of required properties contributes to the reliability and robustness of a software. For example, the first formally verified micro kernel, seL4, was verified using Haskell and formal verification to ensure proper security functionality and security assurance [7]. Another example is the streaming library "pipes". The author, Gabriel Gonzales, used equational reasoning to prove the correctness of the library [5].

1.2 Why are functional programming languages great for equational reasoning?

An equation is a statement that signifies the equality of two expressions (e.g. \( 1 + 3 = 4 \)). Reasoning is the process of thinking about something in

1The function [fmap](#) takes a function \( a \to b \) as its first parameter and a container that contains elements of type \( a \). It returns a container with elements of type \( b \). This is called [mapping](#). The type of [fmap](#) is \( (a \to b) \to f a \to f b \).
a logical way in order to form a conclusion. The technique of equational reasoning implies that we show correctness by showing that two expressions of a program are equal.

The reason we can use equational reasoning in functional programming is referential transparency. A piece of code can be replaced by its value; this is true in purely functional languages. The absence of side effects allows us to replace the left-hand side of a definition with the right-hand side and vice versa. For example if you have the following Haskell definition:

\[ x = 23 \]

That means that any \( x \) can be substituted by 23.

This is not possible in programs written in an imperative programming language because there the value of an expression depends on the context of the execution environment. In an imperative language, \( x = 23 \) is an assignment, not an equality. The variable \( x \) is mutable and the value of the expression \( x \) can change at runtime. Hence, we must not substitute \( x \) with 23 in your program because arbitrary values could be assigned to the variable \( x \) in other statements.

Haskell is a purely functional language, which means that functions do not have side effects. Hence, we can verify programs written in Haskell using equational reasoning. For this reason, all examples in this article are written in Haskell.

1.3 What properties will we discuss?

If we want to verify a program, we have to define one or more properties. These properties are part of the program specification. We could define arbitrary properties, but this article mainly deals with properties derived from the so called type class laws. In Haskell, type class laws are exhibited by type classes. For example, the property formed by equation 1 is a type class law of the type class Functor (cf. appendix A.1).

Type classes are a concept to describe the behavior of a type. They are very similar to interfaces. If we want to make a type an instance of a type class, we have to implement the functions given by the type signature declarations of the type class. Figure 1 illustrates the relation between values, types and type classes in Haskell. A type can be a member of several type classes. A type class can contain several types. A value has exactly one type but several values can be of the same type.

An instance of a type class must obey the corresponding type class laws. These laws allow us to make assumptions about the behavior of the program and to reason about the code. Properties of existing code allow us to rely on
expected behavior and deduce further properties for new code. For example, if the property formed by equation 1 holds, we can rest assured that \texttt{fmap} only applies the function to the items in the container and has no additional effects. If an implementation of \texttt{fmap} would change the structure of the container, it would break the property of equation 1.

The Haskell compiler does not enforce type class laws. We need to verify them ourselves when implementing an instance of a type class. We can use equational reasoning to prove that an implementation of a type class obeys the corresponding laws.

### 1.4 Overview

This article will illustrate in detail how equational reasoning works in practice using the type class laws of the type class \texttt{Monoid} as the running example. The article doesn’t require prior knowledge of type classes or \texttt{monoids}. In order to understand every step of the process, the next two sections will explain the fundamental concepts. Section 2 will give a short introduction to type classes and section 2.2 describes monoids. In section 3 we will give an introduction to equational reasoning in general. In section 4 we will prove that an implementation of the \texttt{Monoid} type class obeys the first monoid law.

This article gives answers to the following questions:

- What are desired properties of a program? Where do they come from? Why are they useful?
- What is equational reasoning? What is the difference between testing and property proving? How do we use equational reasoning to verify properties of a program?

### 2 Type classes

The first part of this section will give a short description of the concept of a type class. The second part will describe monoids in general and the type class \texttt{Monoid}.

The concept of a type class was introduced as a construct that supports overloaded functions and \texttt{ad hoc polymorphism}. Overloaded functions
can be used with a variety of types, but with different definitions for the different types. For example, the function calls `show 1` and `show "hello"` use different function definitions. The function `show 1` is of type `Int -> String` and `show "hello"` is of type `String -> String`. The definition used depends on the type of the argument. The next section will describe ad hoc polymorphism and the relation to type classes in more detail.

2.1 Polymorphism and type classes

There are two types of polymorphism in Haskell [1]: Parametric polymorphism and ad hoc polymorphism. Type classes are used for ad hoc polymorphism.

**Parametric polymorphism** refers to functions which work over more than one type. For example, the library function `length` returns the length of a list `[]` that contains items of arbitrary type. It can be used to calculate the length of a list of integers, a list of strings, a list of booleans, etc. There are no constraints. The type of `length` is

```
length :: [t] -> Int
```

`t` is a type variable. That is, for any type `t` the function `length` has type `[t] -> Int`. A type that contains a type variable is called polymorphic. Hence, `length` is a polymorphic type. The `length` function can be used with any type but it has a single definition. At compile time, the type variables are substituted with a concrete type. For example `[Int] -> Int`.

**Ad hoc polymorphism** is a synonym for function overloading or operator overloading. An overloaded function uses different function definitions depending on the types of the arguments. Suppose we want to define a function that converts a list containing items of arbitrary type (`[t]`) to a string. We would write a function with the following type signature declaration:

```
showlist :: [t] -> String
```

It takes a list of arbitrary type `t` and returns a string. The definition could look like this:

```
showlist [] = ""
showlist (x:xs) = show x ++ showlist xs
```
We need a way to make sure that the function \texttt{show} is defined for the type of the value \texttt{x}. \texttt{show} can’t be a polymorphic type because the conversion depends on the type. There’s no single definition that can convert an arbitrary type to a string.

There’s a set of types. \texttt{show} is defined over all members of this set. This set is called a type class. The type class \texttt{Show} for example contains all types that can be converted to a string with the function \texttt{show}. In order to prevent the application of the function \texttt{showlist} with an argument that isn’t a member of the type class \texttt{Show}, we must constraint the type variable in the type signature declaration \texttt{t}:

\[
\texttt{showlist :: Show \texttt{t} \Rightarrow \texttt{[t]} \rightarrow \texttt{String}}
\]

Functions declared by the type class are defined over all members of the type class. And certain type classes exhibit properties that every definition of the corresponding functions must obey.

### 2.2 Monoid

In mathematics, a \textbf{monoid} is an algebraic structure with single associative binary operation and an identity element. Monoids are semigroups with identity \cite{17} \cite{12}. Several elements of a monoid can always be reduced to a single element by applying the corresponding binary operator. It doesn’t matter in which order we apply the operator, the result is always the same. This is called \textbf{associativity}. The set of elements has an identity element. For example, the set of natural numbers \(\mathbb{N}\) form a monoid under multiplication. The number 1 is the identity element. Multiplication of the identity and any other number \texttt{x} results always in \texttt{x}.

In Haskell there is a type class for monoids. Types that form a monoid can become part of the \texttt{Monoid} type class. For example, the type list \[\] forms a monoid. Two values of type list can always concatenated to another list with the \texttt{++} operator. The empty list \[\] is the identity element.

The example in section 4 shows a plugin system that contains a monoid. Plugins can be composed with a binary operator. An arbitrary number of plugins can be composed to a single plugin. Because the composition operator is associative, plugins can be evaluated in arbitrary order.

#### 2.2.1 Functions of the type class monoid

Members of the type class \texttt{Monoid} have to implement the functions \texttt{mempty} and \texttt{mappend} amongst others (see appendix A.3 for a complete declaration). These functions have the following type signature declaration.
Figure 2: An Applicative that encapsulates a Monoid is a Monoid

\[
\begin{align*}
\text{mempty} & : m \\
\text{mappend} & : m \rightarrow m \rightarrow m
\end{align*}
\]

The type variable \(m\) is the type of the corresponding monoid. \texttt{mempty} returns the identity value. \texttt{mappend} is the binary function that takes two values of the same type and returns another value of that type.

The type class \texttt{Monoid} exhibits several laws. We will only describe the one that we will prove in the example of section 4, the left identity law. When making monoid instances, we need to make sure that \texttt{mempty} acts like the identity with respect to the \texttt{mappend} function. This property can be expressed with the following equation:

\[
\text{mappend}(\text{mempty}, x) = x \tag{2}
\]

Equation 2 states that \texttt{mempty} has to behave like the identity with respect to \texttt{mappend}. When \texttt{mappend} is applied with the identity and an other element \(x\) of the monoid, it returns \(x\).

### 2.2.2 Example monoid implementation

There is a useful property of the \texttt{Applicative} type class with respect to the \texttt{Monoid} type class (the \texttt{Applicative} type class is described in more detail in the appendix, section A.2). The example in section 4 will use this property. If \(f\) is an \texttt{Applicative} and \(b\) is a \texttt{Monoid} then \(f\ b\) is also a \texttt{Monoid}. If a type is part of the \texttt{Applicative} type class and the type contains a \texttt{Monoid} we can create a \texttt{Monoid} instance with the implementation in listing 1. Figure 2 illustrates this property.
Listing 1: Monoid instance implementation of IO

```haskell
instance (Applicative f, Monoid a) => Monoid (f a) where
  mempty = pure mempty
  mappend = liftA2 mappend
```

mempty is of type \( f \ a \). Hence \( \text{pure mempty} \) has to be of type \( f \ a \). As \( f \) is an Applicative, it implements \( \text{pure} \). The type of \( \text{pure} \) is \( a \rightarrow f \ a \) (see appendix A.2). We call \( \text{pure} \) with \( \text{mempty} \) of type \( a \). We know that \( a \) is part of \( \text{Monoid} \) because of the type constraints. The compiler will use \( \text{mempty} \) of \( a \). \( \text{liftA2} \) is an utility function of \( \text{Applicative} \). It encapsulates the \( \text{mappend} \) function in an applicative functor.

In section 4 we prove that the function definition of listing 1 obeys the first monoid law formed by equation 2 with the verification technique equational reasoning.

### 3 Equational Reasoning

Verification is the process of checking if software does what its specification demands. To verify a program, a specification is required. In the case of functions that implement an instance of the type classes of section 2 the specification is defined in form of the type class laws. This section will compare different verification techniques and describe the method equational reasoning by example using type class laws as specification. In section 4 equational reasoning will be applied to proof the property of listing 1 in section 2.

There are several ways to check the behavior of a program. We will describe the difference with a simple example. Given the following property:

\[
\text{reverse(reverse (xs)) = xs}
\]

Equation 3 shows that if we apply \( \text{reverse} \) twice on the same list \( xs \) we get back the original list \( xs \). \( \text{reverse} \) is the inverse of \( \text{reverse} \) (other functions, like \( \text{id} \), have this property too). Verification techniques allow us to check if equation 3 holds. We describe three techniques. The first two, testing and property-based testing, are very common and the third one is the topic of this article.
**Testing** Run the program with a selected input and check if it behaves as expected. In order to check the behavior, a function evaluates both sides of equation \( \text{Eq. 3} \) and compares the values. The following listing shows an example test.

```haskell
input = [1,2,3]
test_reverse :: Bool
test_reverse = reverse (reverse input) == input
```

Listing 2: Function definition for testing

The selected input is \([1,2,3]\). test_reverse evaluates to a Boolean expression to indicate if the property holds for the given input. It’s necessary to run the program to evaluate test_reverse. An advantage of this method is, that the programmer doesn’t have to define general properties. The specification is expressed with a concrete input value and a concrete output value. It’s easier to think about a concrete input and the corresponding output value than to find general properties that a function must obey.

**Property-based testing** The input for the test program is generated randomly. The tests are executed by a tool (e.g. QuickCheck).

**Proof** A Proof can show that a property holds in all circumstances. To prove a property we use the technique equational reasoning. This technique requires knowledge of the function definition.

Figure 3 compares the input coverage of the described methods. Testing checks if the program behaves correctly with one chosen point of the input space. Property-based testing checks the behavior at hundreds of randomly-generated points. Proving a property covers all possible cases of the possible input. It is the most reliable verification.

### 3.1 Reasoning about algebraic properties

Equational reasoning is a method originally used in algebra. It’s the process of proving a given property by substituting equal expressions. For example, it’s possible to show that the following property holds:

\[
(x + a)(x + b) = x^2 + (a + b)x + ab
\]  

\( (4) \)

To show that the equality holds, we have to transform the expression on the left-hand side \((x + a)(x + b)\) to an equal expression on the right-hand side.
with the help of the basic algebraic properties of numbers (distributive law, commutative law) until we get $x^2 + (a + b)x + ab$.

In the first step (equation 5) we use distributivity to expand the term on the left-hand side.

$$(x + a)(x + b) = x^2 + ax + xb + ab \text{ (use distributivity)} \quad (5)$$

In the second step we use commutativity to substitute $xb$ with $bx$.

$$x^2 + ax + xb + ab = x^2 + ax + bx + ab \text{ (use commutativity)} \quad (6)$$

In the last step we use distributivity to factorize $x$ and we get $x^2 + (a + b)x + ab$.

$$x^2 + ax + bx + ab = x^2 + (a + b)x + ab \text{ (use distributivity)} \quad (7)$$

All we did was substituting expression according the algebraic properties.

### 3.2 Reasoning about Haskell programs

A function definition in Haskell means that we can substitute the left-hand side with the right-hand side and vice versa. This is possible because Haskell is a purely functional language. Hence, we can use the same approach to prove that a property of a program written in Haskell holds, as we used to reason about mathematical expressions. For example, it’s possible to show that the length of a list with one element is actually 1. This general property can be formed as an Haskell expression.

$$\text{length } [x] = 1$$
The property holds no matter what \( x \) is. To show that, we use the function definition of length as a general description of the behavior. Listing 3 shows the definition of length.

\[
\text{length} \ [\ ] = 0 \\
\text{length} \ (x:xs) = 1 + \text{length} \ xs
\]

Listing 3: Function definition of length

In order to conclude that \( \text{length} \ [x] == 1 \) is always true, we substitute \( \text{length} \ [x] \) until we get 1. Listing 4 shows the step by step substitution. It’s a stepwise transformation of the expression \( \text{length} \ [x] \) to 1.

\[
\text{length} \ [x] \quad -- \quad [x] \text{ is the same as } x:[] \\
= \text{length} \ (x:[]) \quad -- \quad \text{apply definition} \\
= 1 + \text{length} \ [] \quad -- \quad \text{apply definition} \\
= 1 + 0 \quad -- \quad 1 + 0 = 1 \\
1
\]

Listing 4: step by step substitution of length

Function definitions are general descriptions and we can use them to deduce other general properties by substituting equal expressions.

### 3.3 Proof by structural induction

If we apply simple substitution to a recursive function, we run into problems. Consider the function definition of length in listing 3. If we substitute \( \text{length} \ x \) with the definition \( 1 + \text{length} \ xs \), we end up substituting \( \text{length} \ x \) forever. A way to verify recursive programs is to use proof by structural induction. Proof by structural induction can be used for list or algebraic data types with a recursive constructor (e.g. Tree).

The principle of induction states, that it is sufficient to prove a property \( p \) for the base case and that \( p \) is preserved by the inductive case. In order to prove \( p \), two steps are required:

**Base case** Prove \( p(0) \) is true.

**Induction step** Prove \( p(n+1) \) if \( p(n) \) (induction hypothesis) is true.

Proof by induction is similar to writing a recursive function. Recursive functions use a base case (e.g. \([\ ], 0\)). If we use structural induction we proof the base case. We show that the property holds for a concrete input value (e.g. \([\ ], 0\)).
In a recursive function definition we define \( f \ (x:xs) \) and use \( f \ x \) in the right-hand side. In the proof we show that \( p(n+1) \) with the assumption that \( p(n) \) holds.

We explain proof by structural induction with another example from \([14]\). We verify that the overall length of two concatenated lists \( xs \) and \( ys \), is the same as the sum of the length of \( xs \) and the length of \( ys \). The ++-operator concatenates two lists.

\[
\text{length } (xs + + ys) = \text{length}(xs) + \text{length}(ys) \tag{8}
\]

In order to verify property \( \text{(8)} \) we need the function definitions for \text{length} and (++). Listings \([3]\) and \([5]\) show the function definitions of \text{length} and (++)

\[
[] + + xs = xs
(x:xs) + + ys = x:(xs++ys)
\]

Listing 5: Haskell function definition of the concatenation operator

**Base case** We have to show that property \( \text{(8)} \) holds for the base case. The base case, in this example, are the arguments \( [] \) and an arbitrary list for \( ys \). It is not necessary to replace \( ys \) because the definition of ++ uses recursion over \( xs \). \( ys \) will always be the same list. In order to check if property \( \text{(8)} \) holds for the base case, we replace \( xs \) with \( [] \), leading to the following Haskell expression.

\[
\text{length } ([] + + ys) = \text{length } [] + \text{length } ys
\]

We will evaluate the expression stepwise on the left-hand side and the right-hand side separately. The left-hand side evaluates to

\[
\text{length } ([] + + ys) -- \text{apply } ++
= \text{length } ys
\]

The right-hand side evaluates to

\[
\text{length } [] + \text{length } ys -- \text{apply } \text{length } []
= 0 + \text{length } ys
= \text{length } ys
\]

When we evaluate each side of the equation \( \text{(8)} \) with the value \( [] \) for \( xs \), the result is \text{length } ys on both sides. Hence, property \( \text{(8)} \) holds for the base case.
**Induction step** We have to show that if equation 8 holds for any list \( xs \) then it also holds for \( x:xs \) (\( x:xs \) is the list \( xs \) with a element \( x \) attached to its head). Therefore we have to show that the two expressions on both sides of the \( = \) sign of the expression in listing 6 are equal with the assumption that the property of equation 8 holds (induction hypothesis).

\[
\text{length } ((x:xs) ++ ys) = \text{length } (x:xs) + (\text{length } ys)
\]

Listing 6: Equality expression for induction step

Again, we evaluate the the left-hand side of equation in listing 6 step by step.

\[
\begin{align*}
\text{length } ((x:xs) ++ ys) & \quad \text{-- apply definition of } ++ \\
& = \text{length } (x:(xs ++ ys)) \quad \text{-- apply definition of length} \\
& = 1 + \text{length } (xs ++ ys) \quad \text{-- use induction hypothesis} \\
& = 1 + \text{length } xs + \text{length } ys
\end{align*}
\]

If we evaluate the right-hand side of the equation in listing 6 we get:

\[
\begin{align*}
\text{length } (x:xs) + \text{length } ys \quad \text{-- apply definition of length} \\
1 + \text{length } xs + \text{length } ys
\end{align*}
\]

The last listing shows, that the equality in listing 6 follows from the induction hypothesis in equation 8. This completes the induction step and therefore the proof itself.

The previous example used function definitions of \texttt{length} and \texttt{(++)}. In order to apply equational reasoning we have to know the function definitions of involved functions or we can rely on already proven properties. For example all types of the standard library, that are an instance of a type class, satisfy the type class laws (see 2) [18]. Some libraries exhibit properties in their documentation (e.g. pipes library [5]).

### 4 Example Proof for Monoid Laws

In this section we use equational reasoning (see section 3) to prove the left identity law of the monoid type class (see section 2.2) for a new type. The following example is based on a blog post of Gabriel Gonzales [4]. We simplified the types of a plugin and extended the example with an additional plugin implementation.
Suppose we want to build a plugin system. A plugin in our example is an IO action that takes a Char value and does some work with it (e.g. log to a file, potentially with side effects). Hence, a plugin is of type Char -> IO (). Listing 7 shows the definition of the plugin logto. It writes the first argument of type Char to a file named log.txt.

```
import System.IO
logto :: Char -> IO ()
logto c = do
  handle <- openFile "log.txt" WriteMode
  hPutChar handle c
```

Listing 7: Definition of a plugin that writes a character to a file.

Listing 8 shows an application of the logto plugin. We read a character c from the standard input and call logto c.

```
main = do
  c <- getChar
  logto c
```

Listing 8: Calling the logto IO action in main

Listing 9 shows an additional plugin that prints the character to the standard output.

```
print2stdout :: Char -> IO ()
print2stdout c = putChar c
```

Listing 9: Definition of the plugin print2stdout

In order to extend the program from listing 8, we want to be able to compose several plugins. Listing 10 gives an example of the composition of the plugins logto and print2stdout. We apply the mappend function from the Monoid type class to logto and print2stdout. Both arguments are of type Char -> IO (). The return value is also of type Char -> IO (). Listing 10 only works if mappend is implemented for the type Char -> IO (). We will give an implementation later in this section.

```
composedPlugin :: Char -> IO ()
composedPlugin = logto `mappend` print2stdout
```

Listing 10: Composition of two plugins

In addition, we demand that the order in which we evaluate the plugins must not matter. The plugins have to work independently of each other. For example, the behavior of the plugins composed1 and composed2 from listing 11 should be the same (donothing is another plugin of type Char -> IO ()).
composed1 = logto 'mappend' (print2stdout 'mappend' donothing)
composed2 = (logto 'mappend' print2stdout) 'mappend' donothing

Listing 11: The order in which we evaluate the plugins does not matter

The monoid laws state that \textit{mappend} must be associative. Hence, if we can prove that our implementation of \textit{mappend} satisfies the monoid laws, we can combine them, using the monoid function \textit{mappend} and we are able to add plugins without concerning about the order of evaluation. In addition, they are easier to use because they will behave as expected.

The plugins are of type \texttt{Char -> IO ()}. Instead of writing a specialized instance for \texttt{Char -> IO ()}, we use the general implementation of section 2.2. The instance implementation is repeated for convenience:

\begin{verbatim}
{-# LANGUAGE FlexibleInstances #-}
{-# LANGUAGE OverlappingInstances #-}

instance (Applicative f, Monoid a) => Monoid (f a) where
  mempty = pure mempty
  mappend = liftA2 mappend

Listing 12: Monoid instance
\end{verbatim}

The generalization has the advantage of not having that to prove the type class law only once for all types that match the general type declaration. The process of verifying a program is cumbersome and time consuming. Hence, the generalization of proofs is desirable.

We will verify that the definition 12 satisfies the left identity law of the \textit{Monoid} type class in two steps. First, we describe, why the instance implementation of listing 12 makes the type \texttt{Char -> IO ()} part of the \textit{Monoid} type class. Then we prove that the left identity law from section 2.2 holds for the \textit{Monoid} instance implementation of listing 12. We will assume that the instance implementation of listing 12 is a \textit{Monoid} in the first step.

4.1 Generalization of the plugin type

In order to use the instance implementation of listing 12, the type \texttt{(-\rightarrow)} \texttt{Char} (that is the type of a Haskell function) has to be part of the type class \textit{Applicative} and the type \texttt{IO ()} has to be part of the type class \textit{Monoid}. \texttt{IO ()} is a \textit{Monoid} if the type \texttt{IO} is an \textit{Applicative} and \texttt{()} is a \textit{Monoid}.

Here is an overview of all the steps of the argumentation:

1. Show that \texttt{Char \rightarrow IO ()} is a \textit{Monoid}.
2. Show that \texttt{(-\rightarrow)} \texttt{Char} is an \textit{Applicative} and \texttt{IO ()} is a \textit{Monoid}.

15
3. Show that IO is an Applicative and () is a Monoid.

We show the required properties in reversed order.

3. The standard library provides a Monoid instance for () [3] and an Applicative instance for IO [13]. All type class instances of the standard library obey the corresponding type class laws [8].

2. If the implementation from listing [12] satisfies the monoid laws then IO () is a Monoid because IO is part of the Applicative type class [13] and () is a Monoid. The type (->) r (that’s the type of Haskell functions) is a part of the Applicative type class [13].

1. If the implementation from listing [12] satisfies the monoid laws then Char -> IO () is a Monoid because (->) Char is an Applicative and IO () is a Monoid. The compiler will use the instance implementation for the Monoid type class from listing [12] for the type Char -> IO () because it matches the type

(Applicative f, Monoid b) => Monoid (f b)

Notice that we rely heavily on the assumption that listing [12] satisfies the monoid laws. The next section will prove that the implementation is correct.

4.2 Proof

In this section we will show that the implementation in listing [12] satisfies the left identity law of the Monoid type class (see section 2.2). The left identity law demands that:

\[ \text{mappend mempty } x = x \]

We use equational reasoning to show that the left-hand side is equal to x. First we use the definitions of mappend and mempty of listing [12] to substitute the left-hand side. Furthermore we look up the definition of liftA2 in the source code [13] to evaluate the expression. Listing [13] shows the definition of liftA2.

liftA2 f x y = (pure f <<- x) <<- y

Listing 13: Function definition of liftA2

mappend mempty x -- apply def. mappend
= liftA2 mappend mempty x -- apply def. mempty
= liftA2 mappend (pure mempty) x -- apply def. of liftA2
= (pure mappend <*> pure mempty) <*> x

To resolve this expression further, we use the laws of the Applicative type class described in the appendix A.2. One law of the Applicative type class states:

pure f <*> pure x = pure (f x)

We can use this property to substitute the left-hand side. Next we write mappend mempty as lambda function \( \lambda a \rightarrow \text{mappend mempty } a \) and use the monoid law

mappend mempty x = x

To simplify the expression. In the last step we use the first applicative law

pure id <*> v = v

to rewrite the expression as \( x \).

(pure mappend <*> pure mempty) <*> x -- 3. applicative law
= pure (mappend mempty) <*> x -- transform to lambda
= pure (\( \lambda a \rightarrow \text{mappend mempty } a \)) <*> x -- 1. monoid law
= pure (\( \lambda a \rightarrow a \)) <*> x -- a -> a = id
= pure id <*> x -- 1. applicative law
x

That completes the proof.
The example demonstrated several ideas:

- Type classes allow us to generalize definitions. A proof for the generalization is valid for all specializations.

- To prove a type class law we can use equational reasoning.

- Type class laws (or properties) allow to prove further properties.
5 Conclusion

This article illustrated the verification method equational reasoning by example. We proved that the monoid law known as left identity holds for a given function definition.

The type class laws provide a specification for the verification process. In addition, we can rely on properties of existing type class instances to prove further properties.

Type classes allow us to generalize definitions. A proof for the generalization is valid for all specializations. Hence, the proof is reusable.

The examination of the topic improved my comprehension for the advantages of a purely functional language. The reason why verification in a purely functional language like Haskell is easier than in imperative language is because functions are just equalities. We can reason about Haskell code in the same way we reason about mathematical equations. The definitions are stateless. This fact does not apply to mainstream languages. A function definition of an imperative language is allowed to change the context. These definitions are stateful.

Personally, I found the process of proving the left identity law for the given definition tedious and difficult. The proof requires creativity and a strong mathematical background. The verification process would be too cumbersome and expensive to apply it to every piece of software. Although equational reasoning is not suitable for software with a short life cycle, I think it is important to know the difference between testing and verification by proof.

A Appendix

A.1 Functor

Functor is a type class for types, which can be mapped over. Another way to describe functors is that they represent some sort of computational context. Listing 14 shows the declaration of the Functor type class.

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

Listing 14: Declaration of Functor type class

The `f` in the declaration is a type class constructor. Only type constructors can implement `Functor` (e.g. `Maybe`, `[]`).

`fmap` takes any function `a -> b` and a value of type `f a` (f is the container or context, `a` is the type wrapped inside the functor) and returns a
value of type \( f \mathbf{b} \). If \( f \mathbf{a} \) is of type \( \text{Maybe} \ \mathbf{Int} \) and the function of type \( \mathbf{Int} \to \text{String} \), \( \text{fmap} \) returns \( \text{Maybe} \ \text{String} \).

Examples of \text{Functor} instances are:

\textbf{List} \( \text{fmap} \) applies the function to every element in the list.

\textbf{Either} \( \text{Either} \ e \ \mathbf{a} \) is a container. \( \text{fmap} \) applies a function to \( \mathbf{a} \).

To make a type an instance of \text{Functor}, it has to define \( \text{fmap} \). In addition, the instances are expected to obey certain properties. The declaration of the type class doesn’t reveal these properties. They are described in the type class documentation \[10\] \[11\]. These properties are called the functor laws. All \text{Functor} instances in the standard library obey these laws \[18\] \[8\].

A \text{Functor} instance has to satisfy the following laws.

\textbf{Law 1} Mapping the identity function over a \text{Functor} value, will not change the functor value. Formally

\[ \text{fmap} \ \text{id} = \text{id} \]

\textbf{Law 2} It doesn’t matter if we compose two functions and then map them over a functor or if we first map one function over the functor and then map the other function. Formally

\[ \text{fmap} \ (g \circ h) = \text{fmap} \ g \circ \text{fmap} \ h \]

This is the same as \[ \text{fmap} \ (g \circ h) = \text{fmap} \ (\text{fmap} \ g) \circ \text{fmap} \ h \]

If we can prove that a type satisfies these laws, we can make assumptions about how the type will act. We know that \( \text{fmap} \) will not change the structure or the context of the functor. And we know that \( \text{fmap} \) only maps the function over the functor and nothing else.

\textbf{A.2 Applicative Functor}

Applicative functors are abstract characterizations of an applicative style of effectful programming \[9\] \[13\].

The \textbf{Applicative} type class encapsulates the following idea. What if we have a function wrapped in a \text{Functor} (e.g. \text{Maybe} \ (\mathbf{Int} \to \mathbf{Int} \to \mathbf{Int})) and you want to apply the function to another functor (e.g. \text{Maybe} \ \mathbf{Int}). For example, we want to map \text{Just} \ (3 \ *) , a function encapsulated inside a functor, over \text{Just} \ 23 , another functor with an encapsulated \mathbf{Int}. \( \text{fmap} \) doesn’t work here, because it expects a function of type \( \mathbf{a} \to \mathbf{b} \) as first parameter. That’s where the \text{Applicative} type class comes in. The \text{Applicative} type class is defined in listing \[15\] \[13\].
class (Functor f) => Applicative f where
  pure :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b

Listing 15: Declaration of Applicative type class

Every type that is part of Applicative is part of Functor. Hence we can use \texttt{fmap} with Applicative instances. Applicatives are enhanced functors. In addition to \texttt{fmap} we can use the \texttt{(*)-}operator to chain several Applicative values together.

The \texttt{pure} function puts a value of type \texttt{a} in a default context. If applied with a function \texttt{a -> b}, \texttt{pure} returns a functor with a function inside, \texttt{f (a -> b)}, hence the first argument of \texttt{(*)}.

There are several laws that instances of the Applicative type class should satisfy \cite{19} \cite{13}. We only need to know the following:

1. \texttt{pure id <*> v = v}
2. \texttt{fmap g x = pure g <*> x}
3. \texttt{pure f <*> pure x = pure (f x)}

The second law states that applying \texttt{fmap} over a function \texttt{g} over a functor \texttt{x} is the same as putting \texttt{g} in a default context and mapping the resulting function over \texttt{x}.

The third law states that it does not matter if we put the values \texttt{f} and \texttt{x} in a default context first and then apply \texttt{(*)} or if we call \texttt{f} on \texttt{x} first and then put them in a default context.

\section*{A.3 Monoid}

The Monoid type class contains types with an associative binary operation that has an identity Monoids are described in more detail in section \texttt{2.2}. Listing \texttt{16} shows the complete definition.

class Monoid m where
  mempty :: m
  mappend :: m -> m -> m
  mconcat :: [m] -> m
  mconcat = foldr mappend mempty

Listing 16: Definition of type class monoid
References


**Glossary**

*ad hoc polymorphism* Ad hoc polymorphism is also known as function overloading or operator overloading. Polymorphic functions can be applied to arguments of different types, because a polymorphic function can denote a number of distinct implementations depending on the type of the arguments to which it is applied.

*associativity* Within an expression containing two or more occurrences in a row of the same associative operator, the order in which the operations are performed does not matter as long as the sequence of the operands is not changed. That is, rearranging the parentheses in such an expression will not change its value.

*commutative law* Changing the order of two factors, does not change the product, for example.
**distributive law**  Multiplying a number by a sum of numbers is the same as doing each multiplication separately. [10]

**function definition**  In Haskell a function definition is an equation. The left-hand side is the function name and it’s arguments. The right-hand side is contains the expression of the function body. [5] [8] [11] [13]

**monoid**  A monoid is an algebraic structure with single associative binary operation. they are semigroups with identity plural. [4] [6]

**referential transparency**  An expression is said to be referential transparent if it can be replaced with its value without changing the behavior of a program... [3]

**type class**  A type class is a type system construct that supports ad hoc polymorphism. This is achieved by adding constraints to type variables in parametrically polymorphic types. [3]

**type signature declaration**  A type signature declaration tells, what type a Haskell variable has. [3] [5]

**verification**  Verification is the evaluation of whether a system complies with a specification. [2]