CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Proving Type Class Laws in Haskell

Andreas Arvidsson & Robin Touche

Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2016

Proving Type Class Laws in Haskell
Andreas Arvidsson & Robin Touche

Supervisor: Moa Johansson,
Department of Computer Science and Engineering
Examiner: Patrik Jansson,
Department of Computer Science and Engineering

Computer Science and Engineering
Chalmers University of Technology
SE-412 96 Gothenburg
Sweden
Telephone +46 (0)31 772 1000

Department of Computer Science and Engineering
Gothenburg, Sweden, 2016

Proving Type Class Laws in Haskell
Andreas Arvidsson & Robin Touche
Chalmers University of Technology
University of Gothenburg
Computer Science and Engineering
Gothenburg, Sweden

Abstract

Type classes in Haskell are a way to ensure both the programmer and the compiler that a set of functions are defined for a specific data type. Each instance of such functions is often expected to behave in a certain way and satisfy laws associated with the respective class. These are commonly stated in comments and as such there is no real way to enforce this behavior other than by rigorous testing. This thesis presents a tool able to prove such laws using induction and external theorem provers. The tool is an extension of an already existing system, HipSpec, with the added functionality of being able to handle type classes. Using this extended HipSpec, users can automatically prove laws for instances of built-in type classes, as well as define and prove laws for their own type classes. We discover that this is a very promising approach which has several advantages over similar systems.

Keywords: Haskell, Type Class, TIP, HipSpec

Acknowledgements

We would like to thank our supervisor Moa Johansson for the initial thesis proposal and for providing valuable feedback, ideas and information throughout the whole project. Without her aid, the structure and content of both the thesis report and the oral presentation would be a mess, not to mention the semantics of the extension itself.

We would also like to thank our examiner Patrik Jansson for not only helping us with the thesis but for providing valuable information from the field of functional programming and program verification as well.

Someone else who deserves our thanks is Dan Rosén who has helped and supported us during this project. We owe a great deal of gratitude to both him and all the other people who contributed to the development of HipSpec. Without their work, this thesis would not exist. In particular, we want to direct our thanks towards Koen Claessen and Nicholas Smallbone, who along with Moa and Dan published the paper on HipSpec.

Finally, we would like to thank our friends who provided support by giving feedback on our work and thesis as a whole as well as participating in interesting and engaging discussions.

Contents

1 Introduction **6**

2 Background **8**

2.1 Type Classes in Haskell . . . 8

2.1.1 Type Class Laws . . . 9

2.2 HipSpec . . . 10

2.3 TIP . . . 11

2.4 GHC Core . . . 12

2.4.1 The Compilation Process . . . 12

2.4.2 The GHC Core Syntax . . . 13

2.4.3 Type Classes in GHC Core . . . 14

3 Implementation **16**

3.1 Expressing Type Class Laws in Haskell . . . 16

3.2 Specialization Process . . . 17

3.2.1 Dictionary Extraction and Generation . . . 18

3.2.2 Monomorphization . . . 20

3.2.3 Dictionary Inlining . . . 21

3.2.4 Resolving dictionary lookups . . . 22

3.3 Generation of TIP Code and Proof . . . 22

4 Evaluation **25**

4.1 Experimental Setup . . . 25

4.2 Data Types . . . 26

4.3 Built-in Type Classes . . . 26

4.3.1 Class: Eq . . . 27

4.3.2 Class: Ord . . . 29

4.3.3 Class: Monoid (Built-in) . . . 31

4.3.4 Class: Functor . . . 33

4.4 User-defined Type Classes . . . 34

4.4.1 Class: Semigroup . . . 34

4.4.2 Class: Monoid (User-defined) . . . 35

4.4.3 Class: Semiring . . . 37

4.4.4 Class: CommutativeSemiring . . . 39

4.5 Summary and Discussion . . . 40

5 Related Work **42**

5.1 QuickCheck . . . 42

5.2 Isabelle . . . 42

5.3 Zeno . . . 43

5.4 HALO . . . 43

5.5 HERMIT . . . 43

6 Further Work and Limitations **45**
6.1 HipSpec Improvements 45
6.2 Implementation Limitations 46
6.3 TIP Language Improvements 47

7 Summary and Conclusion **49**

Appendix A Laws for Built-in Type Classes **51**

1. Introduction

The concept of type classes was first introduced in 1988 by Philip Wadler and Stephen Blott [25] as a way to control ad-hoc polymorphism in a structured way. By using them, developers are able to define a set of functions which operates differently based on the types of their arguments. This allows for greater flexibility when writing code in Haskell since it allows functions to be defined with a generic implementation for a specific set of data types.

Each type class contains a set of class method declarations which should be implemented when defining instances of that class. These methods often have laws associated with them which describes what properties should hold under which conditions. Sometimes these laws are derived directly from mathematical concepts, as is often the case with the type classes themselves. For example, the `Eq` type class, which represents equality of data types, should satisfy the mathematical relationship of symmetry on the equality operator ($a = b \iff b = a$). Other times the laws are imposed by the designer of the type class in order to give some guarantees when using it, such as the associativity property of the `>>=` operator from the `Monad` type class.

Type class laws must be satisfied in order for the code to behave as expected. However, as they are not enforced by Haskell in any way, it has always been the responsibility of the developer to make sure that their implementations do so. Traditionally, formal proofs of this has been done by pen and paper where the programmer systematically unwraps the definition of the function. This approach does have a few noteworthy drawbacks:

- Requires specialized expertise
- Prone to human error
- Time consuming to write, often requiring pages for all but the most trivial proofs
- Often not trivial to do for complex and optimized real-world implementations
- Must be redone when the implementation changes or an optimization is made

There are tools available which assist with this manual process, such as HERMIT [10], but many of the issues still remain. Most developers are therefore content with just testing the implementations with finite input data, since such tests are faster to write and easier to keep up-to-date. One framework for doing this is HUnit [13] which allows unit tests similar to the Java library JUnit [3]. Another framework is QuickCheck [5] which generates input data and attempts to find counterexamples to user-defined properties. While such tests are often enough to convince the programmers of the correctness of their implementations, they also come with some drawbacks:

- They are not as rigorous as proofs - only tests with a finite set of input values
- It is easy to miss critical edge cases
- HUnit tests requires a set of tests for every individual class instance
- QuickCheck tests requires implementations of test data generators, which themselves can contain bugs or miss edge cases

Some developers do not test their implementations at all, which often leads to hard-to-find bugs and unexpected behavior. This is especially true when using higher order functions, which might assume that a function argument satisfies certain properties (*e.g.* associativity).

Wadler and Blott noted in their paper that a more sophisticated type class system might be able to verify class laws. One way of doing this would be to use already existing theorem provers for the proof process, thus relying on the extensive research and work which lies behind these tools. By doing so, all drawbacks listed above regarding manual proofs and HUnit/QuickCheck testing could be avoided. However, few theorem provers support type classes natively, and those who do are not integrated well with the Haskell language. Also, proving properties about functional programs in general has previously been hard to do in practice. Such programs often make use of recursive data structures, and proving properties for that kind of structure usually requires induction. One problem with inductive proofs is to decide how and when to apply induction and, perhaps more importantly, such proofs often requires the use of auxiliary lemmas which need to either be provided by the user or discovered some other way by the theorem prover.

We have seen interest in this field as prior work has already been done towards testing and verification of type class laws in Haskell. For example, Jeuring, Jansson and Amaral describe in their paper [14] a method for testing type class laws using QuickCheck. They develop a framework for expressing type class laws as QuickCheck properties, which can then be tested with generated input data from the class instances. While this is definitely a step in the right direction, they do note the possibility of integrating their work with a proof checker such as the Haskell Inductive Prover [20] (now a part of HipSpec). Another paper by Andrew Farmer [10] describes a tool called HERMIT. The tool is able to prove various properties of Haskell programs, and even transform them in order to increase performance. It has the ability to use structural induction in order to prove type class laws as well. There is also Isabelle [26], an interactive theorem prover with support for type classes. While not directly related to Haskell, it uses higher-order logic that is very similar to many functional programming languages.

One theorem prover which both supports inductive proofs and the Haskell language is HipSpec, presented in the paper "Automating Inducive Proofs using Theory Exploration" by Claessen et al. [6]. HipSpec uses theory exploration[1] to find lemmas to use for proving otherwise difficult properties required by many inductive proofs. It integrates very well with the Haskell language since it can translate Haskell code into an intermediate language called TIP [7]. Code written in the TIP language can in turn be translated to the input language of different external theorem provers, which allows us to use the strengths of different provers for completing proofs. A description of HipSpec and related topics can be found in chapter 2.

While HipSpec is a good option for proving properties about functional programs, the system does not currently have any support for type classes. Our hypothesis is that if such support were to be added to HipSpec we could use it to prove type class laws as well as give developers useful output in the case of implementation errors. This thesis presents our work towards verifying that hypothesis. The intent is to aid developers in writing type class instances which adhere to these laws, without forcing them to do any manual proofs or considering every possible edge case. We will describe the implementation of this extension in detail in chapter 3 and evaluate the extended HipSpec with some examples of both built-in and user-defined type classes in chapter 4. A comparison between the extended HipSpec and some of the other available systems mentioned above is presented in chapter 5.

We hope that this thesis will shed some light on type class laws and the process of automating proofs for them. We also hope that it contributes toward making Haskell an even better and safer language to work with.

[1]An automated system for finding auxiliary lemmas which will be covered in more detail in section 2.2

2. Background

Some background knowledge is required when reading the subsequent chapters about our extension to HipSpec. In this chapter, we give a brief overview of the systems and languages used in the implementation as well as other information needed in order to follow the rest of this thesis.

Since our goal is to automate the process of proving type class laws, this chapter starts with an in-depth explanation of the type class system as implemented in Haskell. We will also give an explanation of how HipSpec works as well as the TIP language used by HipSpec to formulate first-order problems and communicate with external theorem provers. Finally there will be a summary of the compilation process of GHC[1], as well as the GHC Core language which HipSpec uses when translating Haskell code into TIP code.

2.1 Type Classes in Haskell

Type classes are a solution for allowing overloaded functions in Haskell - so called "ad hoc polymorphism". Before type classes were introduced, there were a few other approaches to this problem. One approach (which is used in Standard ML) is to overload basic operators while functions written in terms of them are not. Another approach is to generate one function definition for each possible combination of types the arguments could have. The traditional way for object-oriented programming languages to handle this is by including methods inside the objects (*e.g.* an equality method for comparison).

As mentioned in chapter 1, the concept of type classes was introduced by Philip Wadler and Stephen Blott in 1988 [25]. Type classes are a collection of abstract class methods which (generally) do not have any implementation. Users can make a data type an instance of a type class by providing implementations for the associated class methods. The used type classes can be seen in the type signature of the function as *class constraints*. This system was created for Haskell and has been included in the Haskell language since the very first version [18].

For example, consider the built-in Haskell type class for equality:

```haskell
class Eq a where
    (==) :: a -> a -> Bool
    (/=) :: a -> a -> Bool

    x == y = not (x /= y)
    x /= y = not (x == y)
```

This class creates two *abstract* functions, *i.e.* functions which have a type signature but can have different implementations depending on the instantiating type. In order to define equality for a data type in Haskell, we make it an instance of the `Eq` type class and as such provide concrete implementations to the abstract class methods. Note that in this case the class has default implementations for both class methods but since they are defined in terms of each other we need to override at least one of them when instantiating the class.

[1]The Glasgow Haskell Compiler

```
data Color = Red | Blue

instance Eq Color where
    Red  == Red  = True
    Blue == Blue = True
    _    == _    = False
```

When a function needs to use equality between polymorphic data types, a constraint for `Eq` is specified in the type signature. This allows the function to use any method from the `Eq` type class without knowing the actual implementation beforehand.

```
refl :: Eq a => a -> Bool
refl a = a == a
```

Type classes can also have super classes. In order to make a data type an instance of such a class, one must also make it an instance of the super class. Class methods from the super class can then be used in all instance implementations.

```
class Eq a => Ord a where
    (<=) :: a -> a -> Bool
```

Constraints can also be applied on type variables in the instance declaration. This notion means that the data type is an instance of the class if (and only if) the constraint is satisfied for those type variables.

```
data Maybe a = Just a | Nothing

instance Eq a => Eq (Maybe a) where
    Just a  == Just b  = a == b
    Nothing == Nothing = True
    _       == _       = False
```

Such instances can be seen as a family of monomorphic instances (one for each type `a`). How this works will be described in more detail in section 2.4.3.

2.1.1 Type Class Laws

As mentioned in chapter 1, some type classes have associated type class laws. These are properties which must hold for the class methods of all instances in order to behave as expected. However, the class laws are implicit and not enforced in any way by GHC or other Haskell compilers. The associated laws for the built-in type classes are presented in the Haskell 2010 Language Report [12]. This document unfortunately does not cover all possible class laws for the built-in classes. An example of this is the `Eq` type class, where one would expect the following laws (from the equivalence relation in mathematics) to hold:

Laws for `Eq`

Reflexivity	$a = a$
Symmetry	$a = b \iff b = a$
Transitivity	$a = b \land b = c \implies a = c$

These are not explicitly stated in any official documentation.

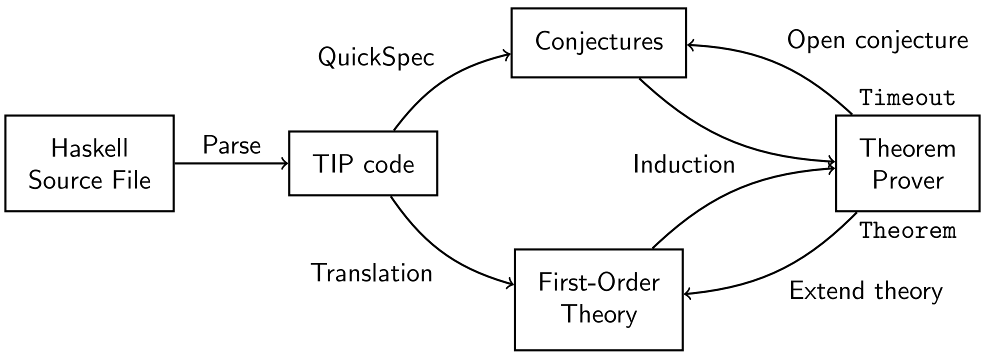
In this thesis, we will explore a way to automate the process of proving class laws for both built-in and user-defined type classes.

2.2 HipSpec

HipSpec [6] is an inductive theorem prover as well as a theory exploration system. It can read Haskell and TIP (see section 2.3) source files and can prove user-defined properties as well as discover auxiliary lemmas needed during the process. Currently HipSpec is only able to handle terminating total values and as such is unable to reason about infinite data structures and non-terminating functions.

HipSpec builds upon a system previously known as Hip [20], integrating another system called QuickSpec [8] to generate potentially interesting conjectures (see below). An overview of the system can be seen in figure 2.1.

Figure 2.1: An overview of HipSpec



QuickSpec is a tool for generating conjectures from functions using QuickCheck [5]. This is done by constructing a set of possible well-typed terms from the input functions and variables (with a limit on the depth to avoid infinite terms). These terms are then tested for equivalence using QuickCheck by generating a number of random test cases for them using provided test data generators for each type. If two terms evaluate to the same value for all of these cases, it is likely that an equality has been found. For each such pair (t_1, t_2), a conjecture $t_1 = t_2$ (for all free variables of the two terms) is created and returned to the user.

HipSpec reads properties from a source file and uses QuickSpec to generate additional conjectures which it then tries to prove using other external first-order theorem provers (*e.g.* Z3 and Waldmeister) and induction. It sends each property to a prover; if that prover is unable to prove the property using first-order logic HipSpec will attempt to apply inductive reasoning. This is done by splitting the property/lemma into separate proof obligations for the base case (or cases) and for the inductive step. These proof obligations are then sent individually to the external theorem prover. If all cases were successfully proven, the original conjecture is also proven and is put into the pool of known lemmas.

The conjectures are sorted by complexity according to some heuristics and the simpler and more general ones are tried first. If HipSpec is unable to prove the conjecture it will get placed in a backlog and tried again after finding other lemmas. This pattern will continue until either:

a. All user-defined properties have been proven, or

b. HipSpec was unable to prove any more lemmas.

The current version of HipSpec uses GHC to compile Haskell code into an intermediate language called GHC Core (explained in section 2.4). This code is then parsed into another language called TIP (see the following section), which is designed to express properties in first-order logic. This is done by finding all stated properties in the Core code and recursively adding other required assets such as functions or data types as it discovers dependencies. This ensures that unnecessary definitions which are not relevant for the proof will be pruned away to facilitate the external provers. The resulting TIP code can then be converted into the input languages of different theorem provers as needed.

2.3 TIP

TIP (*Tons of Inductive Problems*) [7] is a collection of problems used for benchmarking and comparing different theorem provers. The problems contained in the suite are expressed in a special language, the *TIP language*, based on SMT-LIB [2] with added features such as algebraic data types and lambda expressions. Note that *TIP* is often used interchangeably as referring either to the test suite or the related tools/library. For the rest of this thesis, *TIP* will only be used to refer to the conversion tools, unless stated otherwise.

TIP includes tools that can convert Haskell functions and data types to this language. For a simple example of this conversion look at the following code:

```
reverse :: [a] -> [a]
reverse []     = []
reverse (x:xs) = (reverse xs) ++ [x]
```

```
(define-fun-rec
  (par (a)
    (reverse
       ((x (list a))) (list a)
       (match x
         (case nil (as nil (list a)))
         (case (cons y xs)
           (append (reverse xs) (cons y (as nil (list a)))))))))
```

The generated TIP code has a much less readable syntax than the original Haskell source code. For instance, the code makes heavy use of parentheses and there is no syntactic sugar for features such as top-level pattern matching. However, since it has less complicated syntax (featurewise), it works better as an intermedate language for communicating with other theorem provers.

There is also a module available for Haskell which allows users to express TIP properties using special operators and functions. Most of these are simply redefinitions of the standard logical operators (such as `==` and `&&`). A partial list of them can be seen below.

TIP operator	Logical equivalent
`===`	equality
`=/=`	inequality
`.&&.`	conjunction
`.||.`	disjunction
`==>`	implication

As an example, consider the law `reverse (reverse as) = as`, *i.e.* reversing a list twice will result in the original list. This property can be written in Haskell as follows:

```
prop :: [a] -> Equality [a]
prop as = reverse (reverse as) === as
```

Note the `Equality` type which is one of the types used by the TIP module to represent conjectures. Passing this file to TIP will result in the same property represented in the TIP lanugage:

```
(assert-not
  (par (a) (forall ((bs (list a))) (= bs (reverse (reverse bs))))))
```

The property is written as an `assert-not` statement which informs HipSpec that this is a property we want to prove.

A list of other important keywords (and their meaning) used in this thesis is as follows:

TIP keyword	Description
`assert-not`	property to prove
`assert`	axiom
`define-fun`	function definition
`define-fun-rec`	recursive function definition
`declare-datatypes`	datatype and its constructors
`declare-const`	constant (function with arity 0)
`declare-sort`	uninterpreted sort (datatype)

Chapter 3 will go more in-depth on how these are used in the context of proving type class laws.

2.4 GHC Core

During the compilation process of GHC, Haskell code is translated into an intermediate language called GHC Core [23]. Core is a strongly typed language based on System FC calculus [22], which in turn is an extension of the polymorphic lambda calculus System F discovered independently by Jean-Yves Girard [11] and John C. Reynolds [19]. The language is quite a lot simpler than Haskell (albeit much less readable) and contains no special syntactic sugar since its purpose is mainly to simplify transformations and optimizations of source code. Another benefit of the concise nature of the Core language is that it's easier to do code generation for different target languages (*e.g.* C and assembly).

2.4.1 The Compilation Process

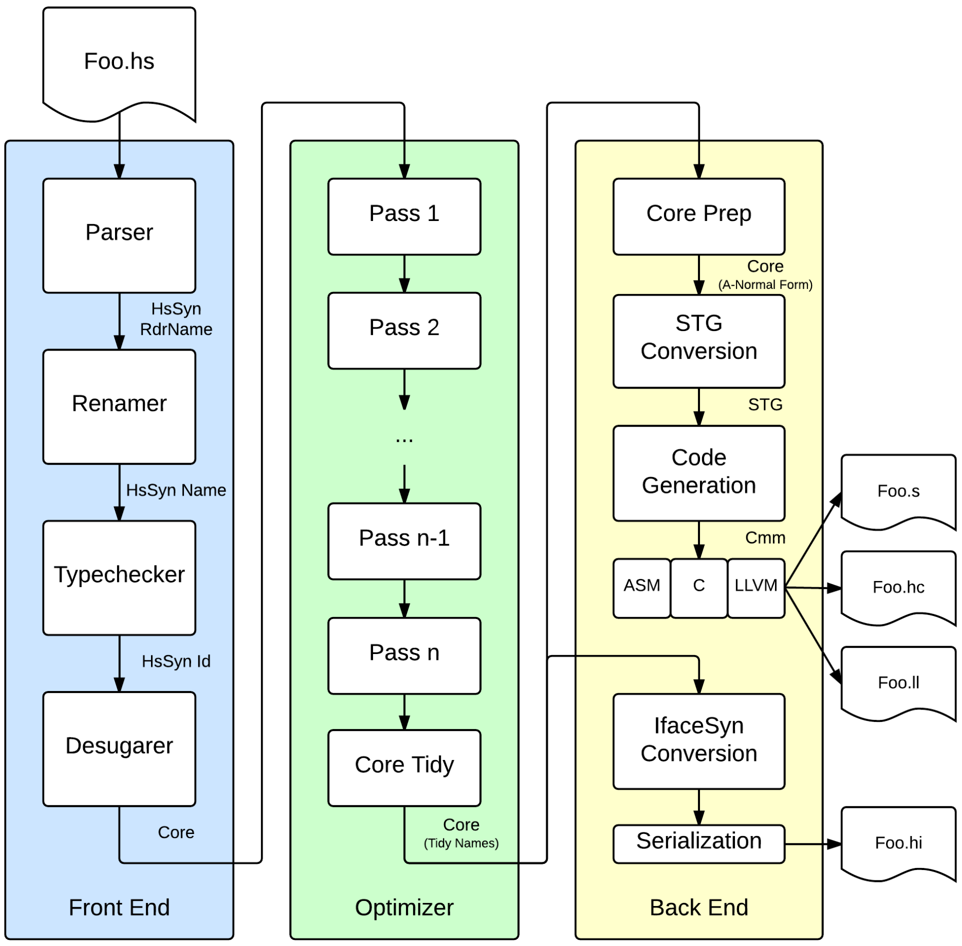
The compilation process consists of three main parts; the front end, the optimizer and the back end [9].

The front end handles the translation from pure Haskell code into GHC Core code. During this step, the code is parsed into an abstract syntax tree (AST) representing the full Haskell syntax. All identifiers are converted into fully qualified names and the resulting tree is then type checked in order to guarantee that no hard crashes will occur at runtime. As a final step, the compiler desugars the AST in order to simplify the expressions and then proceeds to convert it into Core code. The generated code is then sent to the optimizer [17].

The optimizer consists of running the generated Core code through several passes of optimizations. These are dependent on which options are passed to the compiler, but generally include processes like inlining functions, removing dead code etc.

The back end is responsible for translating the Core code into other formats, such as native machine code (for different processor architectures) or C code.

The way the GHC pipeline is structured allows for great flexibility when modifying different aspects of the compiler. For instance, it is possible to add additional passes to the optimization chain, and the code generation could be extended to generate code for any language desired. In this thesis, we will mainly work with the Core code produced by the optimizer since it is this code that is converted into TIP code by HipSpec.



Overview of the GHC compilation process (from [9])

2.4.2 The GHC Core Syntax

In this section, we will give a brief overview of the Core language syntax which will be used in examples in later chapters. For a comprehensive description of the syntax as well as semantics, see [23].

The main relevant features of the GHC Core grammar can be seen in table 2.1.
The notation $\{ \ pattern \ \}^+$ is used to represent one or more repetitions of the inner pattern. Note that unlike Haskell, functions receive type arguments explicitly in the Core language (*i.e.* both type binders and value binders are used in expression application). The same applies to types, which require explicit forall when the type contains polymorphic type variables (type abstraction). Also note that some definitions have been left out for clarity (*e.g.* variable definitions such as *var* and *qvar*).

Table 2.1: Abbreviated GHC Core syntax (from [23])

Value definition	*vdef*	$\rightarrow$	*qvar* :: *ty* = *exp*	
Atomic expression	*aexp*	$\rightarrow$	*qvar*	variable
		|	*qdcon*	data constructor
		|	*lit*	literal
		|	(*exp*)	nested expression
Expression	*exp*	$\rightarrow$	*aexp*	atomic expression
		|	*aexp* { *binder* }$^+$	application
		|	\ { *binder* }$^+$ -> *exp*	abstraction
Argument	*arg*	$\rightarrow$	@ *aty*	type argument
		|	*aexp*	value argument
Binder	*binder*	$\rightarrow$	@ *tbind*	type binder
		|	*vbind*	value binder
Type binder	*tbind*	$\rightarrow$	*tyvar*	
Value binder	*vbind*	$\rightarrow$	(*var* :: *ty*)	
Type	*ty*	$\rightarrow$	*bty*	basic type
		|	%forall { *tbind* }$^+$. *ty*	type abstraction
		|	*bty* -> *ty*	arrow type construction

2.4.3 Type Classes in GHC Core

As mentioned in section 2.4, GHC Core is quite a lot simpler than Haskell in many ways. One of the key differences is that the Core language does not have any notion of type classes. Type classes are instead converted during the desugaring phase of the compiler. For every type class instance present in the compiled source file, a record value (*dictionary*) is generated which contains the identifiers of the class method implementations. Functions which have class constraints in their type signatures are translated into functions which receive dictionaries of suitable type as their first value arguments. When a class method is called inside a function, a lookup is done in the dictionary in order to retrieve the correct identifier for the method implementation.

Dictionary value for the Eq instance of the Color type from section 2.1 (with methods)

```
-- Dictionary value for the "Eq Color" instance
$fEqColor :: Eq Color
$fEqColor = D:Eq (@ Color) $c== $c/=
            └──┘ └───────┘ └───────┘
     constructor    type   instance methods

-- Methods for the "Eq Color" instance (definitions omitted)
$c== :: Color -> Color -> Bool
$c== = \ (a :: Color) (b :: Color) -> ...

$c/= :: Color -> Color -> Bool
$c/= = \ (a :: Color) (b :: Color) -> ...

-- Function using the == method from a dictionary of type Eq
func :: forall a. Eq a -> a -> Bool
func = \ (@ a) ($dEq :: Eq a) (x :: a) -> == (@ a) $dEq x x
               └────────────┘            └─────────────┘
            dictionary argument         dictionary lookup
```

When a class has a super class, all dictionaries generated from the class instances contain a reference to the dictionary of corresponding super class instance. An example of this can be seen below.

Dictionary values for the Ord instance of Color with super class Eq

```
-- Dictionary value for the Ord instance, containing $fEqColor
$fOrdColor :: Ord Color
$fOrdColor =
  D:Ord (@ Color) $fEqColor $ccompare $c< $c<= $c> $c>= $cmax $cmin
  └───┘ └───────┘ └───────┘ └──────────────────────────────────────┘
constructor type  super class            instance methods
```

When an instance definition has a class constraint on a type variable, the dictionary generated from that instance takes as argument a dictionary of same type as the constraint. All class methods will also receive the same dictionary argument in case it is needed in the function definition. For example, in order to get a dictionary for the `Maybe` instance shown in section 2.1, a dictionary for another `Eq` instance must be provided as argument.

Dictionary value for the Eq type class for the Maybe instance

```
$fEqMaybe :: forall a. Eq a -> Eq (Maybe a)
$fEqMaybe = \ (@ a) ($dEq :: Eq a) ->
    D:Eq (@ (Maybe a)) ($c== (@ a) $dEq) ($c/= (@ a) $dEq)
    └──┘ └───────────┘ └─────────────────────────────────┘
constructor   type              instance methods
```

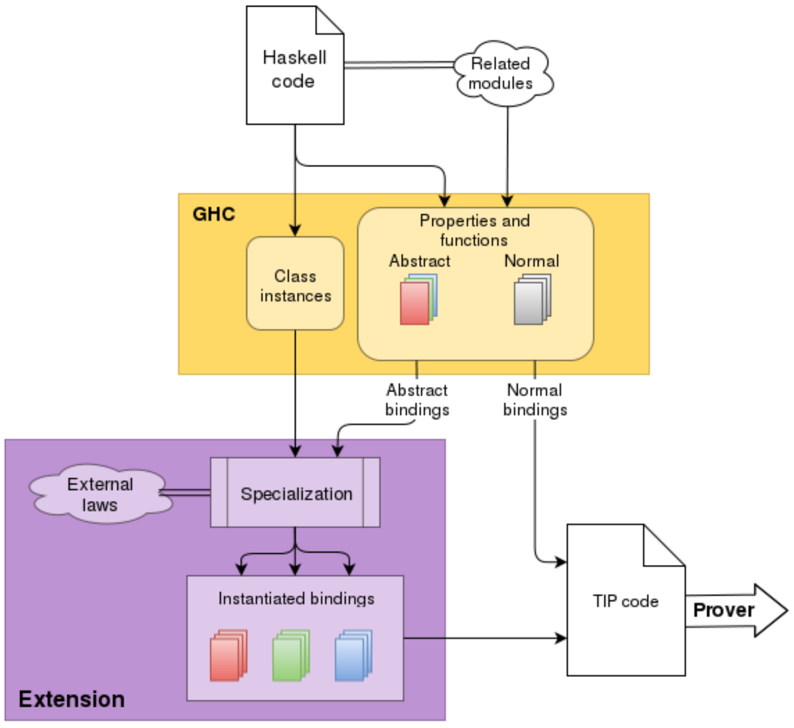
In the next chapter, we will describe how we use the generated Core code in the implementation to instantiate the type class laws and constrained functions.

3. Implementation

We have extended HipSpec with support for expressing and instantiating type class laws. We have also added a set of predefined laws for common built-in type classes (*e.g.* `Eq` and `Monoid`) which can be automatically included in the proof when a file contains instances of those type classes. This chapter explains the implementation of this extension in detail.

The process of proving type class laws can be divided into a series of steps:

1. Specifying the laws in a Haskell source file.
2. Extracting type classes and class instance information.
3. Creating specialized versions of the class laws.
4. Sending generated TIP code to prover (or output to external file).



3.1 Expressing Type Class Laws in Haskell

Type class laws are expressed as normal TIP properties using much the same syntax as presented in section 2.3. The difference is that these laws use abstract functions, *i.e.* functions which need to be supplied concrete class instances in order to be fully defined. This is reflected in the type signature of the law, which contains type variables with class constraints. These laws will give rise to multiple different specialized versions, one for each individual instance of that class. This way, all used functions will be completely defined and the specialized TIP property can be treated as any other property.

Code 3.1: The Monoid type class

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

Consider the `Monoid` type class (with instances such as lists and tuples) presented in code 3.1. Associated laws for the `Monoid` type class are:

Left identity:	`mappend mempty x = x`
Right identity:	`mappend x mempty = x`
Associativity:	`mappend x (mappend y z) = mappend (mappend x y) z`

These can be expressed in Haskell using the TIP module as shown in code 3.2.

Code 3.2: Laws for Monoid type class expressed in Haskell

```haskell
leftIdentity :: Monoid a => a -> Equality a
leftIdentity x = mappend mempty x === x

rightIdentity :: Monoid a => a -> Equality a
rightIdentity x = mappend x mempty === x

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

Note the inclusion of a `Monoid` constraint on the type variables inside the function type signatures. This is the usual way of constraining type variables in Haskell and we use it to also inform HipSpec that they are to be specialized for each instance of the type class. Of course, these could theoretically be inferred by GHC and as such omitted from the code if desired. However, it is still recommended to write them explicitly when defining TIP properties in order to avoid GHC assigning a more general type signature. This happens, for instance, when the property is using methods from the super class only. Since we work with type class laws that are related to a single type class, we currently only allow one class constraint in the type signatures of the properties. While it would be possible to support multiple constraints for expressing relationships between type classes, that purpose is out of scope of this thesis and is left as further work.

In order to avoid forcing the user to rewrite the same laws multiple times, we have included laws for the most common built-in type classes. If a Haskell file contains instances of any such class, the appropriate laws will be automatically included. These laws have been taken from both the Haskell documentation (and the Haskell Report [12]) as well as the field of mathematics. A complete list of all predefined class laws can be found in appendix A.

3.2 Specialization Process

At the time of writing, the current theorem provers used in HipSpec only supports fully polymorphic type variables, *i.e.* type variables which have no constraints on the types they can assume. Since type class laws contains type variables with such constraints they must be converted into specialized versions. The process of generating type-specialized versions of polymorphic functions is called *specialization*. Each constraint on a type variable restricts the possible types that said variable can be substituted with. If multiple class constraints are present, the possible type combinations are the permutations (with repetition) of the instantiated types. Worth noting is that this process is also applied to let-bindings inside the function definitions, since those must also be specialized in order to be useable in the TIP code. Also worth noting is that a property for a class with no instances defined in the target file will not be included in the generated TIP code since we cannot instantiate them.

The algorithm used in the implementation for specializing Core expressions is as follows:

1. **Dictionary extraction and generation**: Find all class instances defined in the file by finding their corresponding dictionary values as well as other related dictionaries defined in the imported modules. Also generate extra dictionaries (*dummy dictionaries*) needed throughout the specialization process.

2. **Monomorphization**: For each definition with a class constraint, check which type variables are constrained and create new definitions for each possible instantiated type among the dictionary values found in the previous step.

3. **Dictionary inlining**: For each monomorphized definition, inline all dictionary arguments by substituting them with dictionaries of the appropriate type.

4. **Resolving dictionary lookups**: Resolve all dictionary lookups by looking inside the dictionary for the correct class method identifier and replace the lookup expression with it.

Each step will be described in more detail in the following sections with some examples of the transformations applied to the GHC Core bindings.[1]

3.2.1 Dictionary Extraction and Generation

Instances of type classes are represented in the Core expression bindings as dictionary values (see section 2.4.3). These contain all the information needed during the specialization process, such as the type classes, instantiating types, class method implementations etc. As an example, consider the list instance shown in code 3.3 for the `Monoid` type class (from code 3.1).

Code 3.3: List instance for the `Monoid` type class

```haskell
data List a = Cons a (List a) | Empty

instance Monoid (List a) where
    mempty = Empty
    mappend Empty ys = ys
    mappend (Cons x xs) ys = Cons x (mappend xs ys)
```

Note that this instance is the minimal definition since it does not implement the `mconcat` method, thus using the default implementation which is defined in terms of the other methods. The Core code for the generated dictionary value is shown in code 3.4. When processing the target Haskell file, we store all such dictionary values for use in the later steps. Since these dictionaries might use other

[1]The GHC Core expressions have been simplified and commented for clarity, and some function definitions are left out to avoid clutter.

Code 3.4: Generated Core code for the `Monoid (List a)` instance

```
$fMonoidList :: forall a. Monoid (List a)
$fMonoidList =
  \ (@ a) -> D:Monoid (@ (List a))
      (Empty (@ a))      :: List a
      ($cmappend (@ a)) :: List a -> List a -> List a
      ($cmconcat (@ a)) :: [List a] -> List a

$cmappend :: forall a. List a -> List a -> List a
$cmappend = ...

$cmconcat :: forall a. [List a] -> List a
$cmconcat = \ (@ a) -> $dmmconcat (@ (List a)) ($fMonoidList (@ a))

-- Default implementation for the mconcat function
$dmmconcat :: forall a. Monoid a -> [a] -> a
$dmmconcat = ...
```

dictionaries (*e.g.* for access to super class methods), we also explore the imported modules and store all the related ones as well. All other (unrelated) dictionaries are thrown away since they are not used during the dictionary inlining process.

As mentioned in section 2.4.3, some class instances have constrained type variables in their type signature. An example of such an instance is the `Monoid` instance for the built-in `Maybe` data type shown in code 3.5, where the inner element of the data type must also be an instance of the `Monoid` type class. During the proof process, we want to prove that such instances satisfies the class laws *assuming* that the

Code 3.5: `Maybe` instance for `Monoid` type class

```
instance Monoid a => Monoid (Maybe a) where
    mempty = Nothing
    mappend Nothing m = m
    mappend m Nothing = m
    mappend (Just m1) (Just m2) = Just (mappend m1 m2)
```

elements also satisfies their associated laws. However, we cannot assert such laws on a fully polymorphic type variable in TIP because that would mean they would hold for all other types as well (including the `Maybe` data type we want to prove them for). Because of this, we create a new monomorphic type (whose name matches the class itself) to make these assertions on (a *dummy type*). We also generate function identifiers for all class methods (specialized to this dummy type) as well as function bindings for all available default class method implementations. From these function identifiers, new dictionaries are generated (*dummy dictionaries*) for each dummy type requiring assertions. Note that, as mentioned in section 2.4.3, some default method implementations may be missing from generated Core code. As such, some dummy function identifiers will lack an implementation, and will be turned into uninterpreted TIP lambda constants (see section 3.3).

In the case of the `Maybe` instance shown above, we need to generate a dummy dictionary for the `Monoid` type class. This dictionary value can be seen in code 3.6. Note that only the `mconcat` function has an actual function binding since it is the only default implementation available (via `$dmmconcat`). We have not been able to force GHC to include all default method implementations in a general way, Code 3.6: Dummy dictionary value for `Monoid` type class

```
Monoid.$fMonoid_dummy :: Monoid Monoid
Monoid.$fMonoid_dummy =
    D:Monoid (@ Monoid)
        Monoid.mempty_dummy  :: Monoid
        Monoid.mappend_dummy :: Monoid -> Monoid -> Monoid
        Monoid.mconcat_dummy :: [Monoid] -> Monoid

Monoid.mconcat_dummy :: [Monoid] -> Monoid
Monoid.mconcat_dummy = $dmmconcat (@ Monoid) Monoid.$fMonoid_dummy
```

but as can be see in chapter 4 the properties can often be proven without them.

Since TIP does not support inheritance of types, we need to generate dummy dictionaries for each child class of the type classes present among the instance constraints. For example, we generate dummy dictionaries for both the `Eq` and `Ord` dummy types for the `Eq` class if we are to prove laws for an `Ord` instance, since we need to resolve functions for both dummy types.

3.2.2 Monomorphization

Monomorphization is the process of generating monomorphic versions of polymorphic functions. We only monomorphize type variables which are constrained by one or more type classes since HipSpec is already able to reason about non-constrained polymorphic types. This is good because the number of possible types for a fully polymorphic type variable is very large (for instance all built-in types).

The algorithm for monomorphizing a Core expression is fairly straightforward:

1. For each type variable constrained by a type class, replace it with the types of all possible instances of that type class defined in the current file.
2. Remove the substituted type variable arguments from the expression while adding any new type variables present in the substituting types.

Going back to the `Monoid (List a)` instance example shown in code 3.3, type variables constrained by the `Monoid` class will be substituted with the ”`List a`” type (and other types if there are any other `Monoid` instances defined in the same file). See code 3.7 for the generated Core expression for the left identity law and the corresponding monomorphized version of it.[2] Note that in this step we have only monomorphized the type; nothing has yet been done with the dictionary arguments or the dictionary lookups. Also note that since the list instance type contains a (fully polymorphic) type variable `a`, the function still receives a type variable as an argument. This variable would otherwise be removed since it would no longer be used in the expression.

We also need to monomorphize the dictionaries if their type signatures contain any constrained type variables. The substituting types in this case are the dummy types generated in the previous step. For instance, consider the generated Core code for the `Monoid Maybe` instance (code 3.5) and its monomorphized version shown in code 3.8. As mentioned in the previous section, in order to make asserts on the (constrained) type variable `a` we need to substitute it with a dummy type (in this case `Monoid`). This substitution is also done in the class methods themselves (they are otherwise already specialized for the instance type). Note that this dictionary still takes a dictionary (of type `Monoid Monoid`) as

[2]Only one law is shown since the other laws are monomorphized in much the same way.

Code 3.7: Monomorphization of left identity monoid law for `List` instance

```
-- Original Core expression
leftIdentity :: forall a. Monoid a -> a -> Equality a
leftIdentity =
  \ (@ a) ($dM :: Monoid a) (x :: a) ->
    === (@ a) (mappend (@ a) $dM (mempty (@ a) $dM) x) x

-- Monomorphized Core expression
leftIdentity :: forall a. Monoid (List a) ->
                               List a -> Equality (List a)
leftIdentity =
  \ (@ a) ($dM :: Monoid (List a)) (x :: List a) ->
    === (@ (List a))
      (mappend (@ (List a)) $dM (mempty (@ (List a)) $dM) x)
      x
```

Code 3.8: Monomorphization of the `Monoid (Maybe a)` dictionary

```
-- Original dictionary
$fMonoidMaybe :: forall a. Monoid a -> Monoid (Maybe a)
$fMonoidMaybe = \ (@ a) ($dM :: Monoid a) ->
  D:Monoid
    (@ (Maybe a))
    ($cmempty (@ a) $dM)  :: Maybe a
    ($cmappend (@ a) $dM) :: Maybe a -> Maybe a -> Maybe a
    ($cmconcat (@ a) $dM) :: [Maybe a] -> Maybe a

-- Monomorphized dictionary
$fMonoidMaybe :: Monoid Monoid -> Monoid (Maybe Monoid)
$fMonoidMaybe = \ ($dM :: Monoid Monoid) ->
  D:Monoid
    (@ (Maybe Monoid))
    ($cmempty $dM)  :: Maybe Monoid
    ($cmappend $dM) :: Maybe Monoid -> Maybe Monoid -> Maybe Monoid
    ($cmconcat $dM) :: [Maybe Monoid] -> Maybe Monoid
```

argument. This is why we need the dummy dictionaries generated in the previous step - we need to use them to inline such dictionaries in order to make a fully specialized dictionary.

When the monomorphization process is completed, all functions will either be monomorphic or fully polymorphic. They will still receive dictionaries as arguments, though now with a known type, which will allow us to inline them in the following step.

3.2.3 Dictionary Inlining

The previous steps have removed all constrained type variables from the dictionaries and functions. Since the types of the dictionaries are now known, we can now inline all dictionary arguments in both dictionary values and function definitions. There exists only one possible dictionary value for each argument. This is due to Haskell not allowing the user to define two class instances (and as such two dictionaries) of the same type; the inclusion of dummy dictionaries does not change this since they all have unique types. Thus the inlining algorithm becomes very simple: for each dictionary argument, find a dictionary value of appropriate type and substitute the argument with it.

As an example, consider the dictionary inlined version of the left identity monoid law for the `List` data type shown in code 3.9. Note that the dictionary argument has been replaced with the dictionary value (`$fMonoidList`). The type of all function definitions and dictionary values are now completely specialized.

Code 3.9: Monomorphization of left identity monoid law for the `List` instance

```
-- Dictionary inlined Core expression
leftIdentity :: forall a. List a -> Equality (List a)
leftIdentity =
  \ (@ a) (x :: List a) ->
    === (@ (List a))
      (mappend (@ (List a)) $fMonoidList
               (mempty (@ (List a)) $fMonoidList)
               x)
      x
```

3.2.4 Resolving dictionary lookups

When all dictionaries have been inlined, we can resolve all dictionary lookups present in the expressions. This is the final step of the specialization process and when finished there will be no notion of dictionaries left in the Core code. The algorithm for resolving dictionary lookups is very simple; traverse the expression syntax trees for lookup expressions, resolve each one by looking inside the dictionary value for the correct identifier for the class method implementation and then substitute the lookup expression with that identifier.

Code 3.10: Resolving dictionary lookups in the left identity monoid law for `List` instance

```
leftIdentity :: forall a. List a -> Equality (List a)
leftIdentity =
  \ (@ a) (x :: List a) ->
    === (@ (List a)) ($cmappend $cmempty x) x
```

3.3 Generation of TIP Code and Proof

As mentioned in section 2.2, HipSpec uses the intermediate language TIP in order to simplify communication with external theorem provers. The process of generating this TIP code has been extended with type classes in mind. During the generation of the internal TIP abstract syntax tree, any known laws for the instantiated classes are retrieved by compiling separate files containing only these laws (defined as TIP properties). These laws are specialized along with the other Core bindings present in the input file (and related module files) using the specialization process described in section 3.2. This allows users to define laws outside the input file and is the way for supplying HipSpec with predefined class laws for built-in type classes.

When the specialization process has been completed, all specialized bindings are renamed to make the output a bit more readable. This is done by prepending the type names of the inlined dictionary arguments to the name of each specialized identifier. After this is done, we begin the TIP code generation process.

Axioms are generated for the properties which have been specialized using dummy dictionaries or dictionaries containing nested dummy types of child classes. All other properties are considered conjectures to be proven by the theorem prover. For example, consider the case where a file contains two instance declarations: `Eq a => Eq (List a)` and `Ord a => Ord (List a)`. The goal is then to prove that each equality law holds for the `List Eq` type (a list with a dummy type representing the `Eq` class) while assuming the equality laws holds when proving the `Ord` laws for the `List Ord` type. The motivation for this is that when proving the class laws for the `Ord` class, we want to only consider potential errors in the instance implementation of that type class. If an error is present in the `Eq` instance of the same type, it would still be desirable to be able to correctly verify the `Ord` class laws despite this.

All dummy types generated during the specialization process (see section 3.2.1) are represented as uninterpreted sorts with arity zero. This allows for the inclusion of data types in the TIP language without providing any information about how they are constructed, which is fine because we only need to make assertions on them. The corresponding dummy class methods are represented as uninterpreted lambda function constants if there are no default implementations available for them.

For example, consider the `Maybe` instance of the `Monoid` type class presented in code 3.5 and the generated TIP code seen in code 3.11. Since the instance requires type variable `a` to be an instance of the `Monoid` type class, we generate a sort using `declare-sort` as well as dummy functions for the `mempty` and `mappend` functions in order to make assertions on them. Other functions of the `Monoid` type class are pruned away since they are not used in the properties and as such not are not relevant for the proof. A data type for `Maybe` is generated as well as the three different laws defined with `assert-not` and the axioms with `assert`. As can be seen on line 25 and 26 the identity laws have been simplified by HipSpec turning them into identical conjectures. This makes it difficult to differentiate the two laws which is an issue that will be discussed further in chapter 6.

After the TIP code has been successfully generated, HipSpec converts it to the native input language of the used external theorem prover and begins the proof process.

Code 3.11: Generated TIP code for the `Monoid` type class for polymorphic data type `Maybe a`

```
; Dummy types and their corresponding methods
(declare-sort Monoid 0)
(declare-const Monoid.mempty_dummy Monoid)
(declare-const Monoid.mappend_dummy (=> Monoid (=> Monoid Monoid)))

; All related data types and functions
(declare-datatypes (a)
  ((list (nil) (cons (head a) (tail (list a))))))
(declare-datatypes (a) ((Maybe (Just (Just_0 a)) (Nothing))))
(define-fun
  MaybeMonoid.mappend
    ((x (Maybe Monoid)) (y (Maybe Monoid))) (Maybe Monoid)
    (match x
      (case (Just z)
        (match y
          (case (Just x2) (Just (@ (@ Monoid.mappend_dummy z) x2)))
          (case Nothing x)))
      (case Nothing y)))

; Laws to be proven
(assert-not
  (forall ((x (Maybe Monoid)) (y (Maybe Monoid)) (z (Maybe Monoid)))
    (= (MaybeMonoid.mappend x (MaybeMonoid.mappend y z))
      (MaybeMonoid.mappend (MaybeMonoid.mappend x y) z))))
(assert-not (forall ((x (Maybe Monoid))) (= x x)))
(assert-not (forall ((x (Maybe Monoid))) (= x x)))

; Axioms for dummy types
(assert
  (forall ((x Monoid) (y Monoid) (z Monoid))
    (= (@ (@ Monoid.mappend_dummy x)
          (@ (@ Monoid.mappend_dummy y) z))
      (@ (@ Monoid.mappend_dummy (@ (@ Monoid.mappend_dummy x) y)) z)
    )))
(assert
  (forall ((x Monoid))
    (= (@ (@ Monoid.mappend_dummy Monoid.mempty_dummy) x) x)))
(assert
  (forall ((x Monoid))
    (= (@ (@ Monoid.mappend_dummy x) Monoid.mempty_dummy) x)))

(check-sat)
```

4. Evaluation

In order to evaluate the tool properly, a few different aspects need to be taken into consideration. Perhaps the most important one is the **coverage** of the problem space, *i.e.* what different type class instances the extended HipSpec is able to successfully prove the associated class laws for and where it encounters problems. Another area of interest is how much **time** it takes for the proofs to complete. Proofs generally needs to be done only once (unless the implementation changes) but a faster proof time is nonetheless desirable. The proof might need to be rerun with other parameters and for some applications (*e.g.* safety critical) one may want to recheck proofs done by someone else (especially since HipSpec currently does not have a standardized output format). The last metric we will consider is the **output** from the tool; a descriptive output outlining the manual induction steps allows the user to gain a better understanding of the proof steps. Also, should the tool fail to complete the proof, a nice output could help the user to complete the proof manually using the successful steps from the tool.

The examples presented in this chapter have been chosen to test various situations which a developer might encounter when working with type classes. They include instances of algebraic structures for testing relationships between data types, as well as built-in type classes for more common real-world applications. Note that the proof time is calculated from the start of the tool until termination (*i.e.* when it cannot prove any more lemmas). HipSpec may be able to prove the user-defined properties early on and then spend some extra time to process generated lemmas. We will highlight such cases when they occur.

For the sake of readability, type class laws are presented using standard first-order logic (where applicable). Also, we will not include the complete proof output for all examples since they can be quite long. Both the code and the examples are available online[1][2] and can be used to reproduce the output if desired.

In section 4.2 we will introduce the various data types used in the evaluation examples. In section 4.3 and 4.4 we will present and discuss the examples and their individual results. Finally, we will summarize all evaluation results and discuss them with a broader perspective in section 4.5. A full comparison between the results presented in this chapter and other systems will be outlined in chapter 5.

4.1 Experimental Setup

These are the specifications of the system used to evaluate HipSpec in this chapter.

Processor	Intel Core i7-3630QM @ 2.40 GHz
Memory	8 GB
Operating System	Arch Linux
GHC version	7.10.3

The flags used to HipSpec was `p=z` to use Z3 as the prover, `v=2` to allow simultaneous induction over two variables (needed for some cases) as well as the `law-dir` flag (added with our extension) with the path to a directory with the files containing the predefined laws.

[1]Code: https://github.com/chip2n/tools

[2]Examples: https://github.com/chip2n/hipspec-typeclasses

4.2 Data Types

We have chosen a few different custom data types which will be used throughout our evaluation examples.

Nat: A recursive definition of natural numbers, which happens to be an instance of many mathematical classes.

Matrix2: Data type representing a 2x2 (square) matrix, which contains multiple fully polymorphic type variables. Due to the nature of the type system in Haskell, they cannot easily be generalized into arbitrary-sized square matrices. It would require storage of the size together with lists, which would be hard for theorem provers to reason about. Because of this, we have chosen to go with fixed size 2x2 matrices.

List: A recursive definition of lists, which also contains fully polymorphic type variables.

Tree: A recursive definition of binary trees, which is a more complex structure than the aforementioned lists since it branches twice at each internal node.

The Haskell implementation of these are shown in code 4.1. Due to the special syntax (`[a]`) of lists preventing us from redefining (and subsequently testing) list instances of the built-in type classes, we have chosen to define a custom list data type. We also use built-in booleans and the `Maybe` data type in some examples but we have chosen not to define them ourselves.

Code 4.1: Data structures used in evaluation files

```haskell
-- Natural numbers
data Nat = Zero | Succ Nat

-- 2x2 matrix
data Matrix2 a = Matrix2 a a a a

-- List
data List a = Empty | Cons a (List a)

-- Binary tree
data Tree a = Leaf a | Branch (Tree a) (Tree a)
```

4.3 Built-in Type Classes

This section contains examples of proving class laws for built-in type classes in Haskell. The aim is to test the tool in some common situations that the everyday Haskell developer might find themselves in. These do not require any properties to be defined manually by the user as they are pre-written and included in the source code of HipSpec. A list of them can be found in appendix A.

Note that the examples in this section uses the actual built-in definitions of the type classes. The definitions presented might differ in some minor way from these (for example by not including default method definitions) and as such they should only serve as a quick reference.

4.3.1 Class: Eq

The `Eq` type class defines equality and inequality between data types. While the Haskell 2010 report does not mention any associated laws, we would expect the `==` method to have the same properties as the mathematical equivalence relation.

Laws for `Eq`

1	$a = a$	Reflexivity
2	$a = b \iff b = a$	Symmetry
3	$a = b \wedge b = c \implies a = c$	Transitivity

Note that the first law (reflexivity) does not completely hold for some built-in data types, for example the special case of `NaN` for the `Float` data type. Despite this we have chosen to include reflexivity as a law since it would be expected in almost every other case.

```
class Eq a where
    (==) :: a -> a -> Bool
    (/=) :: a -> a -> Bool
```

Instance: Natural Numbers

Equality between natural numbers is defined by recursively comparing their data constructors.

```
instance Eq Nat where
    Zero      == Zero      = True
    (Succ a)  == (Succ b)  = a == b
    _         == _         = False
```

Instance: Maybe

Equality between `Maybe` data types is defined by comparing the elements for equality. This requires an `Eq` class constraint on the element type inside the data structure.

```
instance Eq a => Eq (Maybe a) where
    Nothing   == Nothing   = True
    (Just a)  == (Just b)  = a == b
    _         == _         = False
```

Instance: 2x2 Matrices

Equality between matrices is defined by comparing each element for equality. This requires an `Eq` class constraint on the element type inside the data structure.

```
instance Eq a => Eq (Matrix2 a) where
    Matrix2 a11 a12 a21 a22 == Matrix2 b11 b12 b21 b22 =
      a11 == b11 && a12 == b12 && a21 == b21 && a22 == b22
```

Instance: Lists

Equality between lists is defined by comparing each element for equality recursively. This requires an `Eq` class constraint on the element type inside the data structure.

```
instance Eq a => Eq (List a) where
    Empty        == Empty        = True
    (Cons a as) == (Cons b bs) = a == b && as == bs
    _            == _            = False
```

Instance: Binary Tree

Equality between binary trees is defined by comparing each element for equality recursively. This requires an `Eq` class constraint on the element type inside the data structure.

```
instance Eq a => Eq (Tree a) where
    Leaf x            == Leaf y            = x == y
    (Branch x1 x2) == (Branch y1 y2) = x1 == y1 && x2 == y2
    _                 == _                 = False
```

Results: Eq

Instance	1	2	3	**Time**
Nat	✓	✓	✓	1.3 s
Maybe	✓	✓	✓	1.4 s
Matrix2	✓	✓	✓	3.0 s
List	✓	✓	✓	3.1 s
Tree	✓	✓	✓	3.0 s

There is not really much to say about the `Eq` instances as they are quite simple. The proofs are quick and straightforward, and there are no issues during the process.

4.3.2 Class: Ord

The `Ord` type class defines total order between data types. While the Haskell 2010 report does not mention any associated laws, we would expect the methods to have the same properties as the mathematical relations $<$, $\leq$, $>$ and $\geq$.

Laws for `Ord`

1	$a \leq b \wedge b \leq c \implies a \leq c$	($\leq$) Transitivity
2	$a \leq b \wedge b \leq a \implies a = b$	($\leq$) Antisymmetry
3	$a \leq b \vee b \leq a$	($\leq$) Totality
4	$a \leq b \implies a < b \vee a = b$	($\leq$) Definition
5	$a \geq b \wedge b \geq c \implies a \geq c$	($\geq$) Transitivity
6	$a \geq b \wedge b \geq a \implies a = b$	($\geq$) Antisymmetry
7	$a \geq b \vee b \geq a$	($\geq$) Totality
8	$a \geq b \implies a > b \vee a = b$	($\geq$) Definition
9	$a < b \wedge b < c \implies a < c$	($<$) Transitivity
10	$\neg(a < b \wedge b < a)$	($<$) Antitotality
11	$a < b \implies a \leq b \wedge a \neq b$	($<$) Definition
12	$a > b \wedge b > c \implies a > c$	($>$) Transitivity
13	$\neg(a > b \wedge b > a)$	($>$) Antitotality
14	$a > b \implies a \geq b \wedge a \neq b$	($>$) Definition

Note that we need to define laws for all class methods, as opposed to only defining them for the methods present in the minimal definition of the class. These laws cannot be discovered during the theory exploration process since they contain implications and QuickSpec is currently only able to generate equations automatically. They are however needed as lemmas when proving some of the other laws, for example the list instance shown below which uses the $<$ operator on the contained elements.

All instances of `Ord` must also be instances of the `Eq` type class. We will use the `Eq` instances from the previous section. Note that here we have defined the `Ord` instances in a separate module so that we will only test those laws and not the laws associated with the `Eq` instances (which will instead be imported as axioms).

```haskell
class Eq a => Ord a where
    compare :: a -> a -> Ordering
    (<)     :: a -> a -> Bool
    (<=)    :: a -> a -> Bool
    (>)     :: a -> a -> Bool
    (>=)    :: a -> a -> Bool
    max     :: a -> a -> a
    min     :: a -> a -> a
```

Instance: Natural Numbers

Ordering between natural numbers is defined by recursively comparing their data constructors.

```haskell
instance Ord Nat where
    Zero     <= _        = True
    (Succ a) <= Zero     = False
    (Succ a) <= (Succ b) = a <= b
```

Instance: Maybe

Ordering between `Maybe` data types is defined by comparing the order of the contained elements. This requires an `Ord` class constraint on the element type inside the data structure.

```
instance Ord a => Ord (Maybe a) where
  Nothing  <= _          = True
  (Just a) <= Nothing    = False
  (Just a) <= (Just b)  = a <= b
```

Instance: Lists

Ordering between lists is defined by lexiographically comparing each element recursively. This requires an `Ord` class constraint on the element type inside the data structure.

```
instance Ord a => Ord (List a) where
    Empty       <= _             = True
    (Cons a as) <= Empty         = False
    (Cons a as) <= (Cons b bs)
        | a == b     = as <= bs
        | otherwise = a < b
```

Instance: Binary Tree

There are several possible ways to order binary trees; we have chosen to define it by lexiographically comparing the nodes from left to right.

```
instance Ord a => Ord (Tree a) where
    (Leaf _)      <= (Branch _ _)    = True
    (Leaf x)      <= (Leaf y)        = x <= y
    (Branch _ _) <= (Leaf _)        = False
    (Branch l r) <= (Branch l' r') = l == l' && r <= r' || l <= l'
```

Results: Ord

Instance	1	2	3	4	5	6	7	8	9	10	11	12	13	14	Time
Nat	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	36.9 s
Maybe	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	108.8 s
List	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	150.1 s
Tree	⊗	⊗	⊗	⊗	⊗	⊗	⊗	⊗	⊗	⊗	⊗	⊗	⊗	⊗	-

Proving the `Ord` instances takes quite a bit longer to prove than for `Eq`. This seems to be due to the increased number of class methods as this increases the time taken for the theory exploration to generate conjectures. The `Tree` instance generates so many terms that it could not be proven at all; HipSpec runs out of memory during theory exploration.

4.3.3 Class: Monoid (Built-in)

A monoid is a set of elements with a single associative binary operator ($\cdot$), called `mappend` in Haskell, and an identity element (ϵ), called `mempty`.

Laws for monoids

1	$\epsilon \cdot a = a$	Left identity
2	$a \cdot \epsilon = a$	Right identity
3	$(a \cdot b) \cdot c = a \cdot (b \cdot c)$	Associativity

The type class as defined in the Haskell standard library also includes a concatenation function (`mconcat`). We do not specify any laws for this function since it is rarely overridden. Note that if the user *does* override `mconcat` (*e.g.* for performance reasons), they are free to write their own properties.

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

Instance: Natural Numbers

The monoid instance for natural numbers is defined either by addition (with 0 as identity element) or multiplication (with 1 as identity element).

```
instance Monoid Nat where
    mempty  = Zero
    mappend = natAdd

-- Natural numbers addition
natAdd :: Nat -> Nat -> Nat
natAdd Zero a     = a
natAdd (Succ a) b = Succ (natAdd a b)
```

```
instance Monoid Nat where
    mempty  = Succ Zero
    mappend = natMul

-- Natural numbers multiplication
natMul :: Nat -> Nat -> Nat
natMul Zero m     = Zero
natMul (Succ n) m = natAdd m (natMul n m)
```

Instance: Maybe

The monoid instance for the `Maybe` data type is defined by applying the monoid operator to the inner monoidic elements. This requires a `Monoid` class constraint on the element type inside the data structure.

```
instance Monoid a => Monoid (Maybe a) where
    mempty = Nothing
    mappend Nothing m = m
    mappend m Nothing = m
    mappend (Just m1) (Just m2) = Just (mappend m1 m2)
```

Instance: Matrix2

The monoid instance for the `Matrix2` data type is defined by the matrix multiplication operator with the identity matrix as identity element. Note that the matrix multiplication operator requires a custom type class called Semiring (hence the class constraint on the element type). The declaration of this type class can be seen in section 4.4.3.

```
instance Semiring a => Monoid (Matrix2 a) where
    mempty  = Matrix2 one zero zero one
    mappend = matrix2Mul

-- 2x2 Matrix multiplication
matrix2Mul :: Semiring a => Matrix2 a -> Matrix2 a -> Matrix2 a
matrix2Mul (Matrix2 a11 a12 a21 a22) (Matrix2 b11 b12 b21 b22) =
    Matrix2 (add (mul a11 b11) (mul a12 b21))
            (add (mul a11 b12) (mul a12 b22))
            (add (mul a21 b11) (mul a22 b21))
            (add (mul a21 b12) (mul a22 b22))
```

Instance: Lists

The monoid instance for lists is defined by the list concatenation operation with the empty list as identity element.

```
instance Monoid (List a) where
    mempty = Empty
    mappend Empty       a = a
    mappend (Cons a as) b = Cons a (mappend as b)
```

Results: Monoid (Built-in)

Instance	1	2	3	**Time**
Nat (*add*)	✓	✓	✓	4.3 s
Nat (*mul*)	✓	✓	✓	27.3 s
Maybe	✓	✓	✓	2.1 s
Matrix2	✓	✓	✓	2.7 s
List	✓	✓	✓	1.5 s

`Monoid` is another fairly simple class without too many issues. However, note that the `Nat` instance with multiplication as the operator takes about five times longer to prove than the other instances. This is mainly due to the fact that it is a much larger definition containing several different functions; for instance HipSpec needs to discover and prove properties about addition as well. In the case of the addition instance, the user-defined law is proven almost immediately but HipSpec still tries to prove the generated conjectures as well. If the process could finish when the user-defined property is proven it would lead to a significantly improved proof time (more in line with the list and matrix instance).

4.3.4 Class: Functor

A functor is a set of elements which can be mapped over.

Laws for functors

1	`fmap id = id`	Identity
2	`fmap (f . g) = fmap f . fmap g`	Composition

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

Instance: Lists

The functor instance for lists is defined by applying the function to each element in the data structure.

```
instance Functor List where
    fmap f Empty = Empty
    fmap f (Cons a as) = Cons (f a) $ fmap f as
```

Results: Functor

Instance	1	2	**Time**
List	⊗	⊗	-

Unfortunately, this cannot (yet) be proven by HipSpec. The translation between HipSpec and QuickSpec does not yet support partial application of function arguments, which is used in both class laws.

This means that HipSpec is unable to prove instances of classes like `Monad`) as well. We have therefore decided to not include such evaluation examples.

4.4 User-defined Type Classes

In this section we will evaluate examples of some user-defined type classes, most of which are taken from the field of mathematics and represented in Haskell. These examples contain interesting relationships between them which will serve to test the capabilities of our implementation. Furthermore, it is a potentially interesting use case for HipSpec to be able to aid in the proof process outside of the field of functional programming.

4.4.1 Class: Semigroup

A semigroup is a set of elements with a single associative binary operator $(\cdot)$.

Laws for semigroups

1	$(a \cdot b) \cdot c = a \cdot (b \cdot c)$	Associativity

```haskell
class Semigroup a where
    op :: a -> a -> a
```

Instance: Natural numbers

Natural numbers form a semigroup under both addition and multiplication since both those operators are associative.

```haskell
instance Semigroup Nat where
    op = natAdd

-- Natural numbers addition
natAdd :: Nat -> Nat -> Nat
natAdd Zero a     = a
natAdd (Succ a) b = Succ (natAdd a b)
```

```haskell
instance Semigroup Nat where
    op = natMul

-- Natural numbers multiplication
natMul :: Nat -> Nat -> Nat
natMul Zero m     = Zero
natMul (Succ n) m = natAdd m (natMul n m)
```

Instance: Maybe

The `Maybe` data type form a semigroup by applying the operator on the inner element, provided that it is an instance of Semigroup as well.

```haskell
instance Semigroup a => Semigroup (Maybe a) where
    Nothing  `op` m        = m
    m        `op` Nothing  = m
    (Just a) `op` (Just b) = Just (a `op` b)
```

Instance: 2x2 Matrices

Square matrixes form a semigroup under matrix multiplication as long as the contained elements form a semiring (see section 4.4.3). The semiring instance is required since it provides addition and multiplication of the elements which matrix multiplication uses. Therefore, we add a class constraint in the instance definition of the semigroup instance.

```
instance Semiring a => Semigroup (Matrix2 a) where
    op = matrix2Mul
```

Instance: Lists

Lists form a semigroup under list concatenation.

```
instance Semigroup (List a) where
    Empty        `op` a = a
    (Cons a as) `op` bs = Cons a (as `op` bs)
```

Results: Semigroup

Instance	1	**Time**
Nat (*add*)	✓	4.0 s
Nat (*mul*)	✓	21.4 s
Maybe	✓	1.3 s
Matrix2	✓	2.2 s
List	✓	1.3 s

These results are as expected since we have already proved the same law for most instances in the `Monoid` example of the previous section. Worth noting is that the theory exploration step discovers a few interesting properties about these instances. For example, both left and right identity properties are found for both addition and multiplication of natural numbers as well as concatenation of lists. This means that these instances are, as expected, monoids (see next section).

4.4.2 Class: Monoid (User-defined)

A monoid is a set of elements with a single associative binary operator $(\cdot)$ and an identity element (ϵ).

Laws for monoids

1	$\epsilon \cdot a = a$	Left identity
2	$a \cdot \epsilon = a$	Right identity

Monoids are included in Haskell as a built-in type class which we have evaluated in section 4.3.3. Here we use a more mathematically flavored definition which uses the `Semigroup` type class (from section 4.4.1) as a super class. We can therefore skip the law for associativity (from section 4.3.3) when defining the properties in Haskell, since that law is already provided by the Semigroup declaration.

```
class Semigroup a => Monoid a where
    identity  :: a
```

Instance: Natural Numbers

Natural numbers form a monoid under both addition (with identity element 0) and multiplication (with identity element 1). Note that the following examples import different instances of the `Semigroup` type class which provides addition and multiplication respectively.

```
-- Uses semigroup instance for Nat with addition
instance Monoid Nat where
    identity = Zero
```

```
-- Uses semigroup instance for Nat with multiplication
instance Monoid Nat where
    identity = Succ Zero
```

Instance: Maybe

The monoid instance for the `Maybe` data type is defined by applying the monoid operator to the inner monoidic elements with `Nothing` as identity element. This requires a `Semigroup` class constraint on the element type inside the data structure (from the super class instance).

```
instance Semigroup a => Monoid (Maybe a) where
  identity = Nothing
```

Instance: Maybe

The monoid instance for the `Matrix2` data type is defined by the matrix multiplication operator with the identity matrix as identity element.

```
instance Semiring a => Monoid (Matrix2 a) where
    identity = Matrix2 one zero zero one
```

Instance: Lists

The monoid instance for lists is defined by the list concatenation operation with the empty list as identity element.

```
instance Monoid (List a) where
    identity = Empty
```

Results: Monoid (User-defined)

Instance	1	2	**Time**
Nat (*add*)	✓	✓	4.2 s
Nat (*mul*)	✓	✓	18.4 s
Maybe	✓	✓	1.3 s
Matrix2	✓	✓	2.4 s
List	✓	✓	1.4 s

We note a slight improvement in the proof time for the natural numbers instance using multiplication compared to the same instance for the built-in monoid type class. The reason for this is that we do not need to prove associativity for this operation (associativity is a property of the super class and is thus considered an axiom in this case).

4.4.3 Class: Semiring

A semiring is a set coupled with operators for addition $(+)$ and multiplication $(\cdot)$ which satisfies the laws below. We have also chosen to include the identities for both operations in this definition, which is common practice [4].

Laws for semirings

1	$(a + b) + c = a + (b + c)$	Commutative monoid for $(+)$ and 0
2	$0 + a = a + 0 = a$	
3	$a + b = b + a$	
4	$(a \cdot b) \cdot c = a \cdot (b \cdot c)$	Monoid for $(\cdot)$ and 1
5	$1 \cdot a = a \cdot 1 = a$	
6	$a \cdot (b + c) = (a \cdot b) + (a \cdot c)$	Multiplication distributes over addition
7	$(a + b) \cdot c = (a \cdot c) + (b \cdot c)$	
8	$0 \cdot a = a \cdot 0 = 0$	Annhilation

We have implemented this structure as a simple type class without any super classes.

```
class Semiring a where
    zero :: a
    one  :: a
    add  :: a -> a -> a
    mul  :: a -> a -> a
```

There are a few alternative ways to define this type class. One way is to let the type class definition contain two polymorphic type variables constrained to a commutative monoid and regular monoid type class respectively (for a description of monoids, see section 4.4.2). If the identities were to be left out, the type variables could be constrained to two semigroups instead. However, since we have not implemented support for such class definitions yet (see chapter 6), we've decided to stick with the simple version presented above.

Instance: Natural numbers

Natural numbers form a semiring under addition and multiplication.

```
instance Semiring Nat where
    zero = Zero
    one  = Succ Zero
    add  = natAdd
    mul  = natMul

-- Natural numbers addition
natAdd :: Nat -> Nat -> Nat
natAdd Zero a     = a
natAdd (Succ a) b = Succ (natAdd a b)

-- Natural numbers multiplication
natMul :: Nat -> Nat -> Nat
natMul Zero m     = Zero
natMul (Succ n) m = natAdd m (natMul n m)
```

Instance: Booleans

Booleans form a semiring under logical conjunction and disjunction. In fact, there are two possible instances: one using conjunction as semiring addition with identity element `True` and one using disjunction as semiring addition with identity element `False`.

```
instance Semiring Bool where
    add  = (||)
    mul  = (&&)
    zero = False
    one  = True
```

```
instance Semiring Bool where
    add  = (&&)
    mul  = (||)
    zero = True
    one  = False
```

Instance: 2x2 Matrices

Square matrices form a semiring as long as the contained elements are also semirings. This is represented by a constraint on the element type variable in the instance declaration. Note that the definition of the matrix multiplication operation is inlined here (as opposed to using `matrix2Mul` shown previously). This was done to avoid an issue in the development version of HipSpec causing the system to fail the proofs of a couple of laws on some systems.

```
instance Semiring a => Semiring (Matrix2 a) where
    zero = Matrix2 zero zero zero zero
    one  = Matrix2 one zero zero one
    add (Matrix2 a11 a12 a21 a22) (Matrix2 b11 b12 b21 b22) =
        Matrix2 (add a11 b11) (add a12 b12)
                (add a21 b21) (add a22 b22)
    mul (Matrix2 a11 a12 a21 a22) (Matrix2 b11 b12 b21 b22) =
        Matrix2 (add (mul a11 b11) (mul a12 b21))
                (add (mul a11 b12) (mul a12 b22))
                (add (mul a21 b11) (mul a22 b21))
                (add (mul a21 b12) (mul a22 b22))
```

Results: Semiring

Instance	1	2	3	4	5	6	7	8	**Time**
Nat	✓	✓	✓	✓	✓	✓	✓	✓	28.2 s
Bool (1)	✓	✓	✓	✓	✓	✓	✓	✓	2.9 s
Bool (2)	✓	✓	✓	✓	✓	✓	✓	✓	2.9 s
Matrix2	✓	✓	✓	✓	✓	⊗	⊗	✓	29.3 s

HipSpec manages to prove most properties for this type class but runs into trouble when attempting to prove the distributive properties of matrix multiplication. We are unsure why this is the case. Most likely the prover is missing a lemma required for the proof and this lemma has not been found during theory exploration due to the complexity of the laws or function definitions. Increasing the timeout and number of variables for induction did not make any difference in this case.

We note a rather big difference in time between the boolean instances and the other instances. The main reason for this is that the external theorem prover (Z3) can prove the conjectures very quickly since they are represented using its native boolean operations. HipSpec is also able to simplify boolean expressions quite heavily when compiling directly to TIP code. This is actually an issue when viewing the generated TIP code, since it is hard to see which lemma corresponds to which user-defined property. For a more in-depth discussion of this, see chapter 6.

During the proof process, HipSpec discovers that the commutativity property holds for natural numbers and the two boolean instances. This means that they also form commutative semirings (see next section).

4.4.4 Class: CommutativeSemiring

A commutative semiring is a semiring in which multiplication is a *commutative* monoid. This relationship is represented using a type class without any class methods and with semiring as its super class. This allows us to effectively extend the law definitions for such type classes.

Laws for commutative semirings

1	$a \cdot b = b \cdot a$	Commutativity for $(\cdot)$

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

Results: CommutativeSemiring

Instance	1	**Time**
Nat	✓	21.0 s
Bool (1)	✓	2.8 s
Bool (2)	✓	2.5 s
Matrix2	⊗	11.9 s

As expected from the results of the previous section, the commutativity property holds for the natural numbers and boolean instances. Note that the matrix instance fails as expected; matrix multiplication is not commutative.

Also worth noting is that HipSpec spends the majority of the time proving conjectures generated from the theory exploration, which is why the instance for natural numbers takes longer than a few seconds to complete (the commutativity property is proven very early in the proof process). This is not the case with the instance for matrices however; HipSpec spends the majority of the time trying (and failing) to actually prove commutativity for matrix multiplication.

4.5 Summary and Discussion

In this section we will present a summary of the results in this chapter as well as discuss some interesting points discovered during the evaluation.

Table 4.1: Built-in Type Classes

Eq		
Instance	**Time**	**Success**
Nat	1.3 s	✓
Maybe	1.4 s	✓
Matrix2	3.0 s	✓
List	3.1 s	✓
Tree	3.0 s	✓

Ord		
Instance	**Time**	**Success**
Nat	36.9 s	✓
Maybe	108.8 s	✓
List	150.1 s	✓
Tree	-	⊗

Monoid		
Instance	**Time**	**Success**
Nat *(add)*	4.3 s	✓
Nat *(mul)*	27.3 s	✓
Maybe	2.1 s	✓
Matrix2	2.7 s	✓
List	1.5 s	✓

Functor		
Instance	**Time**	**Success**
List	-	⊗

Table 4.2: User-defined Type Classes

Semigroup		
Instance	**Time**	**Success**
Nat *(add)*	4.0 s	✓
Nat *(mul)*	21.4 s	✓
Maybe	1.3 s	✓
Matrix2	2.2 s	✓
List	1.3 s	✓

Monoid		
Instance	**Time**	**Success**
Nat *(add)*	4.2 s	✓
Nat *(mul)*	18.4 s	✓
Maybe	1.3 s	✓
Matrix2	2.4 s	✓
List	1.4 s	✓

Semiring		
Instance	**Time**	**Success**
Nat	28.2 s	✓
Bool (1)	2.9 s	✓
Bool (2)	2.9 s	✓
Matrix2	29.3 s	⊗

CommutativeSemiring		
Instance	**Time**	**Success**
Nat	21.0 s	✓
Bool (1)	2.8 s	✓
Bool (2)	2.5 s	✓
Matrix2	11.9 s	⊗

HipSpec manages to prove the majority of the evaluation examples for both built-in and user-defined type classes. Some examples, most notably the more complex ones involving trees and matrix operations, were not fully proven. In those cases, however, HipSpec managed to prove a subset of the class laws, and it provided information about which laws failed. This means that the tool is still valuable in those situations, since fewer lemmas need to be proven manually.

One issue we have noticed is with the simplification process of HipSpec. HipSpec processes each property and function definition with various passes which modifies the original property to make it more suitable to external theorem provers. This affects our implementation, since the class laws are not always easily identifiable when reading the output. The problem is even worse when HipSpec have discovered many new lemmas, in which case the class laws are mixed with the new lemmas. There are some solutions for this, such as storing the original property definition for nicer output and partitioning the presented lemmas into user-defined and discovered lemmas. This will be discussed more in chapter 6 - Further Work and Limitations.

Worth noting is that the version of HipSpec used during evaluation is currently under active development and contains some limitations (see chapter 6) which causes issues when communicating with QuickSpec and the external theorem provers. However, the extension generates valid TIP code in those cases and as such more type classes and type class laws will be supported as HipSpec continues to be developed and improved.

5. Related Work

As mentioned earlier there have been several other approaches to reason about type classes and class laws. In this chapter, we will compare the extended version of HipSpec to some of these approaches in the context of proving type class laws. We will see that HipSpec is a very promising tool for this purpose.

5.1 QuickCheck

QuickCheck [5] is a module for Haskell which is used to find counterexamples to user-defined properties. It is perhaps the most common way of ensuring correctness of Haskell programs, along with HUnit [13]. QuickCheck works by generating many possible input values and testing them against properties, making sure they hold for all the input values. Since QuickCheck only relies on testing and not complete proof it is not fully comparable to our work. However, as described in section 2.2 HipSpec uses QuickCheck (via QuickSpec) to generate many conjectures for use in concrete proofs.

QuickCheck is quite good at finding small counterexamples quickly but it does not allow users to specify class laws. A framework for testing type class laws in QuickCheck was presented in the paper "Testing Type Class Laws in Haskell" by Jeuring, Jansson and Amaral [14]. The frameworks support partially defined values, which HipSpec does not. It does however require some manual work for each class law and for each instance data type and as such is not fully automatic. Also, as mentioned before it does not actually prove the class laws unlike HipSpec.

5.2 Isabelle

Isabelle [26] is an interactive theorem prover and as such it will prove complete correctness of properties, instead of just probable correctness via testing. It also has native support for type classes (which makes it a useful comparison to our work), as well as a suite of known lemmas from the field of mathematics. Isabelle uses its own input language and while it is very similar to a functional programming language it has no native support for Haskell code. However, it does support generation of Haskell code from its theories.

While Isabelle is a very powerful theorem prover, using it to prove type class laws requires the user to rewrite the Haskell class and instance definition in another language. This extra step is not only time-consuming, but users also run the risk of introducing (or correcting) errors during the translation process. Isabelle is an interactive proof assistant (in contrast to HipSpec which is an automatic theorem prover) which means that it does not use any form of theory exploration. As such the user is expected to manually supply Isabelle with lemmas which it may be unable to prove. However, it is possible to use external systems for this. One such system, called Hipster, is presented in the paper "Hipster: Integrating Theory Exploration in a Proof Assistant" by Johansson et al. [15]. Hipster actually uses HipSpec for its theory exploration capabilities but proofs are performed inside Isabelle.

Worth noting is that there exists experimental support for converting TIP code into Isabelle theories. This means that we can use HipSpec to generate all TIP properties from Haskell, and prove it using Isabelle instead of Z3. However, it would not use Isabelle type classes when proving type class laws, since the laws have already been specialized in the TIP code.

5.3 Zeno

Zeno [21] is an automated theorem prover which is quite similar to HipSpec. It uses an internal language called HC, which is a simpler version of GHC Core, together with its own internal prover to reason about programs. This is in contrast to HipSpec which have chosen to use a language designed to express first-order logic to allow for easier communication with external theorem provers. While the lack of support for different theorem provers does make Zeno less flexible overall, it is able to output proofs for verification in Isabelle.

One disadvantage is that Zeno has no support for theory exploration but instead uses a system called *lemma calculation* which is less powerful. This means that Zeno might have trouble finding some auxiliary lemmas needed during proofs. Also, Zeno does suffer from the same issues as HipSpec in that there is no support for partial or non-terminating functions.

An earlier comparison of Zeno and HipSpec shows that they performed similarly [6]. Unfortunately Zeno is currently no longer maintained and we have been unable to install and test it for a full comparison of the two systems.

5.4 HALO

HALO [24] is another system similar to HipSpec. It translates Haskell code to first-order logic via an intermediary language called λ_{HALO} to be proven with external provers. HALO does have some interesting qualities in the translation process in that it can handle partial and infinite vaules, which HipSpec cannot. Another nice difference is that the system ensures that if a property holds in the translated version it also holds in the original Haskell program as well. This is seldom actually proven in other similar system; the translation is just assumed to be correct.

Like Zeno, however, HALO is no longer supported or maintained, and we have been unable to test it properly for a full comparison with HipSpec.

5.5 HERMIT

HERMIT [10] is a system for mechanizing equational reasoning about Haskell programs. It can be used to simplify the process of transforming functions and proving equivalence between naïve, easily proven functions and fast, complex versions. Properties are written as GHC rewrite rules in the Haskell source code and proven either interactively in the HERMIT shell or by writing a script which automates the process. It is the first system to have support for the full Haskell language, including type classes and language extensions [9].

HERMIT uses a vastly different approach to proving properties about Haskell programs compared to HipSpec. While HipSpec is fully automatic and utilizes external theorem provers, HERMIT mechanizes and simplifies semi-formal proof processes. This comes with both advantages and disadvantages. The biggest advantage is arguably the support for the full Haskell language and the ability to prove more complex properties than HipSpec (since the proof process is manual). This makes it a very valuable and flexible tool which can be used in many situations. Another advantage is the scripting functionality, which allows proofs to be rerun very quickly since it's basically just rewriting the properties until both are equivalent. However, since the proofs needs to be manually written, HERMIT shares some of the drawbacks with regular, pen and paper proofs. It does require specialized expertise and knowledge about proving properties, and they can also be time-consuming to write. In the context of proving type class laws, proofs for each law will need to be written separately for each instance. HipSpec is comparatively easy to use since it can do all this automatically. Also, HERMIT does not yet do any theory exploration which means that the users are required to input auxiliary lemmas as needed. As such it lacks one of the nicest features of HipSpec, *i.e.* the ability to discover new properties, which is useful even outside the context of proving type class laws.

HERMIT is a great tool for proving properties of Haskell programs, but it does require a great deal of extra work compared to HipSpec. We feel that the two systems fill slightly different purposes, and both are very valuable tools in the Haskell developer's toolbox. In fact, one can well use both systems in conjunction when proving properties by using HipSpec for proving and discovering as many conjectures as possible and then use them in HERMIT for the rest of the proof process.

6. Further Work and Limitations

As can be seen in chapter 4, HipSpec manages to prove laws about many instances of type classes but not all of them. The version of HipSpec used in this thesis is a version that is still in development. Therefore there still exists some issues in the underlying HipSpec system that fall outside the scope of this particular project but affects the performance of our extension. We expect that most of these issues will be fixed in the future and that our failing examples will work without any further modifications to our extension. As such it is our hope that the development of HipSpec continues, and that it becomes a tool used by every Haskell programmer.

In this chapter, we will discuss a few limitations of this extension; some of which were expected from the beginning and some of which we discovered during the development and evaluation stages. We will also explore some potential improvements for HipSpec and the TIP language as a whole.

6.1 HipSpec Improvements

While HipSpec is competing with (and sometimes outperforming [6]) many other theorem provers, there is still room for improvement. For example, as mentioned in section 2.2, HipSpec is not able to reason about non-terminating functions. HipSpec will assume that all functions are terminating and as such the proofs may not hold for those cases. It would be useful to have HipSpec recognize (at least some) non-terminating cases and alert the user beforehand (like Agda's termination checker [1]).

Another useful enhancement would be for HipSpec to be able to give a more detailed output. Currently HipSpec only outputs information about *which* properties it was unable to prove, not *why* it was unable to do so. While some theorem provers used (*e.g.* Waldmeister) is able to do this there is no native support in HipSpec. Maybe it would be possible to use QuickCheck to find potential counterexamples to the failing (user-defined) conjectures in order to assist with the debugging process. This would also provide the advantage of HipSpec failing proofs earlier if this was done before sending the conjectures to the external theorem provers. Another helpful feature related to the proof output is to let HipSpec suggest conjectures which the user can prove manually. When a conjecture has failed to be proved, HipSpec could use theory exploration to find other conjectures which it is unable to prove but if true would allow HipSpec to prove the original conjecture. The user could then prove these discovered conjectures manually through other means, whether that is pen and paper proofs or with other theorem provers.

We also discovered during the evaluation process that it was hard to get a good overview of the actual proof results. It could be a good idea to separate the proven user-defined lemmas and lemmas discovered through theory exploration when presenting the results. A related issue to this is that some information about the original conjectures is lost during HipSpec's simplification process. This makes it more difficult to discern which lemmas correspond to which user-defined conjectures, as HipSpec will only output the simplified versions. To solve this, metadata containing information such as the original names or definitions of the Haskell properties could be stored and used during the proof output.

Another issue is that the output from the external prover is interleaved with the output from HipSpec, thus hindering readability. This could be a problem when using different provers to test a code base, since comparisons between the outputs becomes more difficult. Perhaps HipSpec could parse the output from the different provers and output it in a standardized format.

Finally, some improvements could be made to the theory exploration system of QuickSpec. Currently, only equalities (*i.e.* no implications) are generated and tested. While it would not be trivial to do so, generating conjectures with implications would be beneficial in many cases. In fact, we encountered one such case when defining class laws for the `Ord` instance in chapter 4, which forced us to include additional laws for methods not included in the minimal definition.

In the hypothesis of this project we mentioned that we wished to provide a clear output when the program was incorrectly implemented. While there are some issues with HipSpec's output it is still able to give sufficient information of what it could not prove.

6.2 Implementation Limitations

We have chosen to restrict the extension to only support standard type classes without any language extensions. This means that type classes cannot have multiple parameters (*e.g.* `class Monad m => VarMonad m v`), since it requires the `MultiParamTypeClasses` pragma. Another extension not supported is the `FunctionalDependencies` pragma. Both of the above extensions would add another layer of complexity when specializing functions and generating dummy types. However, we believe it is possible to add support for these and we might see it in the future.

During the project we deemed specialization to have several advantages over some of the other approaches we considered in the early stages of this project (such as implementing support for dictionary passing in the TIP language). However, it also comes with some disadvantages. One such disadvantage is the program size, which could grow quite large if there are many different instances of each type class. This is discussed in the paper "Dictionary-free overloading by partial evaluation" [16], where they find that you can actually *decrease* program size by specializing functions. This is because it is possible for the compiler to inline and optimize differently when it knows the exact implementation called for each function call. HipSpec does some pruning and inlining as well which may or may not achieve similar results, but we have not examined to what extent this would affect code size as it has not been an issue during the evaluation. All unused functions will be pruned away from the generated TIP code, so the size is often not too big, but complex modules could still potentially grow quite large. It would be interesting to do some research on this and see how much of an issue this would be for large projects.

Another size issue (which cannot be solved by pruning and inlining) is that the number of specialized functions generated grows exponentially with the number of constraints in the type signature of a function. For example, consider the type signature of the function presented in code 6.1. During the monomorphization process, we need to consider every possible combination of types the constrained type variables could assume. With just two instances of the `Eq` class we will need four specialized versions of this function (as seen in code 6.2). Each extra constraint would double the number of specialized versions (or more depending on the number of class instances). In practice, however, this is not a huge issue since constraints are rarely applied to more than one type variable at a time, especially when processing type class laws since they almost always work on a single type.

Code 6.1: The type for a function with multiple constraints

```
function :: (Eq a, Eq b) => a -> b -> Bool
```

Code 6.2: The specialized versions of code 6.1

```
  instance Eq Data1 where
    ...

  instance Eq Data2 where
    ...

  function1 :: Data1 -> Data1 -> Bool
  function2 :: Data1 -> Data2 -> Bool
  function3 :: Data2 -> Data1 -> Bool
  function4 :: Data2 -> Data2 -> Bool
```

6.3 TIP Language Improvements

We have found that the TIP language is very suitable for expressing different kinds of first-order logic formulas. However, our implementation of the type class support in HipSpec makes heavy use of uninterpreted sorts. As mentioned in chapter 3, these sorts are created when making assertions on instance types with constrained type variables. This causes a problem similiar to the size issues described in the previous section, as this can generate a lot of code when a class has one or more super classes. Consider the following instances of the `Eq` and `Ord` type classes:

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

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

When generating TIP code for these two instances, each assert on the `Ord` sort also needs to be created for the `Eq` sort. This is because all laws related to the `Eq` class should also hold for the `Ord` class. For example, the TIP code generated for the reflexivity law of the `==` operator is as follows:

```
(declare-sort Eq 0)
(declare-sort Ord 0)
(declare-const Eq.== (=> Eq (=> Eq Bool)))
(declare-const Ord.== (=> Ord (=> Ord Bool)))
...
(assert (forall ((a Eq)) (@ (@ Eq.== a) a)))
(assert (forall ((a Ord)) (@ (@ Ord.== a) a)))
...
```

Also note that we need two uninterpreted dummy functions, `Eq.==` and `Ord.==` for the two different sorts. Contrary to the previous problems, however, this could be fixed if inheritance for sorts were introduced into the TIP language. All asserts on a sort could hold for all children of that sort, and all functions receiving a parent sort as an argument can be used with the children sorts as well. For example:

```
(declare-sort Eq 0)
(declare-sort (Eq -> Ord) 0)
(declare-const Eq.== (=> Eq (=> Eq Bool)))
...
(assert (forall ((a Eq)) (@ (@ Eq.== a) a)))
...
```

However, this would increase the complexity of the TIP language, and a larger code size might be preferable to that if the goal is for external theorem provers to support TIP natively.

Another suggestion for the TIP language is to implement a rudimentary module system. This would allow for code reuse which would be useful in the context of proving type class laws since we could include precompiled laws for the dummy types of built-in type classes. We know that this is something being currently worked on, and it is possible we will see it included in HipSpec soon.

7. Summary and Conclusion

As we've noted in the previous chapters, verifying program and system correctness is a topic which directly affects developers of both large systems (*e.g.* GHC) and small applications alike. Most Haskell developers use unit testing (via HUnit) or QuickCheck tests which provide a low-barrier entry to the world of system verification. While these tools are generally easy to use, they only test with a finite set of input data as opposed to proving properties for all possible inputs. When it comes to proving type class laws, existing systems with such capabilities often require either manual input during the proof process or manual translation into an intermediate language.

The main motivation behind this thesis was to automate the process of proving properties related to type class laws in such a way that the everyday Haskell developer can incorporate it into their workflow. Our hypothesis was that HipSpec could be extended to support this and that this extension could be used to effectively prove type class laws. Since the intermediate language used (TIP) does not have any notion of type classes, we decided to remove the partial polymorphism found among the class constraints by specializing each property for each instantiating data type defined in the file.

We also decided to include some properties for built-in type classes as a convenience for the developers so that they would not need to be redefined in each project. While there are only a few built-in type classes with laws actually stated in the documentation, most of the built-in classes have certain properties that one would expect to hold. For instance you would probably expect the `==` operator to be a complete equivalence relation (something we later found out is not entirely the case for some built-in types). These built-in laws can be automatically proven for custom instances of the classes with HipSpec to ensure to anyone using these (including the developers themselves) that they behave as expected. For now, the usage of these laws are not very convenient since they require a special flag pointing to the directory they reside in. This could potentially be fixed in the future, perhaps using Template Haskell or similar approaches to generate the laws at runtime.

During the evaluation, we found that this specialization process worked very well and we managed to prove many of the instances for both built-in and user-defined type classes. We did encounter some issues, most notably a few laws which could not be proven as well as one example in which we encountered memory issues. In most of those cases, however, HipSpec still managed to prove a subset of the class properties. The remaining laws could potentially be proven using other methods, so HipSpec does still provide value to developers in those cases. Also, some of the more interesting properties involving higher order functions (*e.g.* laws for `Functor` and `Monad`) could not be proven since the development version of HipSpec we used does not currently support such definitions. This is expected to be fixed in future versions, which will make the usage of HipSpec to prove type class laws even more powerful. We also discovered some drawbacks of our implementation, most notably the size of the generated TIP code which can grow quite large if many classes is defined in the file and/or multiple class constraints are present in the function type signatures. We noted that some issues could be solved by extending TIP with additional functionalty, although with increased language complexity.

All in all, we are satisfied with our results and we feel this approach to verifying Haskell programs shows great promise. The current version of HipSpec is still in development and is therefore missing some features which of course does affect the effectiveness of our extension. However, HipSpec is actively being worked on by several parties and we have no doubt that any issues will be fixed in the (hopefully near) future. It is our hope that HipSpec will develop into a powerful and widely used system for ensuring correctness of Haskell programs.

Appendices

A. Laws for Built-in Type Classes

Eq Equality between data structures.

Reflexivity[1]:	$a = a$
Symmetry:	$a = b \iff b = a$
Transitivity:	$a = b \wedge b = c \implies a = c$

Ord Data structures which can be ordered.

$(\leq)$ Transitivity:	$a \leq b \wedge b \leq c \implies a \leq c$
$(\leq)$ Antisymmetry:	$a \leq b \wedge b \leq a \implies a = b$
$(\leq)$ Totality:	$a \leq b \vee b \leq a$
$(\leq)$ Definition:	$a \leq b \implies a < b \vee a = b$
$(\geq)$ Transitivity:	$a \geq b \wedge b \geq c \implies a \geq c$
$(\geq)$ Antisymmetry:	$a \geq b \wedge b \geq a \implies a = b$
$(\geq)$ Totality:	$a \geq b \vee b \geq a$
$(\geq)$ Definition:	$a \geq b \implies a > b \vee a = b$
$(<)$ Transitivity:	$a < b \wedge b < c \implies a < c$
$(<)$ Antitotality:	$\neg(a < b \wedge b < a)$
$(<)$ Definition:	$a < b \implies a \leq b \wedge a \neq b$
$(>)$ Transitivity:	$a > b \wedge b > c \implies a > c$
$(>)$ Antitotality:	$\neg(a > b \wedge b > a)$
$(>)$ Definition:	$a > b \implies a \geq b \wedge a \neq b$

Num Representing numerical data structures.

Sign: $|x| \cdot sgn(x) = x$

Monoid Representing structures with single associative binary operation and an identity element.

Left identity:	`mappend mempty x = x`
Right identity:	`mappend x mempty = x`
Associativity:	`mappend x (mappend y z) = mappend (mappend x y) z`

Functor Representing structures which can be mapped over.

Identity:	`fmap id = id`
Distribution:	`fmap (p . q) = (fmap p) . (fmap q)`

Monad Representing computations as sequences of steps.

Left Identity:	`return a >>= f = f a`
Right Identity:	`m >>= return = m`
Associativity:	`(m >>= f) >>= g = m >>= (\x -> f x >>= g)`

[1] Note that reflexivity does not necessarily have to hold (for instance an invalid Float `NaN` $\neq$ `NaN`), but it is often expected to hold so we have chosen to include it. The proof output will still tell whether the other laws hold if this fails.

Bibliography

[1] *Agda Reference Manual.* 2016. URL: http://wiki.portal.chalmers.se/agda/pmwiki.php?n=ReferenceManual.TerminationChecker.

[2] Clark Barrett, Aaron Stump, and Cesare Tinelli. "The SMT-LIB Standard - Version 2.0". In: *International Workshop on Satisfiability Modulo Theories.* 2010.

[3] Stefan Bechtold, Sam Brannen, Johannes Link, Matthias Merdes, and Marc Philipp. *JUnit 5 User Guide.* 2016. URL: http://junit.org/junit5/.

[4] Jean Berstel and Dominique Perrin. "Theory of Codes". In: 1985. Chap. Preliminaries, p. 26.

[5] Koen Claessen and John Hughes. "QuickCheck: a lightweight tool for random testing of Haskell programs". In: *Proceedings of the fifth ACM SIGPLAN International Conference on Functional Programming.* 2000.

[6] Koen Claessen, Moa Johansson, Dan Rosén, and Nicholas Smallbone. "Automating Inducive Proofs using Theory Exploration". In: *Proceedings of the Conference on Automated Deduction.* 2013.

[7] Koen Claessen, Moa Johansson, Dan Rosén, and Nicholas Smallbone. "TIP: Tons of Inductive Problems". In: *Proceedings of the Conference on Intelligent Computer Mathematics (CICM).* 2015.

[8] Koen Claessen, Nicholas Smallbone, and John Hughes. "QuickSpec: Guessing formal specifications using testing". In: *Proceedings of TAP.* 2010.

[9] Andrew Farmer. "HERMIT: Mechanized Reasoning during Compilation in the Glasgow Haskell Compiler". PhD thesis. University of Kansas, 2015.

[10] Andrew Farmer, Neil Sculthorpe, and Andy Gill. "Reasoning with the HERMIT: Tool support for equational reasoning on GHC Core programs". In: *Proceedings of the 8th ACM SIGPLAN Symposium on Haskell.* 2015, pp. 23–34.

[11] Jean-Yves Girard. "Une Extension de l'Interpretation de Gödel à l'Analyse, et son Application à l'Élimination des Coupures dans l'Analyse et la Théorie des Types". In: *Proceedings of the Second Scandinavian Logic Symposium, Amsterdam.* 1971.

[12] *Haskell 2010 Language Report.* Tech. rep. 2010. URL: https://www.haskell.org/definition/haskell2010.pdf.

[13] Dean Herington. *HUnit 1.0 User's Guide.* 2002. URL: https://wiki.haskell.org/HUnit_1.0_User's_Guide.

[14] Johan Jeuring, Patrik Jansson, and Cláudio Amaral. "Testing Type Class Laws in Haskell". In: *Haskell Symposium.* 2012, pp. 49–60.

[15] Moa Johansson, Dan Rosén, Nicholas Smallbone, and Koen Claessen. "Hipster: Integrating Theory Exploration in a Proof Assistant". In: *Proceedings of the Conference on Intelligent Computer Mathematics (CICM).* 2014.

[16] Mark P. Jones. "Dictionary-free overloading by partial evaluation". In: *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation.* 1994.

[17] Simon Marlow and Simon Peyton Jones. "The Architecture of Open Source Applications". In: vol. 2. 2012. Chap. 5. The Glasgow Haskell Compiler.

[18] *Report on the Programming Language Haskell, Version 1.0.* Tech. rep. 1990. URL: `https://www.haskell.org/definition/haskell-report-1.0.ps.gz`.

[19] John C. Reynolds. “Towards a theory of type structure”. In: *Colloquium sur la programmation.* 1974.

[20] Dan Rosén. “Proving equational Haskell properties using automated theorem provers”. MSc thesis. University of Gothenburg, 2012.

[21] William Sonnex, Sophia Drossopoulou, and Susan Eisenbach. “Zeno: An automated prover for properties of recursive data structures”. In: *Tools and Algorithms for the Construction an Analysis of Systems.* 2012.

[22] Martin Sulzmann, Manuel M. T. Chakravarty, Simon Peyton Jones, and Kevin Donnelly. “System F with Type Equality Coercions”. In: *ACM SIGPLAN Workshop on Types in Lanugage Design and Implementation.* 2007.

[23] Andrew Tolmach and Tim Chevalier. *An External Representation for the GHC Core Language (For GHC 6.10).* 2010. URL: `https://www.haskell.org/ghc/docs/6.10.3/html/ext-core/core.pdf`.

[24] Dimitrios Vytiniotis, Simon Peyton Jones, Koen Claessen, and Dan Rosén. “HALO: Haskell to logic through denotational semantics”. In: *Proceedings of POPL’13. ACM.* 2013.

[25] Philip Wadler and Stephen Blott. “How to make ad-hoc polymorphism less ad hoc”. In: *ACM Symposium Principles of Programming Languages.* 1988.

[26] Makarius Wenzel. *The Isabelle/Isar Reference Manual.* 2016. URL: `https://isabelle.in.tum.de/dist/Isabelle2016/doc/isar-ref.pdf`.