Verification of Haskell Type Classes

By
Feng Wang, B. Eng.

A Thesis
Submitted to the School of Graduate Studies
in partial fulfilment of the requirements for the degree of

M.A.Sc. in Software Engineering

McMaster University

MASTER OF APPLIED SCIENCE (2007)
(Computing and Software)

McMaster University
Hamilton, Ontario

TITLE: Verification of Haskell Type Classes

AUTHOR: Feng Wang, B. Eng. (Northeastern University, China)

SUPERVISOR: Dr. Wolfram Kahl

NUMBER OF PAGES: vi, 55

Abstract

The Haskell programming language uses type classes to deal with overloading. Functions are overloaded by defining some types to be instances of a class. A meaningful instance should satisfy the invariants of the class.

In this thesis we present one method to validate the type instances of classes informally, and another one to verify them in a formal way.

The first method uses QuickCheck, which is an automatic testing tool for Haskell programs. We introduce how to specify the properties of type classes in QuickCheck by some examples, and I also present testing for Haskell standard types and classes.

The second method I adopted uses the theorem prover Isabelle/HOL. To facilitate the usage of Isabelle/HOL for Haskell programmers, I define a set of translation rules from Haskell programs to Isabelle/HOL, and design a simple automatic translating tool based on those rules. Logical differences between Haskell and Isabelle/HOL need to be considered in the translation. For example Isabelle/HOL is not suitable to describe the semantics of lazy evaluation and of Haskell functions that are nonterminating. I also prove some type instances to illustrate how the properties are verified in Isabelle/HOL.

Acknowledgements

I got a lot of help during the writing of this thesis. Here I like to express my thanks and appreciations to those who give me support.

First I give my thanks to Dr. Wolfram Kahl who is my supervisor. It is him to lead me into an amazing world of Haskell programming. During two years study under his supervision, I learn a lot from him, not only in my research area, but also in a wide spectrum of computer science. They are so important to me for a successful career in the future.

Second I like to give my thanks to Paul Hachmann and Scott West. They are my colleagues who are also supervised by Dr. Kahl. They provide great help on the revision of this thesis. I got the final version from draft in a short time following their detailed suggestions.

Third I want to express my thanks to my family, Yankun Li who is my wife gives me great support from home, and my son Roger Wang who bring family the happiness.

I also want to thank rest of others not mentioned here, I want they know their help and regard are so important to me.

Contents

Chapter 1

Introduction

The major task of this thesis is to present some useful methods for verifying type classes of Haskell [HJW+92]. This chapter introduces the background of research in this area and some technologies involved.

1.1 Background and Motivation

Software reliability is very often an objective of software designers and developers. Currently, improvement on this issue is achieved through diverse ways: careful system design, high quality source code, rigorous testings, formal methods and some other techniques to detect and reduce bugs that leads the system to failure. In this thesis I present two methods: testing in QuickCheck [CH00] to validate and Isabelle/HOL [NPW02] to verify type class laws for instances of a class.

The type class system is a unique feature of the Haskell programming language. It is always used to deal with overloadings. The basic idea is that class declarations group together overloaded functions which have some relation amongst themselves. The instance declaration includes a type into a class by providing definitions for members of the class. For example, the type class Eq provides the function $(\equiv)$ to allow two values of its argument type to be compared in equality.

We declare the type Int to be a member of Eq

```
instance Eq Int where
  (≡) x y = prim_Eq_Int x y
```

type Bool also could be compared

```
instance Eq Bool where
```

```
(≡) x y = prim_Eq_Bool x y
```

Not only can users declare some user-defined types to be member of an existing type class, but they can also define their own classes and include both predefined or user-defined types in them.

A type class is actually a general programming interface to which a type can easily be "adapted" by a third programmer even if the two programmers who wrote the type and type class were working separately and with no knowledge of each other. To be adapted correctly, frequently the interface exposed by a type class is expected to satisfy certain mathematical laws. It is customary to state these laws with documentation but Haskell has no mechanism to check theses laws.

Look at the following datatypes:

```
data List a = Nil | Cons a (List a)
data CoList a = Nil | Cons (CoList a)
data Stream a = Cons a (Stream a)
```

and type classes of Functor, Applicative, Monad, CoMonad and Monoid

Datatype List can be a member of any type class but CoMonad, CoList can be an instance of Monad and CoMonad, but violates the invariants of them, and a Stream instance of type class Monoid makes no sense.

Validations or verifications are necessarily required to build confidence for programmers when they are working on type classes. Motivated by this issue, I present in this thesis some methods to test and verify the type instances.

1.2 Approach Involved

QuickCheck

An automatic testing tool is highly recommended for validation of Haskell programs. It eases the testers from executing the programs again and again. Haskell programs are suitable for automatic testing due to its pure functional programming nature which has no side-effects. QuickCheck is a lightweight tool for testing of Haskell programs.

To be tested, properties should be written in some specification language. QuickCheck uses the Haskell language itself. All properties were written as Haskell functions with return values of Bool type. As a simple example, testing the laws of the list reverse function defined in the Haskell prelude shows how to write properties and test them in QuickCheck.

QuickCheck can apply to higher-order types such as testing laws for function types. After defining the extensional equality (===) by (f === g) x = f x ≡ g x you can test the associativity of function composition. While QuickCheck prints the counterexamples for the failed tests, four combinators makes it capable of monitoring test data. Combinator classify counts the trivial cases; combinator collect reports the data distribution, and so on.

QuickCheck provides a set of testable types in its library. All of these types have a predefined data generator respectively. When a property ranges over those types, the testing is pretty simple, when users need to use some types that are not in the testable set they can write their own data generator through data generator combinators. one even can define function generators that explain the reason why properties on function types can be tested.

QuickCheck is easy to use and is efficient. Most bugs can be exposed in a short time by running test cases in QuickCheck. But there are some shortcomings to prevent it from being fully trustable. Like other testing tools, it cannot ensure testing data covers all conditions. For some properties that are too general, the difficulties are obvious. If the laws of type classes could be verified by formal methods, it will greatly increase the program's quality concerning the usage of type classes, and raise the confidence of Haskell programmers. We choose Isabelle/HOL [NPW02] in this case.

Isabelle/HOL

Isabelle is a theorem proving framework which has built-in support for several logics including several first-order logics, Simple Type theory, and Zermelo-Fraenkel set theory [Pau89, Pau90b]. Isabelle/HOL is the specialization of Isabelle for HOL. We decided to choose Isabelle/HOL because of its functional programming feature which can be used to formalize the specifications that were originally expressed in the Haskell programming language; another reason is that HOL is a typed logic; its type system is close to that of Haskell.

Working with Isabelle/HOL is a procedure of creating theories and finding proofs. A theory is composed by a collection of types, terms and formulae written in HOL syntax.

Types are an important component of an Isabelle theory. In a typed system, every element should be well-typed. The definitions need not be explicitly typed because of Isabelle's type inference mechanism.

Isabelle/HOL [NPW02] provides some built-in types such as nat, Pair, Bool, List, product,set etc.. It also allow the users to define new types. The general format to define a new type is of the form:

datatype $(\alpha_1, \ldots, \alpha_n)t = C_1\ \tau_{11} \ldots \tau_{1k_1} \mid \ldots \mid C_m\ \tau_{m1} \ldots \tau_{mk_m}$

where α_i are distinct type variables, C_i are the type constructors and τ_{ij} are specific types. This definition resembles the data declarations of Haskell. Recursive datatypes are allowed. Besides new datatype definition, Isabelle/HOL [NPW02] offers type synonyms to define a alias for an existing type. For example

types number = nat

types gate = "bool ⇒ bool ⇒ bool"

Terms mainly refers to function application. Isabelle supports some basic structures which are widely used in Haskell such as conditional expressions, let expressions, and case expressions.

Formulae are terms of type bool.

Isabelle/HOL can be used to verify Haskell programs. The example of proving the specifications of Haskell function reverse shows us the possibility and manner of verification. This example can be found in [CH00].

1.3 Related Work

Currently there are two projects in development which can be used to formalise and prove the Haskell programs; they are also implemented in Haskell. One is programatica [Hal03] which is developed at the OGI School of Science & Engineering. It is a program development environment which allows programmers to assert properties of program elements as part of their source code. Programatica integrates several tools to provide a range of validation options from low-cost automatic testing, to machine-assisted proof and formal methods through a generic interface. To be validated or verified, the properties expressed by a logic, named P-logic [HK05, Kie02], need to be translated to other logics. Property assertions are annotated with certificates that provide evidence of validity, and are managed by property management tools which provide users with facilities to browse or report on the status of properties and associated certificates within a program, and to explore different validation strategies.

Another project is the heterogeneous toolset HETS which is developed by the DFK1 Lab Bremen and department of Computer Science, University of Bremen, Germany. The purpose of this toolset is to specify large systems where heterogeneous multi-logic specifications are needed. Different logics which have their strengths in particular fields can be used to specify different aspects of a complex problem. Different approaches being developed in different environments can be related, and in HETS the combination takes little effort. Heterogeneous specification is based on individual (homogeneous) logics and logic translations [MML07]. In HETS, logic and logic translations are called institution and institution comorphisms. Haskell logic's

proof relies on Isabelle by translating Haskell specifications into Isabelle through an institution comorphism between them. HETS provides a translator from Haskell to Isabelle/HOLCF; but it does not support property-specified verification. Instead, *P*-logic of Programatica has been integrated into HETS for specification purposes.

P-logic, as the main logic of the Programatica project, is a modal logic. Its intended domain is interpreted as a family of sets with a particular structure instead of a simple set. Two modalities of the logic, called weak and strong respectively, determine whether a predicate is interpreted by a set of normalized values of its type, or by a set of computations of its type, which may or may not terminate [HK05]. *P*-logic is used to reason about the lazy-evaluation semantics of Haskell. However, while *P*-logic provides the advantage of specifying Haskell programs with precise semantics it also brings some difficulty. *P*-logic is a much more complex logic than Haskell; writing specification in *P*-logic is a big challenge to Haskell programmers. This thesis intends to provide simple methods by which Haskell programmers can specify properties solely using Haskell functions.

For validation strategy, there are some other tools that are designed for testing Haskell programs, such as SmallCheck [CH06a] and SparseCheck [Nay06]. SmallCheck, which will be introduced briefly in a later chapter, is a upgrade to QuickCheck designed by the same team. The purpose of SparseCheck is to test properties which have a sparse input domain. Those properties are usually associated with a condition that is not satisfied by a large portion of the input space. To test such properties, QuickCheck and SmallCheck spend most of their time in generating test data that falsifies the condition. The current version of SparseCheck is not used practically due to limitations of completeness and efficiency.

1.4 Organization

Chapter 2 introduces polymorphism, the Hindley/Milner type system [Hin69, DM82], and Haskell type classes. In this chapter the thesis also describe the laws for type classes, which thesis is working on.

Chapter 3 introduces the testing tool, QuickCheck; thesis discuss how to test properties of type classes in this incomplete way.

Chapter 4 introduces the theorem prover Isabelle/HOL,In section 4.2 thesis will discuss the syntax and semantics difference between Haskell and Isabelle/HOL, thesis will give suggestion on transformation from Haskell to Isabelle/HOL. In section 4.3 the goal is to show how to verify type classes in Isabelle/HOL.

In Chapter 5 discusses the conclusions on the work and contributions. In this

chapter thesis also propose some future work.

Chapter 2

Polymorphic type and type classes in Haskell

This chapter provides a brief introduction to type polymorphism and type classes in the Haskell.

2.1 Type polymorphism

In programming languages, polymorphism is a way that allows names to be reused many times. It refers to a single definition of data or functions that can evaluate to or apply to many types. A polymorphic function definition can be used to replace several type-specific functions [CW85], and a polymorphic operator can apply to expressions of multiple types. Many programming languages implement polymorphism; for example the object-oriented languages such as C++ and Java.

A function is polymorphic if it takes parameters with various types and results in a value which ranges over different types. In addition to functions, polymorphism data types also could be polymorphic if some components of a type comes from multiple other types. It is said to be a polymorphic data type.

Basically there are two kinds of polymorphism: Parametric polymorphism and ad-hoc polymorphism; the later one is also called overloading. Parametric polymorphism ranges over any type while the number of types that ad-hoc polymorphism is constrained to is finite, and the combinations must be specified before use. Parametric polymorphism allows one to implement code without being concerned about the difference among types and the code can be easily applied to any number user-defined new types.

As mentioned above, the concept of polymorphism started to be popular since the object-oriented programming languages caught attention from programming communities. They consider polymorphism a unique feature and a major benefit of object-oriented programming languages. However functional programming languages also take advantages of polymorphism.

2.1.1 Parametric polymorphism

Parametric polymorphism was first implemented in programming languages in 1976 by ML [MTM97], after that more and more programming languages adopted it, such as Miranda [Tur90] and Haskell [HJW+92]. etc..

Parametric polymorphism is most often used in generic programming. Functions are defined generically to handle the objects of any data types. They don't need to consider the differences between types and treat them equally. Here we take the instance of calculation of length of a list: length, it is concerned with the number of elements of list, it does not care about the type of elements in the list. It calculates the length of lists of integer, length of lists of Bool and length of lists of tuples pairing with any other types. Let α be a type variable, a list generated from it has a type $[\alpha]$, we declare the function length with a type signature $[\alpha] \rightarrow Int$; it shows parametric polymorphism parametrized by type variable α. By applying parametric polymorphism, the programs will be more expressive.

Parametric polymorphism is classified as **predicative** and **impredicative**. The difference is in the way how one instantiates the type variables in a parametric polymorphism. In predicative parametric polymorphism, the types substituted for type variables are not polymorphic types themselves. In the impredicative situation, the types substituted could be polymorphic types or any types.

Subtyping polymorphism Some textbooks mention subtyping polymorphism, which is a special case of parametric polymorphism. It allows a function defined on a type T to work well on a type S if type S is a **subtype** of type T. We take the notation $\preceq$ and $\succeq$ to describe the relation between these types. S $\preceq$ T shows S is subtype of T, and T $\succeq$ S tells us T is supertype of S. In object-oriented programming languages this kind of polymorphism is known as inheritance, it is defined on the objects of virtual classes, and applied over objects of inherited classes. Its operation relies on late binding or dynamic binding.

2.1.2 Ad-hoc polymorphism(Overloading)

Not all kinds of functions can be written in generic form to apply to any type. Although sometimes one can find same function name could be used to different types due to the same objective of operation, the implementation details are very different. In some object-oriented languages, one can use $(+)$: $a \times a \rightarrow a$ to concatenate two strings while $+$ is a sum operator for NUM types. In most situations one should not implement some operation on some types. This second form of polymorphism is **ad-hoc polymorphism** or so called **overloading**

Overloading allows one to define a single name to various functions, all of which take different number of arguments, different types of arguments or same arguments but with different orders from others. By passing specific parameters the complier decides to call the right one. It is kind of **user-friendly** mechanism,users just need to know the function name, its arguments and types for use, but producers have to implement multiple functions for each combination of parameters. This also determines that users can not apply the a overloading function to any arbitrary types. But overloading allows a function to perform some things completely different.

Overloading is commonly used in object-oriented programming languages, where are often seen a bunch of functions sharing same name in a class definition. More specific is function overriding which allows to completely change the behavior of existing functions or operators.

Functional programming language Haskell [HJW+92] supports parametric polymorphism and ad-hoc polymorphism by the use of type classes.

2.2 Type classes in Haskell

2.2.1 Hindley/Milner type system

Before we talk about polymorphism in Haskell [HJW+92] language by means of type classes,It is useful to mention Hindley/Milner type system [Hin69, DM82], which we refer to as HM. HM has been adopted as the type language basis for most functional programming languages such as ML [MTM97], Miranda [Tur90], Haskell [HJW+92], etc..

HM is a restricted form of polymorphism. Type abstraction and application are implicit; it frees programmers from the effort of providing explicit type declaration. It's type inference algorithm was to used to infer the type information for each implicitly defined type. Take some examples [Jon97] from:

```
data List a = Nil | Cons a (List a)
```

which defines a new data type with two constructors:

$\mathsf{Nil} :: \forall\ \mathsf{a}.\ \mathsf{List\ a}$

$\mathsf{Cons} :: \forall\ \mathsf{a}.\ \mathsf{a} \rightarrow \mathsf{List\ a} \rightarrow \mathsf{List\ a}$

The function head takes the first element from a list

$\mathsf{head} :: \forall\ \mathsf{a}.\ \mathsf{List\ a} \rightarrow \mathsf{a}$

$\mathsf{head\ (Cons\ x\ xs)} = \mathsf{x}$

While the function length computes the number of elements of a list.

$\mathsf{length} :: \forall\ \mathsf{a}.\ \mathsf{List\ a} \rightarrow \mathsf{Int}$

$\mathsf{length\ Nil} = 0$

$\mathsf{length\ (Cons\ x\ xs)} = 1 + \mathsf{length\ xs}$

What makes the HM type system successful in programming language is the following features:

Static type values involved in the program were statically typed, which means the type of a value was determined at compile time. It guarantees that the program does not go wrong due to type errors at run time. It significantly reduces the number of bugs compared to some dynamically typed programming languages, which have to check the type information at run time and hiding some bugs even for the software's whole life cycle.

Type inference refers to the technology to deduce the type of the value derived from evaluation of an expression automatically, either partially or fully. The complier takes responsibility to infer the most general type without the requirement of as explicit type annotation or type signature on the expression. HM's type inference algorithm is used to determine whether a given program term is well-typed, and to calculate the principal type. Even though some programmers prefer to give explicit type information in their programs, they still can take advantage from type inference by means of type consistency checks that compares the given type signature and the type inferred automatically.

Flexibility of usage: the most important feature of HM is allowing polymorphism. The users define the functions applied on various types with uniform behavior. It eases the programmers from defining functions which would otherwise need to be defined several times regularly.

Although the HM type system provides the above attractive features to language designers the limitations are also obvious. It is easy to find some classical examples [Jon95] that shows the difficulty designers meet. We take examples of function of equality (=) and function of arithmetic addition (+)

If we define equality function over some type constructor C as a monomorphic

function $C \rightarrow C \rightarrow Bool$, the equality of two values of type C could be compared; but it is not as general as we hoped to compare two values of any types. If it is defined as polymorphic type $a \rightarrow a \rightarrow Bool$ equality of function types has to be included as a case.

When we define the addition function over integer type, it works well as a monomorphic function of type $Int \rightarrow Int \rightarrow Int$ to add two integer values. But it is impossible to be used to add two float point values together because it is less general as we expect. If we just pursue the general and define the function as a polymorphic type $a \rightarrow a \rightarrow a$ we will take the risk that add two values of any type when addition just makes sense on numeric types.

From the above examples we find that while monomorphic types set up a dilemma for us by being too less general, polymorphic types push us to another extreme by being too general. Some languages must then find solutions to fix these problems. For example, ML [MTM97] uses a special type variables to range over the types on which equality or addition are defined [NP93]. What causes these limitations is that both equality and addition differ from other polymorphic functions not only because of their restricted domain but also the reality of the dual property of both polymorphism and overloading.

Type classes in Haskell [HJW+92] programming language were introduced to solve those problems by means of a middle step between of monomorphic and polymorphic types. It allows functions to be defined over a range of types without necessarily ranging over all types.

2.2.2 Type classes basics

A type class could be treated as a set of types. It comes up with a class name and one or more operations. If a type needs to be an element of a specific type class, it needs to be declared an instance of that class by implementing the operations provided by it. Class Eq provides operations $(\equiv)$, $(\not\equiv)$; class Ord provides operations $(<), (\leqslant), (\geqslant), (>)$ etc...

```
class Ord a where
  (<), (⩽), (⩾), (>) :: a → a → Bool
  max, min           :: a → a → a
```

the predicate Ord a says that type a is a member of type class Ord, it is also called the context of class that always appears before a type expressions, e.g.

```
sort :: (Ord a) ⇒ [a] → [a]
```

Context Ord a imposes a constraint on the type of function sort which allows it only to be applied on those types included in Ord class, not any arbitrary type. To understand type classes, we should describe how to define a class, how the new types are declared an instance of a class, and how the existing type classes may be extended.

2.2.3 Deriving classes and instance declaration

Haskell [HJW+92] programmers often deal with type classes in their work, especially when a new datatype is defined. If one wants some standard functions to be applied on the new datatypes, the simplest way is integrating a list of classes as a component of the definition of the datatype, For example

```
data Month = Jan | Feb | Mar | Apr | May | Jun
  | Jul | Aug | Sept | Oct | Nov | Dec
    deriving (Eq, Ord, Show, Text)
```

The **deriving** clause in the third line of the datatype definition tells the compiler to generate instances of the given classes for Month. The newly defined type Month will belong to standard classes: Eq, Ord, Show, Text. Now the datatype Month is a member of above four classes; it automatically overloaded the operations provided by them. For example:

```
Predule|>| elem Mar [Jan,Feb,Mar,Apr]'
True'
Predule|>| show [Jan,Feb,Mar,Apr]'
[Jan,Feb,Mar,Apr]'
```

where elem has the type of Eq a ⇒ a → [a] → Bool

It allows the value of data to be comparable when deriving a class of Eq. It allows the value of data could be transfered to characters and printed on screen when deriving a class of Show.

Deriving is a convenient way to allow programmers to include some datatypes in some type classes but sometimes one will find he is in a position deriving does not work at all, such as if the user-defined datatype is not suitable to inherit some classes directly since the derived version of the overloaded function doesn't satisfy the semantics. For example of equality of Set [Jon95]. Define a datatype for set:

```
data Set a = Set [a] deriving (Eq)
```

The resulting for set equality will be

$$\mathsf{Set\ xs} \equiv \mathsf{Set\ ys} = \mathsf{xs} \equiv \mathsf{ys}$$

But if xs and ys take the value of $[1,2]$, $[1,2,2]$ respectively, xs and ys are not identical. At least we can see the element numbers in the two lists are not same. However in the sense of Set Set xs and Set ys are equal, because in the sets there are no duplicate elements. It is necessary for programmers to interpret the meaning of equality for the user-defined types. An alternative way to extend a class is the **instance** clause. To show how to declare an instance of a class we take a look of standard type class Eq

```
class Eq a where
  (≡) :: a → a → Bool
  (≢) :: a → a → Bool
  x ≢ y = !(x ≡ y)
  x ≡ y = !(x ≢ y)
```

above code defines a type class Eq on polymorphic type a, the class provides two operations: equality and inequality of the type $\mathsf{a} \to \mathsf{a} \to \mathsf{Bool}$; the last two lines give the default definition of operations (≡) and (≢) in terms of each other. If one of these functions is defined then the other one will be defined implicitly.

To enable sets being comparable we declare an instance of class Eq on type Set by implementing one of the two operations on type Set with desired semantics. Here we redefine the datatype for Set and make an instance of class Eq

```
newtype Set a = MkSet [a]
instance Eq a ⇒ Eq (Set a) where
  (≡) (MkSet x) (MkSet y) = (subset x y) ∧ (subset y x)
```

This instance describes the equality of sets. Function subset eliminates the duplicate in lists x and y, and returns the truth value if its first argument is the subset of its second argument. The complete definition of subset is found in the technical report by Mark Utting [Utt94].

2.2.4 Defining class

In last section we talk about how to use classes and how to extend an existing class to include some new defined types. In most time this is enough for programmers, but

sometimes programmers will greatly benefit from defining their own type classes and instance some important types.

Before create a class, one has to get the idea what operators they want to provide and these operators should reflect general properties among the datatypes one will make the instance. We define a Tree class [Jon95]and instance some popular tree types [Jon95] on it to show a complete procedure of application of classes.

Trees are widely used data structures in programming languages especially in functional programming languages, Many of algorithms are designed on the datatype of trees, such as some search algorithms based on search trees, parsers based on abstract syntax trees(ASTs). There are some basic calculations on trees, such as depth, size, paths, mirrors. The computations depend on the form of the trees. Different trees implement their own version of the functions. By the recursive definition of tree types, we can find there exist some general computations; each iteration evaluates its subtree first which could be considered as general operations of a class. Following are definitions of various of tree types:

```
data BinTree a = Leaf a
  | BinTree a :^: BinTree a
```

This defines a binary tree, each leaf node contains the data, every inner node takes two subtrees.

```
data LabTree l a = Tip a
  | LFork l (LabTree l a) (LabTree l a)
```

label tree, besides the value of type a in each leaf node, every inner node is indexed by a label of type l.

Binary search tree, with data values of type a in the body of the tree. These values typically be used in conjunction with an ordering on the elements of type a in order to locate a particular item in the tree [Jon95].

```
data STree a = Empty
  | Split a (STree a) (STree a)
```

Rose tree, in which each node is a labelled with a value of type a, and may have an arbitrary number of subtrees.

```
data RoseTree a = Node a [RoseTree a]
```

A simple abstract syntax tree which represents λ expression in a interpreter

```
type Name = String
data Term = Var Name
  | Ap Term Term
  | Lam Name Term
```

After the definitions of tree types. we design a type class with a general operator subtrees which computes the all subtrees of a given tree.

```
class Tree a where
  subtrees :: a → [a]
```

Function subtrees returns the list of subtrees of a given tree t of type a. We declare an instance for every tree types defined above.

```
instance Tree (BiTree a) where
  subtrees (Leaf n) = []
  subtrees (l : ^ : r) = [l, r]


instance Tree (LabTree l a) where
  subtrees (Tip x) = []
  subtrees (LFork x l r) = [l, r]


instance Tree (STree a) where
  subtrees Empty = []
  subtrees (Split x l r) = [l, r]


instance Tree (RoseTree a) where
  subtrees (Node a gts) = gts


instance Tree Term where
  subtrees (Var _) = []
  subtrees (Ap f x) = [f, x]
  subtrees (Lam v b) = [b]
```

Based on the definitions of subtrees, a library of functions could be defined, such as computation of depth and size. The function subtrees is also an important part of implementations of some complex algorithms such as depth-first algorithm and breath-first algorithm of tree data structures [Jon95].

2.2.5 Functor and Monad

Functor and Monad are two important classes in Haskell prelude.

Functor's mathematic foundation is category theory; a functor is the mapping between categories. The mathematical definition is

Let C and D be categories. A functor F from C to D is a mapping that

associates to each object $X \in C$ an object $F(X) \in D$,

associates to each morphism $f : X \to Y \in C$ a morphism $F(f) : F(X) \to F(Y) \in D$

and two properties should hold:

$F(id_X) = id_F(X)$ for every object $X \in C$

$F(g \circ f) = F(g) \circ F(f)$ for all morphisms $f : X \to Y$ and $g : Y \to Z$.

In Haskell, a functor could be defined as:

```
map :: (a → b) → ([a] → [b])
map f [] = []
map f (x : xs) = f x : map f xs
```

This is a functor composed by the type constructor [ ] and the map function. An obvious shortcoming of this functor is that the domain of the functor is restricted to the list type. An overloaded version of map will be general to fit for more types It is declared by class Functor with an operator fmap:

```
class Functor f where
fmap :: (a → b) → (f a → f b)
```

Some of the types defined in the prelude are treated as functors, such as Id, List, Maybe, Either ...

```
instance Functor Id where
  fmap f (Id x) = Id (f x)
instance Functor [a] where
  fmap f [] = []
  fmap f (x : xs) = f x : fmap f xs
instance Functor Maybe where
  fmap f (Just x) = Just (f x)
  fmap Nothing = Nothing
instance Functor (Either a) where
```

```
fmap _ (Left x) = Left x
fmap f (Right y) = Right (f y)
```

Monads are special cases of functors. A monad in functional programming is a way to build parts of purely functional programs so that the functions involved are applied in a sequence, as an imperative language does. You can perform a sequence of operations while still taking a functional approach. The monad class has the following definition with basic operators of return and "bind".

```
class Functor m ⇒ Monad m where
  return :: a → m a
  ≫= :: m a → (a → m b) → m b
```

Note: this is not valid Haskell98 definition.

Some datatypes mentioned before are monads. For these, a monad instance can be declared:

```
instance Monad Id where
  return = id
  id x ≫= f = f x
instance Monad [a] where
  return x      = [x]
  [] ≫= f       = f
  (x : xs) ≫= f = f x ++ (xs ≫= f)
instance Monad Maybe where
  return x       = Just x
  (Just x) ≫= f  = f x
  Nothing ≫= f   = Nothing
```

2.2.6 Laws of type classes

In last section we discussed the Functor and Monad classes. As mathematical concepts, there are some basic laws that they should obey. The laws determine what these mathematical definitions are, state what properties they have, and provide basic rules in proof and computation.

The laws for functors reflect the properties of its mathematical foundation category theory:

On any category C one can define the identity functor Id which maps every object and morphism to itself

```
fmap id = id
fmap f ∘ fmap h = fmap (f ∘ h)
```

What make the Monad a Monad are the laws of [Wad92]

```
1 return a >>= f ≡ f a
2 f >>= return ≡ f
3 f >>= (λx → g x >>= h) ≡ (f >>= g) >>= h
```

The first law says that return a >>= f ≡ f a. As we think about Monads as computation, this law states that if we construct a computation which just returns the value of a without considering its means, and then binds its result to another computation f, the whole job just can be replaced by a computation of f on value a.

The second law says f >>= return ≡ f, when computation f binds the result to a return, it does the same thing by returing f along with it.

The third one is the associativity law for monads.

Like the Functor class and the Monad classes many other classes are associated with implicitly understood laws to their mathematical definitions. Eq takes laws of equivalence, Ord class has the laws of total order with respect to (≡). In the coming chapters we will discuss how to test and prove a given instance of a class satisfy its laws.

Chapter 3

Validation of type instances in QuickCheck

QuickCheck is a tool for testing Haskell programs. In this chapter thesis will give a brief introduction of QuickCheck, showing how to specify properties, how to define data generators, and then discuss how to test type instances. thesis also specify the properties of some standard classes.

3.1 Overview of QuickCheck

For long time, testing has remained an important approach to ensure software quality in software development. It aims to find errors in software by means of running the software. Testing comes in many flavors: unit testing [CH02], property testing, regression testing, contract checking, black-box/white-box testing [JJ06].

A test tool should deliver the testing result, that is a message of test success or failure and some counterexamples, in a short time, and it should be repeatable when the program or specification is modified [CH00].

Testing could be manual or ideally automatic. Manual testing gives the tester flexibility to control the testing process and allows the tester to check the programs possibly completely, but it is somewhat laborious where the tester has to repeat one operation again and again. Automatic testing free testers from exhaustive repetition. For purpose of automatic test, the specification should be chosen in a formal way.

Haskell is suitable for automatic testing. It takes this advantage from its nature as a functional programming language. Free of side-affects, Haskell is not concerned with state transfer in an execution. At the same time, as a term language, it also

plays the part of the specification language.

QuickCheck was designed by Koen Claessen and John Hughes [CH00] to test Haskell programs; it is based on the technology of property satisfaction checking. Programmers write a function along with some other specifications which are used to formulate the properties of that function. In the QuickCheck environment those specifications are tested to find whether the properties are satisfied by the function. All the programs, both function and specifications, are written in Haskell language. The random input is automatically generated by the QuickCheck system.

3.1.1 How to specify the properties

QuickCheck was designed to test the functions' properties which are in the form of a set of parameterized assertions. To study how to check those properties, let us take a look at a simple example [CH00]. Function reverse is defined to reverse a list of some type. It can be found in the Prelude module of the Haskell language. To reverse the order of an list of elements correctly, function reverse should obey three basic laws [CH00]:

```
reverse [x] = [x]
reverse (xs ++ ys) = reverse ys ++ reverse xs
reverse (reverse xs) = xs
```

The first two laws specify the situations in which the list is just only one element, and when the list consist of two sublists by concatenating them together.

To be tested by QuickCheck, we need to write some assertions for those laws in the form of QuickCheck properties.

```
prop_RevUnit x =
  reverse [x] == [x]
prop_RevApp xs ys =
  reverse (xs ++ ys) == reverse ys ++ reverse xs
prop_RevRev xs =
  reverse (reverse xs) == xs
```

Here, these definitions are not different from general Haskell expressions, all of them return a value of boolean type. If it is true for all arguments the property holds. We pass these assertions as arguments to QuickCheck and run it in an interpreter environment, such as Hugs or GHCi, QuickCheck will generate 100 data values randomly for it, if it pass them all, a message of "OK, passed 100 tests" will be reported

to the user. otherwise QuickCheck will give a counterexample to show the test failed. Since the test is based on the evaluation of expressions, and we see those laws were written in overloaded form, the testers have to provide more information: we need to define the types of the arguments explictly. For examples, if we want to test the function reverse on list of Int, Following type signatures are necessary

```
prop_RevUnit :: Int → Bool
prop_RevApp :: [Int] → [Int] → Bool
prop_RevRev :: [Int] → Bool
```

For convenience, types informations can be presented in alternative way:

```
prop_RevUnit x =
  reverse [x] ≡ [x]
    where types = x :: Int
```

The examples show how to represent the properties for the laws of Haskell functions and how the result is reported. In fact QuickCheck as a practically useful tool can do not only that.QuickCheck is also capable of observing test case distribution.

Observing Test Case Distribution

QuickCheck does this by incorporating special functions into the properties. Each time the properties are tested the trivial cases are collected, and a summary is displayed when tests are finished.

```
trivial :: Testable a ⇒ Bool → a → Property
```

is the function used to counting the trivial cases

The example [CH00] shows how it works.

```
prop_Insert x xs =
  ordered xs ==>
    null xs 'trivial' ordered (insert x xs)
      where types = x :: Int
```

Test cases for which the value of null xs is True are classified as trivial. Thus the QuickCheck output might be

OK, passed 100 tests (58 % trivial)

The current version of QuickCheck defines 4 such observations. They are:

```
label :: Testable a ⇒ String → a → Property
collect :: (Show a, Testable b) ⇒ a → b → Property
```

```
classify :: Testable a ⇒ Bool → String → a → Property
trivial :: Testable a ⇒ Bool → a → Property
```

3.1.2 Create the data generators

Testing data of QuickCheck is randomly and automatically produced by data generator.

```
class Arbitrary a where
  arbitrary :: Gen a
  coarbitrary :: a → Gen b → Gen b
```

The above is a type class that makes data generator of polymorphic type a. Any type that could be tested in QuickCheck has an instance of Arbitrary class. It provides two operations: arbitrary is the type of Gen a which represent a generator, coarbitrary helps to generate function types.

```
newtype Gen a = Gen (Int → StdGen → a)
```

where StdGen defined as an abstract type

Generators of some basic types of Bool, Int, Float, Double, tuple, list were predefined in QuickCheck.

Following example shows how to define a data generator for Int type.

```
instance Arbitrary Int where
  arbitrary = sized $ λn → choose (−n, n)
  coarbitrary n = variant (if n ⩾ 0 then 2 ∗ n else 2 ∗ (−n) + 1)
```

this definition produces a generator for Int type by calling function choose.

Sometimes users needs to define custom generators for the new types introduced. QuickCheck provides some combinators to make the definitions easier.

The simplest combinator is oneof, which takes one from a list of generators which have equal chances to be selected. For example, the generator of type Bool is defined:

```
instance Arbitrary Bool where
  arbitrary = oneof [return True, return False]
```

It also can be defined through an alternative way

```
instance Arbitrary Bool where
  arbitrary = elements [True, False]
```

frequency lets the user specify the frequency with which each alternative is chosen [CH00]. Let's revisit the BinTree type in last chapter and produce a data generator for it by using function frequency.

```
data BinTree a = Leaf a
  | Branch (BinTree a) (BinTree a)
instance Arbitrary a ⇒ Arbitrary (Tree a) where
  arbitrary = sized arbTree
arbTree 0 = liftM Leaf arbitrary
arbTree n = frequency [(1, liftM Leaf arbitrary),
  (8, liftM2 Branch (arbTree (n 'div' 2)) (arbTree (n 'div' 2)))]
```

Function frequency is defined as

```
frequency :: [(Int, Gen a)] → Gen a
```

It chooses a generator from the list randomly, but weights the probability of choosing each alternative by the factor given.

QuickCheck is also capble of testing functions. By defining extensional equality (===) as

```
(f === g) x = f x ≡ g x
```

and given property

```
prop_CompAssoc f g h =
  f ∘ (g ∘ h) === (f ∘ g) ∘ h
```

we can testing the associativity of functions.To test function types, there are two things one should know. First, it is impossible to report a counterexample of function types. The solution is to print function values as a constant string like "<<functions>>". The second thing is one needs to define the data generators for function type.

function type of Int → StdGen → (a → b) represents Function generator of type Gen (a → b), It is equivalent to a → Int → StdGen → b by reordering the parameters. And it can be further rewritten as a → Gen b. So we define a function

```
promote :: (a → Gen b) → Gen (a → b)
```

This can be used to create generators for function types. One already noticed that in Arbitrary class there are two methods, another one is coarbitrary, It modifies a generator from it's first argument, it can be thought of as a generator transformer. Given functions of promote and method coarbitrary we can define:

```
instance (Arbitrary a, Arbitrary b) ⇒ Arbitrary (a → b) where
  arbitrary = promote (`coarbitrary` arbitrary)
  coarbitrary f gen = arbitrary >>= ((`coarbitrary` gen) ∘ f)
```

If one wants to generate random values for type a, they just need to define the method of arbitrary, while if want to generate functions, they need to define both of arbitrary and coarbitrary methods. By defining coarbitrary different values are interpreted as independent generator transformers. This can be done by variant

```
variant :: Int → Gen a → Gen a
```

for natural numbers i and j, $i \neq j$, variant i g and variant j g are independent generator transformers. A definition on boolean type gives a concise example of how to use variant.

```
instance Arbitrary Bool where
  arbitrary = elements [True, False]
  coarbitrary b = if b then variant 0 else variant 1
```

After QuickCheck, Koen Claessen and John Hughes designed another lightweight testing tool for Haskell: SmallCheck. Basically SmallCheck is very similar to QuickCheck, such as the idea of using type-based data generators, and the way of properties expressed and reporting the testing results, however it also improves on QuickCheck in many ways. Instead of using randomly generated values, SmallCheck tests properties for finitely many values up to some depth, progressively increasing the depth used [CH06b]. This mechanism ensures that any counter-examples found are minimal. Writing properties of SmallCheck for user-defined types is very easy, and properties use existential as well as universals. More advantages and usages are noted in the user guide [CH06b]. In this thesis we will concentrate on QuickCheck, which is enough to show the basic ideas of testing of type instances in a automatic Haskell program testing environment.

3.2 Testing the laws of type classes in QuickCheck

After reviewing the type classes and QuickCheck, this section will use following examples to illustrate how to test a instance of a type class by means of QuickCheck.

Maybe type is a standard data type which is pre-defined in the Prelude of Haskell. It is well known a Functor. We already saw its instance of Functor class in the previous chapter.

Maybe is not a pre-defined type of QuickCheck, If we want to make it testable, we must make a data generator for it by instancing it in the Arbitrary class.

```
instance (Arbitrary a) ⇒ Arbitrary (Maybe a) where
  arbitrary = frequency [(1, return Nothing),
    (2, liftM Just arbitrary)]
  coarbitrary Nothing = variant 0
  coarbitrary (Just n) = variant 1 ∘ coarbitrary n
```

As previously noted Maybe a is a polymorphic type itself. To show the work of QuickCheck, I test Maybe Int type here. As a functor, Maybe Int must satisfy two laws: id morphism and morphism composition.

```
prop_id fa = fmap id fa ≡ fa
  where types = fa :: (Maybe Int)
prop_comp f g fa = fmap g (fmap f fa) ≡ fmap (g ∘ f) fa
  where types = (f :: Int → Int, g :: Int → Int, fa :: Maybe Int)
```

Note that "types" in the **where** clause was used to provide a place to restrict the types of parameters, in this case the types of f, g, fa.

One more example here is the Tree type. The procedure of checking Tree type involves more features of QuickCheck such as control over the distribution of generated values and limitation on size of values. Define a Tree type and Declare an instance of Functor

```
data Btree a = Nil
   | Branch (Btree a) a (Btree a)
  deriving (Eq, Show)
instance Functor Btree where
  fmap f Nil = Nil
  fmap f (Branch l a r) = Branch (fmap f l) (f a) (fmap f r)
```

In the data generator for type Btree, function sized imposes a bound to limit the number of nodes in the generated trees.

```
instance Arbitrary a ⇒ Arbitrary (Btree a) where
  arbitrary = sized arbTree
```

```
    coarbitrary Nil =
      variant 0
    coarbitrary (Branch t1 l t2) =
      variant 1 ∘ coarbitrary t1 ∘ coarbitrary l ∘ coarbitrary t2
  arbTree 0 = return Nil
  arbTree n = frequency [(1, return Nil),
    (8, liftM3 Branch (arbTree (n `div` 2))
      arbitrary (arbTree (n `div` 2)))]
```

Running the QuickCheck to test the Id and composition laws of Btree functor for Int type.

```
prop_id_tree fa = fmap id fa ≡ fa
  where types = fa :: (Tree Int)
prop_comp_tree f g tre = fmap g (fmap f tre) ≡ fmap (g ∘ f) tre
  where types = (f :: Int → Int, g :: Int → Int, tre :: Tree Int)
```

This provides the evidence that Maybe and Btree types can be proved to be functors after including in Functor class and checking them in QuickCheck.

Monad is another class that we use to show how the instances of a class are tested by QuickCheck.

Monad laws are three basic rules that all monads must obey. left unit right unit. The third one is the composition which show the associative of morphism.

The following functions specify the properties of Maybe monad in QuickCheck's form.

```
prop_leftunit_maybe f a = (return a >>= f) ≡ f a
  where types = (f :: Int → Maybe Int, a :: Int)
prop_rightunit_maybe f = f >>= return ≡ f
  where types = f :: Maybe Int
prop_comp_maybe f g h = f >>= (λx → g x >>= h) ≡ (f >>= g) >>= h
  where types = (f :: Maybe Int, g :: Int → Maybe Int, h :: Int → Maybe Int)
```

The function constructor → was not defined to be a monad internally. By declaring it an instance of Monad one can check whether it is a monad. To be included in Monad, → should be a functor first, we just skip the step of instance of Functor.

```
prop_Monad_lUnit f x y = (return x >>= f) y ≡ (f x) y
  where types = (f :: Int → (→) Int Int, x :: Int, y :: Int)
```

```
prop_Monad_rUnit f x   = (f >>= return) x ≡ f x
  where types = (f :: (→) Int Int, x :: Int)
prop_Monad_comp f g h x = (f >>= (λx → g x >>= h))
  x ≡ ((f >>= g) >>= h) x
  where types = (f :: (→) Int Int,
    g :: Int → (→) Int Int,
    h :: Int → (→) Int Int, x :: Int)
```

The above examples illustrate how to justify the type class rules for different types. One obvious inconvenience is types must be specified explictly in a where clause. For example: the property prop_mMonad_comp takes four parameters and one has to give all of them the type signatures.

The following steps will improve the specifications written down in Haskell language and make them more general.

First we repeat the three properties which specify the laws of Monads in new form

```
prop_Monad_lUnit f y   = (λx → return x >>= f) y ≡ f y
prop_Monad_rUnit f     = (f >>= return) ≡ f
prop_Monad_comp f g h = (f >>= (λx → g x >>= h))
                      ≡ ((f >>= g) >>= h)
```

By type inference, we obtain three type signatures for the above functions. Respectively they are

```
prop_Monad_lUnit :: (Eq (m b), Monad m) ⇒ (a → m b) → a → Bool
prop_Monad_rUnit :: (Eq (m a), Monad m) ⇒ m a → Bool
prop_Monad_comp :: (Eq (m b), Monad m) ⇒ m a → (a → m a1) →
  (a1 → m b) → Bool
```

To test these properties, the corresponding type should be given when running QuickCheck. When test left unit property of Monad for some type one should run the command:

quickCheck (prop_Monad_lUnit :: (Int -> Maybe Int) -> Int -> Bool)

or

quickCheck (prop_Monad_lUnit :: (Int -> Maybe Char) -> Int -> Bool)

for type Maybe Int or Maybe Char

There are some other interesting type classes I like to formalize. In abstract algebra, a monoid is an algebraic structure with a single, associative binary operation and an identity element. Its formal definition is:

A monoid is a set M with binary operation

$* : M \times M \rightarrow M$, obeying the following axioms:

Associativity: for all a, b, c in M, $(a * b) * c = a * (b * c)$

Identity element: there exists an element

e in M, such that for all a in M, $a * e = e * a = a$.

In module Data.Monoid a type class was defined with name of Monoid and some types were defined as its instances.

```
class Monoid a where
  e :: a
  op :: a → a → a
instance Monoid [a] where
  e = []
  op = (++)
instance Monoid (a → a) where
  e = id
  op = (∘)
instance Monoid Integer where
  e = 1
  op = (*)
```

obviously Monoid class has the following properties to represent the laws:

```
prop_monoid_lid op x = e `op` x ≡ x
prop_monoid_rid op x = x `op` e ≡ x
prop_monoid_assoc op x y z = (x `op` y) `op` z ≡ x `op` (y `op` z)
```

Applicative is a functor with application which was defined in Haskell library. It describes a structure intermediate between a functor and a monad: it provides pure expressions and sequencing, but no binding. This class pre-includes the Maybe,[], IO and → etc..

```
class Applicative f where
  pure :: a → f a
  ⊛ ::f (a → b) → f a → f b
```

Any instances should satisfy the following laws:

identity: pure id ⊛ v = v

composition: pure (∘) ⊛ u ⊛ v ⊛ w = u ⊛ (v ⊛ w)

homomorphism: pure f ⊛ pure x = pure (f x)

interchange: u ⊛ pure y = pure ($ y) ⊛ u

The laws could be formalized in QuickCheck as

```
prop_applicative_id v = (pure id ⊛ v) ≡ v
prop_applicative_comp u v w = (pure (∘) ⊛ u ⊛ v ⊛ w) == (u ⊛ (v ⊛ w))
prop_applicative_homo u f x = (pure f ⊛ pure x) ≡ asTypeOf (pure (f x)) u
prop_applicative_intch u y = (u ⊛ pure y) ≡ (pure ($y) ⊛ u)
```

3.3 Testing for some predefined classes

Besides the Functor and Monad classes, some standard type classes and types were also defined in Haskell Prelude. Most of these classes have a mathematical foundation behind them. We talk about them in this section and find the proper way to present the laws that make them what they are.

Eq class

Previously we already introduced the definition of Eq class. All basic datatypes except for functions and IO are instances of this class. Instances of Eq can be derived for any user-defined datatype whose constituents are also instances of Eq [PJ+03]

In prelude it was defined in an alternative way.

```
class Eq a where
  (≡), (≢) :: a → a → Bool
  x ≢ y = ¬ (x ≡ y)
  x ≡ y = ¬ (x ≢ y)
```

This declaration gives default method declarations for both /= and ==, each being defined in terms of the other. If an instance declaration for Eq defines neither == nor /=, then both will loop. If one is defined, the default method for the other will make use of the one that is defined. If both are defined, neither default method is used [PJ+03]

Eq is superclass of some other type classes. such as Num, Ord.

Eq encapsulates a mathematical structure of equivalence relation with the laws of: Reflexivity, Symmetry and Transitivity.Given equilvalence relation R, It's laws:

Reflexivity: a R a

Symmetry: if a R b then b R a

Transitivity: if a R b and b R c then a R c.

The laws was expressed as following QuickCheck Assertion.

```
prop_Eq_Reflexive r a       = True ==> a `r` a
prop_Eq_Symmetric r a b     = (a `r` b) ==> (b `r` a)
prop_Eq_Transitive r a b c  = ((a `r` b) ∧ (b `r` c)) ==> a `r` c
```

To be tested one must give their types

```
prop_Eq_Reflexive :: (Testable a) ⇒ (t → t → Bool) → t → Property
prop_Eq_Symmetric :: (t → t → Bool) → t → t → Property
prop_Eq_Transitive :: (t → t → Bool) → t → t → t → Property
```

when one executes the check they has to offer the relations explicitly as parameter to the property being checked. Besides that one needs to supply the types of the rest of the type signature as well.

For example when checking the symmetric property of an equivalence relation, one must run the following command:

```
quickCheck ((prop_Eq_Symmetric (==)) :: Int -> Int -> Property)
```

Num class

Numeric types are important in almost every programming language, Haskell Prelude defines the most basic numeric types: fixed sized integers (Int), arbitrary precision integers (Integer), single precision floating (Float), and double precision floating (Double).

Tthe type class Num defines the arithmetic operations that those Numeric types share like (+) (-) (*),but not (/). It also provides the some other operations such as abs which computes absolute value, fromInteger which converts a Integer to any other numeric values.

Num encapsulates the mathematical structure of a (not necessarily commutative) ring, with the laws of Associativity, Identity(Left and Right), Distributivity.

Writing properties for these laws is straightforward.

```
prop_Num_Associative a b c = a * (b * c) ≡ (a * b) * c
prop_Num_rIdentity a       = a * 1 ≡ a
prop_Num_lIdentity a       = 1 * a ≡ a
prop_Num_Distributive a b c = a * (b + c) ≡ a * b + a * c
```

Ord class

Ord is defined as

```
class (Eq a) ⇒ Ord a where
        compare              :: a → a → Ordering
        (<), (⩽), (⩾), (>) :: a → a → Bool
        max, min :: a → a → a

    compare x y
     | x ≡ y = EQ
     | x ⩽ y = LT
     | otherwise = GT

    x ⩽ y                = compare x y ≢ GT
    x < y = compare x y ≡ LT
    x ⩾ y = compare x y ≢ LT
    x > y = compare x y ≡ GT
```

It is used for totally ordered datatypes, All basic data types except for function and IO are instances of this class. User defined data types also could be included in this class if they obey the constraint of ordering.

The Ordering datatype allows a single comparison to determine the precise ordering of two objects.

Ord encapsulates a mathematical structure of total order with the laws of Reflexivity, Antisymmetry and Transitivity. Another law is that any two elements in the set are comparable.

The following expressions formulate the properties of the Ord class.

```
prop_Ord_Reflexive a       = a ⩽ a
prop_Ord_aSymmetric a b = ((a ⩽ b) ∧ (b ⩽ a)) ==> a ≡ b
prop_Ord_Transitive a b c = ((a ⩽ b) ∧ (b ⩽ c)) ==> a ⩽ c
prop_Ord_Total a b         = (a ⩽ b) ∨ (b ⩽ a)
```

QuickCheck is a practical testing tool for Haskell language; users write the properties using the same language as the programs. The testing data was automatically and randomly generated. Users control the data distribution to some degree and get the testing result in a short time. Although QuickCheck cannot guarantee the correctness of a program completely, it will still give testers the chance to find and eliminate most bugs of programs. A more reliable way to improve the software quality is via formal methods. This requires the program specifications to be written in some kind of logic, so they can be verified in a mathematical way. In the next chapter we discuss how to prove the type classes' laws by Isabelle, which is a theorem prover.

Chapter 4

Verification of Instances in Isabelle

This chapter introduces Isabelle/HOL [NPW02] and discusses how to verify classes laws using it.

4.1 Overview of Isabelle/HOL

The Isabelle theorem prover is an interactive theorem proving framework; various logics are included in it, such as several first-order logic, simply type theory, and Zermelo-Fraenkel set theory [Pau89, Pau90b]. Each new logic is formalized within Isabelle's meta-logic; new types and constants express the syntax of the logic, while new axioms express its inference rules [Pau90a]

Isabelle/HOL [NPW02] is the specialization of Isabelle for HOL [NPW02], which refers to Higher-Order Logic. It provides some useful features to facilitate theory definitions and proving within a pre-defined library.

One of reasons to choose Isabelle/HOL [NPW02] is that a theory could be constructed easily from the specifications written in Haskell language due to the fact of:

$$\text{HOL} = \text{Functional Programming} + \text{Logic [NPW02]}$$

A typical Isabelle/HOL [NPW02] theory file contains definitions, lemmas and theorems, and proof scripts.

4.1.1 Types in Isabelle/HOL

HOL is a simply typed logic whose type system resembles that of functional programming languages like ML [MTM97] or Haskell [HJW+92] [NPW02]. Thus, there are

basic types, such as bool, nat, etc.., type constructors which are used to build new types, function types which are denoted by => and type variables which are in a form of polymorphism:'a,'b....

Inductive datatypes are part of almost every application of Isabelle/HOL [NPW02]. As such, packages are provided to facilitate datatype definitions. Its mode of definition is: Users give simple description of new inductive types using a notation similar to ML or Haskell [HJW+92, NPW02]; the system then automatically generates a sizeable amount of characteristic theorems.

A general datatype specification in Isabelle/HOL [NPW02] is of the following form [BW99]:

$$\textbf{datatype } (\vec{\alpha})t_1 = C_1^1\tau_{1,1}^1 \ldots \tau_{1,m_1^1}^1 \mid \ldots \mid C_{k_1}^1\tau_{k_1,1}^1 \ldots \tau_{k_1,m_{k_1}^1}^1$$

$$\vdots$$

$$\textbf{and} \qquad (\vec{\alpha})t_n = C_1^n\tau_{1,1}^n \ldots \tau_{1,m_1^n}^n \mid \ldots \mid C_{k_n}^n\tau_{k_n,1}^n \ldots \tau_{k_n,m_{k_n}^n}^n$$

where $\vec{\alpha} = (\alpha_1), \ldots, \alpha_h$ is a list of type variables, C_i^j are distinct constructor names and $\tau_{i,i'}^j$ are admissible types containing at most the type variables $\alpha_1, \ldots, \alpha_h$. A type τ occurring in a datatype definition is admissible if and only if

- τ is non-recursive, i.e. τ does not contain any of the newly defined type constructors, or
- $\tau = (\vec{\alpha})t_{j'}$ where $1 \leqslant j' \leqslant n$, or
- $\tau = (\tau_1^{'}), \ldots, \tau_{k'}^{'}t'$ where t' is the type constructor of an already existing datatype and $\tau_1^{'}, \ldots, \tau_{h'}^{'}$ are admissible types.
- $\tau = \sigma \rightarrow \tau'$, where τ' is an admissible type and σ is non-recursive(i.e. the occurrences of the newly defined types are strictly positive)

If some $(\vec{\alpha})t_{j'}$ occurs in a type $\tau_{i,i'}^j$ of the form

$$(\ldots, \ldots (\vec{\alpha})t_{j'} \ldots, \ldots)t'$$

this is called a nested occurrence.

Types in HOL must be non-empty [BW99]. Each of the new datatypes $(\vec{\alpha})t_j$ with $1 \leqslant j \leqslant n$ is non-empty iff it has a constructor C_i^j with the following property: for all argument types $\tau_{i,i'}^j$ of the form $(\vec{\alpha})t_{j'}$ the datatype $(\vec{\alpha})t_{j'}$ is non-empty.

If there is no nested occurrences of the newly defined datatypes, to be non-empty at least one of the newly defined datatypes $(\vec{\alpha})t_j$ must have a constructor C_i^j without recursive arguments a base case. If there are nested occurrences,a datatype can still be non-empty without having a base case itself.

For example, the datatype definition of list

datatype 'a list = Nil ("[]") [] is syntax annotation of empty list.
| Cons 'a "'a list" (**infixr** "#" 65) "#" infix operator of Cons

Sometimes it is unnecessary to define new types. For these cases one can use **type synonyms**. They are created by a types command

types number = nat
gate = "bool ⇒ bool ⇒ bool"
('a,'b)alist = "('a*'b)list"

Type synonyms are intended to improve the readability of theories,

4.1.2 Functions and Terms

Isabelle/HOL provides two mechanisms to define recursive functions [NPW05]

Primitive recursion applies only on datatypes.

Its general form is the keyword **primrec** is followed by a list of equations [NPW02]

$$f\ x_1\ \ldots\ (C\ y_1\ \ldots\ y_k)\ \ldots\ x_n = \tau$$

where C is a type constructor of a datatype t. All recursive calls of f in τ are

$$f\ \ldots\ y_i\ \ldots .$$

This ensures f terminates, since one argument becomes smaller with every recursive call [NPW02]. It is required that there is at most one equation for each constructor C_i for a datatype t, τ can contain on free variables on the left-hand. Functions are in arbitrary order. It is unnecessary to define functions for every constructor of a datatype, for any that are omitted, function is defined to return a default value.

Following example shows a primitive recursive function definition

consts app :: 'a list ⇒ 'a list ⇒ 'a list
primrec
"app [] ys = ys"
"app (x # xs) ys = x # app xs ys"

Primitive recursion is suitable for total functions that have a natural recursive definition. But there are some drawbacks to prevent defining all functions in a primitive recursive form.

The first limitation is that the set of primitive recursive functions does not

include every possible computable function.

The second limitation is because there must be at most one reduction rule for each constructor one can not use full pattern-matching.

In Isabelle/HOL, general recursive function could be defined by using the keyword recdef. It requires to provide a well-founded relation to control the recursion. Recursion does not need to apply on datatypes; termination is proved by showing that arguments of all recursive calls decrease under some relation.

It is hard to avoid incompleteness of function definitions by using recdef, patterns also overlap. Proper order of patterns disambiguates the overlapping, and functions return default value for missed patterns.

The Fibonacci function is a typical example to show a function definition using recdef.

```
consts fib :: "nat ⇒ nat"
recdef fib "measure(λn. n)"
  "fib 0 = 0"
  "fib (Suc 0) = 1"
  "fib (Suc(Suc x)) = fib x + fib (Suc x)"
```

A function measure of λn. n was embedded in this definition. It requires that the measure of the argument of fib on the left-side is strictly greater than that of the argument of each recursive call [NPW02]. This requirement was obviously satisfied because Suc (Suc x) is strictly greater than Suc x and x

measure is an operator provided by Isabelle/HOL to build well-founded relations. The package automatically proves the relation which was constructed by measure is well-founded. Isabelle/HOL defines 5 such operators.

- **less_than** is relation of "less than" on natural number
- **measure** f, where f is a map of $\tau \Rightarrow nat$
- **inv_image** R f is a generalisation of measure.
- R_1<***lex***>R_2 is the lexicographic product of two relations
- **finite_psubset** is the proper subset relation on finite sets

Terms in HOL are formed by applying functions on arguments.

Isabelle offers some basic structures which also are implemented in the Haskell programming language such as

-conditional expressions:

if b **then** t_1 **else** t_2

-Let expressions:

let x = t **in** u

-and case expressions:

case e **of** $c_1 \Rightarrow e_1 \mid ... \mid c_n \Rightarrow e_n$

4.1.3 Specifications and Proof

Specifications are represented in Isabelle as a theorem or a lemma; they are Isabelle expressions prefixed with keyword **theorem** or **lemma** respectively. The two keywords are interchangeable; the only concern is to emphasize the importance of some properties.

theorem *rev_rev*[*simp*] : "$rev(rev\ xs) = xs$"

- establishes a new theorem to be proved.
- gives the theorem the name *rev_rev*
- imposes the attribute simp on the theorem; declaring it as a simplification rule.

Given a theorem or lemma, the proving process in Isabelle is to apply a sequence of commands. For example: the proof of above theorem.

apply(induct_tac xs)

apply(auto)

For the complete example refer to [NPW02]. Proof of most theories is not so simple; it involves a sequence of commands, rules and method, sometimes requiring one to define and prove a set of lemmas that help to establish the ultimate theorem.

4.2 Suggestions for a translator from Haskell to Isabelle

To verify the Haskell programs we need to transfer the types, functions and specifications written in Haskell into the form of Isabelle/HOL [NPW02]. But HOL is a logic of total functions and is not suitable to express the non-strict semantics of Haskell directly. For example, it is hard for Isabelle/HOL [NPW02] to express the

lazy evaluations of the Haskell language. However our goal is not to translate every structure of Haskell to Isabelle; thus it is suitable to describe the semantics of Haskell functions that always terminate and that do not make essential use of laziness.

The translation from Haskell to a model in the theorem prover Isabelle/HOL [NPW02] is mostly syntactic and can be automated. This section of the thesis will discuss the similarities and differences between Haskell and HOL syntax. It also propose the rules for possible translations.

4.2.1 Rules for translating Terms

Rules for general structures

Functions play a major role in Haskell programming language. A program is composed of various function definitions and function calls. Functions could be defined curried or uncurried, the difference between these forms is the type of arguments. Function application associates to the left while the Function constructor $\rightarrow$ associates to the right, so as an example the function *add*

```
add :: Int → Int → Int
add x y = x * y
```

Its type Int $\rightarrow$ Int $\rightarrow$ Int has another equivalent form of Int $\rightarrow$ (Int $\rightarrow$ Int) and its application add 1 2 could be expressed as (add 1) 2. A function can be returned as a value by means of partial application of a curried function: (add 1). This was called section. Haskell has other program structures, such as **if** expression, **let** expression, **where** expression, **case** and λ etc.. They can be translated to HOL in the following manner.

$$(\!|\, \textbf{if } b \textbf{ then } t_1 \textbf{ else } t_2 |\!) := \textbf{if } (\!|b|\!) \textbf{ then } (\!|t_1|\!) \textbf{ else } (\!|t_2|\!)$$

$$(\!|(op\ e)|\!) := \%x \rightarrow x\ (\!|op|\!)\ (\!|e|\!)$$
$$(\!|(e\ op)|\!) := \%x \rightarrow (\!|e|\!)\ (\!|op|\!)\ x$$

$$(\!|\, \textbf{let } x_1 = t_1; \ldots; x_n = t_n \textit{ in } \mathbf{E} |\!) := \textbf{let } (\!|x_1|\!) = (\!|t_1|\!); \ldots; (\!|x_n|\!) = (\!|t_n|\!) \textit{ in } (\!|\mathbf{E}|\!)$$

$$(\!|\, f\ \alpha_1, \ldots, \alpha_n = \mathbf{E}(\vec{z}) \textbf{ where } z_1 = \mathbf{F}_1\ (\vec{\alpha_i}) \ldots z_n = \mathbf{F}_n\ (\vec{\alpha_j}) |\!) :=$$
$$\textbf{let } (\!|z_1|\!) = (\!|\mathbf{F}_1|\!)\ ((\!|\vec{\alpha_i}|\!)); \ldots; (\!|z_n|\!) = \mathbf{F}_n\ ((\!|\vec{\alpha_j}|\!)) = (\!|\mathbf{F}_n|\!)\ ((\!|\vec{\alpha_n}|\!))\ \textit{in}\ (\!|\mathbf{E}|\!)((\!|\vec{z}|\!))$$

$$(\!|\textbf{case } e \textit{ of } p_1 \rightarrow \mathbf{E}_1; \ldots; p_n \rightarrow \mathbf{E}_n|\!) := \textbf{case } (\!|e|\!) \textit{ of } (\!|p_1|\!) \Rightarrow (\!|\mathbf{E}_1|\!) \mid \ldots \mid (\!|p_n|\!) \Rightarrow (\!|\mathbf{E}_n|\!)$$

$$(\!|\ \lambda x.\ \mathbf{E}\langle x\rangle\ |\!) := \%(\!|x|\!).\ (\!|\mathbf{E}|\!)\langle(\!|x|\!)\rangle$$

$$(\!|\ gd_1 = e_1 \mid gd_2 = e_2 \mid \ldots \mid gd_n = e_n\ |\!) :=$$
$$\textbf{if } (\!|gd_1|\!) \textbf{ then } (\!|e_1|\!) \textbf{ else if } (\!|gd_2|\!) \textbf{ then } (\!|e_2|\!) \textbf{ else if } \ldots \textbf{else } (\!|e_n|\!)$$

Notation $(\!|\_|\!)$ defines the conversion function from Haskell to Isabelle. Above structures are often seen in Haskell programs. Understanding their meanings is helpful to translate them correctly into HOL. Semantics of terms could be defined by giving identities that relate those constructs to case expressions

Semantics of conditional expression

$$\textbf{if b then T else F} = \textbf{case b of } \{\mathsf{True} \rightarrow \mathsf{T}; \mathsf{False} \rightarrow \mathsf{F}\}$$

Semantics of **let** expression

$$\textbf{let } p_1 = e_1; \ldots; p_n = e_n \textbf{ in } e_0 = \textbf{let } (p_1, \ldots, p_n) = (e_1, \ldots, e_n) \textbf{ in } e_0$$

$$\textbf{let } p = e_1 \textbf{ in } e_0 = \textbf{case } e_1 \textbf{ of } p \rightarrow e_0$$

where no variables in p appears free in e_1

For λ abstraction, the following identity holds:

$$\lambda p_1 \ldots p_n \rightarrow e = \lambda x_1 \ldots x_n \rightarrow \textbf{case } (x_1, \ldots, x_n) \textbf{ of } (p_1, \ldots, p_n) \rightarrow e$$

where the x_i are new identifiers.

Semantics of partial function application(section) is defined by a λ-abstractions. HOL does not provide syntax for sections, but a translation to λ-abstractions will properly preserve the semantics of Haskell. Let expressions introduce a nested, lexically-scoped, mutually-recursive list of declarations [PJ+03] The declarations scope the expression e and the right hand side from declarations. In Haskell another way to create nested scope in an expression is **where** clause. A **where** clause is only allowed at the top level of a set of equations or case expression. The same properties and constraints on bindings in let expressions apply to those in **where** clauses [HPF99].

We suggest translating some **where** clauses into HOL as **let** expressions. However there are some limitations. For example:

```
g x y | y > z = ...
      | y ≡ z = ...
      | y < z = ...
      where z = x * x
```

A **let** expression can not be used to express above statement. These two forms of local declarations look similar, but in fact they have some differences, **let** is

a expression and **where** is only the part of function declaration and **case** expression [HPF99].

Function bindings in a **let** expression are allowed, an example in Haskell

```
let y = a * b
  f x = (x + y) / y
in f c + f d
```

This expression can be used in HOL after changing some operators.

Not all pattern bindings can be used in **let** expressions, for example If we have a datatype of State

datatype ('a,'s) State = ST "'s ⇒ 'a* 's"

We can define a function containing pattern binding

```
(ST g) = gg a
```

This pattern binding is allowed in Haskell, but not in HOL. One solution to this problem is to define a deconstructor of of ST naming unST:

```
unST (ST x) = x
  g' = gg a
  g = unST g
```

Rules for recursion

Functions provided by type classes are important to verifications, as they are overloaded by specific types. When proving the correctness in Isabelle/HOL, those functions must be translated into Isabelle/HOL. In addition to the overloaded class methods, some auxiliary functions are also necessary to be translated.

To define the rules for translating functions we must understand how functions are defined both in Haskell and Isabelle/HOL.

Function types in Haskell are constructed by type constructor (→), they are in the form of $\tau_1 \rightarrow \tau_2$. Since τ_i could be function types themselves, function types were declared recursively. A function is defined in Haskell in the following form

name pattern1 pattern2 ... pattern = expression (n /=0)

where name is the function name and patterns play the roles of parameters. The functions declaration and definition in HOL are similar to those of Haskell, the obvious difference is the function type constructor (⇒); both (→) and (⇒) associate

to right. While type signatures can be ignored in Haskell, they must be explicitly given in HOL. Type inference does not infer the types of functions defined on top level.

In both Haskell and Isabelle/HOL, inductive datatypes lead to recursive functions. Isabelle/HOL provides two mechanism to define recursive functions: primitive recursion and well-founded recursion. Primitive recursion applies only on datatypes, the termination relies on the one fixed argument becoming smaller [NPW02]. There must be at most one reduction rule for each constructor [NPW05]. Well-founded recursion is a more general method: its termination is guaranteed by a well-founded relation. Besides the recursive function definitions, Isabelle/HOL provides constant definitions which can be used to formalize some structures that primitive and well-founded recursion can not, an example of which we will show later.

Analyzing function definitions of Haskell provide the guidance on how to translate functions and what form of recursions in Isabelle/HOL should be chosen. Two things should be addressed: function bindings and pattern matching (discussed in the previous section). Function binding binds a variable to a function value [PJ+03].

$$x\ p_{11} \ldots p_{1k}\ match_1$$
$$\ldots$$
$$x\ p_{n1} \ldots p_{nk}\ match_n$$

each p_{ij} is a pattern, and where each $match_i$ is of the general form: $= e_i$ where $decls_i$. $match_i$ is also a form of a guard expressions.

The meaning of function binding expressed by **case** expression is

$$x = \lambda x_1 \ldots x_k \rightarrow case\ (x_1, \ldots, x_k)\ of$$
$$(p_{11}, \ldots, p_{1k})\ match_1$$
$$\ldots$$
$$(p_{n1}, \ldots, p_{nk})\ match_n$$

This can be expressed by HOL. The $match_i$ in the form of guards can be represented by nested conditional statements.

Functions also can be defined in **let** and **where** expressions locally.

The following rules determine how functions are translated:

$(\!|f\ p_1\ \vec{x} = e_1\ \ldots\ f\ p_n\ \vec{x} = e_n|\!) :=$

primrec"$(\!|f|\!)\ (\!|p_1|\!)\ (\!|\vec{x}|\!) = (\!|e_1|\!) \ldots (\!|f|\!)\ (\!|p_n|\!)\ (\!|\vec{x}|\!) = (\!|e_n|\!)$"

where patterns p_i are in the form of $C_i\ \vec{y}$ for the datatype $t\ \vec{\alpha}\ =\ C_i\ \vec{\tau}$. One requirement for translating recursive functions in Haskell to primitive recursion in HOL is C_i is distinct in pattern $\vec{p}$, the reason is the HOL restriction of at most one

reduction rule for each constructor. Some functions with full pattern matching can not be processed in this thesis, for example a function defined on list type

```
f [] = []
f [x] = p x
f (x : xs) = q x : f xs
```

Functions defined in this form are suitable to be translated to general recursion, which need an associated well-founded relation to ensure termination. In this case we can use relation of "*ensure* $\lambda xs.$ *length* xs". HOL provides some operators to generate the relations. How to use them for formalizing Haskell functions will be a topic for future discussion.

$$(\!|f\,\vec{x} = C\ \$\ e|\!) := \textbf{constdefs}\ f :: \tau \quad "(\!|f|\!)\ (\!|\vec{x}|\!) = (\!|C|\!)(\!|e|\!)"$$

$$(\!|f\,\vec{x} = e|\!) := \textbf{constdefs}\ f :: \tau \quad "(\!|f|\!)\ (\!|\vec{x}|\!) = (\!|e|\!)"$$

The above two rules translate some Haskell code to constant function definitions. The first one says that when no arguments contain constructors of given type; and the function returns the value prefixed with a constructor, such functions are transformed to a constant function definition. One example is return of Monad.

The second rule translates general Haskell functions which have no pattern matching. The patterns used, and placement of the constructors of the arguments affect the choice of which rule is applied.

4.2.2 Types translation

Types are important to both Haskell and Isabelle/HOL. Haskell defines some base types in its prelude module, such as Int, Float, Bool, Char .... and also some composite types: tuple types $(t_1, t_2, \ldots, t_n)$, list types $[t_1]$, and function types $(t_1 \rightarrow t_2)$ where $t_1, \ldots, t_n$ are types themselves [Tho99]

Types definitions

It is hard for Haskell to define enumerate types such as a type representing days of week, or a type containing either a number or a string [Tho99] by means of composition of base types. All these types can be introduced by algebraic types via a **data** statement.

$$\textbf{data}\,[context\ \Rightarrow]\textbf{type}\,tv_1\ \ldots\ tv_i\ =\ con_1\ \tau_1 \ldots\ \tau_n$$

$$
\begin{array}{l}
\mid \ldots \\
\mid con_m\ \tau_1\ \ldots\ \tau_q \\
\quad [deriving]
\end{array}
$$

Given **type** is the type name or type constructor, tv_i are a set of type variables. con_i are a set of data constructors, each of which was followed by a list of type variables or type constants. [*context* $\Rightarrow$] restricts some type variables of tv_i to some type classes. [*deriving*] includes the new defined type into some type classes.

Types can be introduced through other ways; one is expressed by a **type** statement which creates a type synonym and it does not use new data constructors; another is expressed by statement of **newtype** which changes the type name and supply a data constructor.

Similarly Isabelle/HOL predefines a set of types, such as nat, the type of natural numbers, bool, the type of boolean and some composition type such as list, pair, tuple. To be able to describe the real world, Isabelle/HOL also provides mechanism to introduce new types. In section 3.1.1 we already saw the syntax for defining a new recursive datatype which has a set of constraints. Isabelle also provide type synonyms, defined via keyword types.

If we ignore the statements [context $\Rightarrow$], and [*deriving*] of Haskell language; the syntax difference on new datatype definition between Haskell and Isabelle/HOL is subtle. The type variables on the left hand side of HOL type definitions have different order from those of Haskell and are grouped in different form. They all consist of a type name(type constructor), a sequence of type variables on the left hand side, and a set of data constructors on the right hand that are all associated with some types(possibly none). Types defined in Haskell and Isabelle/HOL may each be polymorphic and both must be non-empty. We define the translation rules for types definitions below:

$$
\begin{array}{l}
(\!|\textbf{data}\ type\ \alpha_1\ \ldots\ \alpha_i\ =\ con_1\ \tau^1_{1,1}\ \ldots \tau^1_{1,m^1_1} \mid \ldots \mid con_m \tau^1_{k_1,1} \ldots \tau^1_{k_1,m^1_{k_1}} |\!) := \\
\textbf{datatypes}\ ((|\alpha_1|), \ldots, (|\alpha_2|))(|type|) \quad = \quad (|con_1|)(|\tau^1_{1,1}|) \ldots (|\tau^1_{1,m^1_1}|) \quad \mid \quad \ldots \quad \mid \\
(|con_m|)(|\tau^1_{k_1,1}|) \ldots (|\tau^1_{k_1,m^1_{k_1}}|)
\end{array}
$$

$$
\begin{array}{l}
(|\textbf{newtype}\ type\ \alpha_1\ \ldots\ \alpha_i = con_1\ \tau^1_{1,1}\ \ldots \tau^1_{1,m^1_1}|) := \\
\textbf{datatypes}\ ((|\alpha_1|), \ldots, (|\alpha_2|))\ (|type|) = (|con_1|)(|\tau^1_{1,1}|) \ldots (|\tau^1_{1,m^1_1}|)
\end{array}
$$

$$
(|\textbf{type}\ type\ \alpha_1, \ldots, \alpha_m = \tau|)_H\ := \textbf{types}((|\alpha_1|), \ldots, (|\alpha_m|))(|type|) = (|\tau|)
$$

These three rules are used to translate type definitions from Haskell to Isabelle/HOL. In implementation, one should be aware that in Isabelle/HOL type variables are in the form of $'\alpha_i$. For example: A type of Tree defined in Haskell:

data $Tree\ a\ =\ Leaf\ |\ Branch\ (Tree\ l)\ x\ (Tree\ r)$

translating to Isabelle/HOL datatype by rule 1

datatype $'a\ Tree\ =\ Leaf\ |\ Branch\ "'a\ Tree"\ 'a\ "'a\ Tree"$

From this example we can see that under the syntax of Isabelle/HOL, the algebraic types take form of "$'a\ Type$". Translating type definitions is straightforward, the semantics are also preserved in the sense that the Isabelle type contains exactly the finite fully-defined elements of the Haskell type.

Translating type signatures and type annotations

One advantage of the HM type system is type inference. It greatly reduce the amount of explicit type information that needs to be provided by a user for a Haskell program. But that does not mean that one can abandon the type information completely. In Haskell programs, type signatures are required when declaring functions, for example when one defines a type class with some operators. In Isabelle/HOL type signatures are required to be given explicitly. Isabelle/HOL provides some pre-defined base types, such as nat, int, real and bool and associated theories on those types. Not all built-in base types of Haskell are suitable to be translated to Isabelle/HOL [NPW02]. But some of them, such as Float,Double, and Bool can find their counterparts in Isabelle/HOL.The translation is natural and straightforward, Int $\rightsquigarrow$ int,Float $\rightsquigarrow$ real, Bool $\rightsquigarrow$ bool. For other base types of Haskell, one way is to extend the theory of Isabelle/HOL to include these types. This thesis concentrates on polymorphic types, and is concerned little about those base types of Haskell, so we pay more attention to composition types such as Tuples, Lists, Types of Functions.

$(\!|a|\!) :=' a$

$(\!|(a, b)|\!) := (\!|a|\!) \times (\!|b|\!)$

$(\!|[a]|\!) := (\!|'a|\!)\ (\!|list|\!)$

$(\!|a \rightarrow b|\!) := (\!|a|\!) \Rightarrow (\!|b|\!)$

$(\!|TyCon\ \vec{\alpha}|\!) := (\!|(\vec{\alpha})|\!)\ (\!|TyCon|\!)$

The above rules determine how to translate built-in types. When those types

appear in programs, the direct translation is easy to do and theories associated with those types will simplify programmers' job. Tuples and lists are built up by combining a number of pieces of data into a single object [Tho99]. In a tuple we can combine various values into a single object even of different types. Its general definition consists of components of simpler types: $(t_1, t_2, \ldots, t_n)$. From its definition a tuple comes with various number of components, tuples could be called pairs, triples, quadruples, quintuples .... For completeness, Haskell provides a nullary tuple (). It contains two members, $\perp$ and () [Bir98].

In HOL, an ordered pair (a,b) is of the type of $\tau_1 \times \tau_2$ with a's type being τ_1 and b's type being τ_2. Tuples are constructed by pairs nested to right so that $(\alpha_1, \alpha_2, \alpha_3)$ is equivalent $(\alpha_1, (\alpha_2, \alpha_3))$ and has a type of $\tau_1 \times \tau_2 \times \tau_3$ or equivalently in the form of $\tau_1 \times (\tau_2 \times \tau_3)$ HOL also provides a *unit* ().

List is defined to contain a sequence of values of the same types. In Haskell a list of some type is introduced by type constructor of []. For example, [Int] defines a list of Int type. It also plays the role of a data constructor. [] stands for the empty list, while [0] is a list of Int with only one element. HOL defines a list type through the **datatype** command. Its type constructor is List. [] in HOL just represents the empty list and a non-empty list is built by its data constructor of *Cons* or the infix operator #. Translation of user-defined datatypes is very natural, they take the same set of type constructors, data constructors, and type variables, All that remains is just to reconstruct them in the syntax of Isabelle/HOL. An example:

map :: (a → b) → [a] → [b] transform to HOL

map :: "('a ⇒ 'b) ⇒ ('a list ⇒ 'b list)"

4.2.3 Generating theorems

Programming with Haskell type classes includes type classes declaration, instance declaration and user-defined types. In this thesis we intend to check whether the instance of a class which is newly defined, or standard for some data types no matter built-in or user-defined satisfies the laws of that class. While Haskell type classes handle polymorphic types, this thesis will check monotypes.

In HOL properties which need to be verified were normally represented by theorems or lemmas. They reflect the characters of some datatypes and functions defined on them. To verify the laws of Haskell type classes, the specifications that express the laws should be provided. We can generate the theorems based on those specifications. The rule for translating specifications simple:

$$[\![Spec\ (\vec{\alpha}) = \mathbf{E}(\vec{f})]\!] := lemma\ [\![Spec]\!]\ \vec{\alpha}\ = [\![\mathbf{E}]\!]([\![\vec{f}]\!])$$

where $\vec{f}$ are the polymorphic functions defined in type classes, We Translate Haskell specifications into lemmas of HOL by substituting any polymorphic functions in specifications with monomorphic functions. For example:

A instance of Functor class of type Maybe.

```
instance Functor Maybe where
  fmap f Nothing = Nothing
  fmap f (Just a) = Just (f a)
```

We get monomorphic functions in HOL by translating polymorphic functions of Haskell for specified types.

```
consts
fmap_Maybe :: "('a => 'b) => ('a Maybe => 'b Maybe)"
primrec
  "fmap_Maybe f Nothing = Nothing"
  "fmap_Maybe  f (Just a) = Just (f a)"
```

The specification for functor id law:

checkFunctor_id fa = fmap id fa ≡ fa

Based on the above instance and specification of type Maybe for Functor type class, the following lemma says that the Maybe functor satisfies the id law.

```
lemma checkFunctor_id_Maybe : "fmap_Maybe id fa = fa"
```

4.2.4 Implementing a translator

A translator from Haskell to Isabelle/HOL is implemented based on the translating rules described in last section. It could be used to translate Haskell programs and specifications which are explicitly given by programmers along with the programs. Current version does not support all rules, such as **where** and Let expressions.

Translator gets the abstract syntax tree(AST) from Haskell source code by calling function parseModule. Top level structure of AST is a module storing the variable of type HsModule. By analyzing the **module**, translator constructs all parts that are needed to generate a complete theory file of Isabelle/HOL. Translating is a process involving several scans on AST and keeps a **module** as static environment(means without update). The reason to scan module more than once is some structures taking components from same term but they can not be done at same time.

Most Haskell functions will be translated to Isabelle versions. The functions that represent the specifications will be transfered to proof obligations such as lemmas

or theorems. Functions representing specifications are defined in Haskell with the names prefixed by a string of "check". To do so translator will easily distinguish specifications with other functions. More description refers to appendix.

4.3 Verify class properties in Isabelle/HOL

In this section we will study some examples to show how to verify the instances in Isabelle/HOL.

The type Maybe is predefined in Haskell prelude. The purpose of the Maybe type is to provide a method of dealing with illegal or optional values without terminating the program. It comes with a set of operations like isJust, fromJust, fromMaybe etc.. Besides this, it overloads fmap,$\gg=$ and return functions through the programming interface of type classes of Functor and Monad. In previous section we already translated Maybe type of Haskell and its Functor instance to HOL to illustrate the procedure of generating theorem. We will complete its proof for Functor laws and Monad laws.

```
data Maybe a = Nothing | Just a
```

By the translating rules, in HOL this type could be defined through keyword datatype.

```
datatype 'a maybe = Nothing | Just 'a
```

Revisiting the translation of function fmap

```
consts
fmap_Maybe :: "('a => 'b) => ('a Maybe => 'b Maybe)"
primrec
  "fmap_Maybe f Nothing = Nothing"
  "fmap_Maybe  f (Just a) = Just (f a)"
```

The function fmap was translated into primitive recursion because maybe is an algebraic type although it is not a recursive type. The second reason is that every constructor appears in pattern not more than once. This is also one of principles for automatic translation.

Specifications are not parts of Haskell programs. To express the Functor's properties users could write them in Haskell language explicitly, and for reason of distinguishing from general Haskell functions The names are prefixed with the text "check" such as inthe ollowing examples:

```
checkFunctor_id :: (Functor f, Eq (f a)) ⇒ f a → Bool
checkFunctor_id fa = fmap id fa ≡ fa

checkFunctor_comp :: (Functor f, Eq (f c)) ⇒ (a → b) → (b → c) → f a → Bool
checkFunctor_comp f g fa = fmap g (fmap f fa) ≡ fmap (g ∘ f) fa
```

These two specifications express the identity morphism and morphism composition of Functors. Their counterparts in HOL are lemmas for which we are ready to write some proof scripts.

Theorems which represent laws of Maybe functor and their proof scripts are:

```
lemma checkFunctor_id_Maybe: "fmap_Maybe id fa = fa"
apply(induct_tac fa)
    1. fmap_Maybe id Nothing = Nothing
    2. !!a:: 'a. fmap_Maybe id(Just a) = Just a
apply(auto)
    No subgoals!
done

lemma checkFunctor_comp_Maybe: "fmap_maybe g (fmap_maybe f
fa)
                                = fmap_maybe (g . f) fa"
apply(induct_tac fa)
  1. fmap_Maybe g (fmap_Maybe f Nothing)= fmap_Maybe (g . f)
Nothing
  2. !!a:: 'c. fmap_Maybe g (fmap_Maybe f (Just a))=fmap_Maybe
(g . f) (Just a)
apply(auto)
    No subgoals!
done
```

Proofs for Maybe functor are fairly easy, requiring just one step.

Let's see a more complex example with the State monad. State monad is adopted by Haskell to transfer programs' state internally. By using the state monad, programs can hide the state information which usually must be passed to functions as arguments. It is defined as a function of type $s \rightarrow (a, s)$, from a initial state, a value of type a was returned paired with a new state after computation. The type of State is:

```
type State s a = s → (a, s)
```

Overloading the function of $\gg=$ and return makes the State to be a Monad.

```
instance Monad (State s) where
  return a = λs → (a, s)
  (≫=) m f = λs →
    let (s', a) = m s
        m'      = f a
    in m' s'
```

Three specifications describe the basic properties of a monad along with their signatures.

Left unit.

```
checkMonad_lunit :: (a → m b) → a → Bool
checkMonad_lunit f x = (return x) ≫= f ≡ f x
```

Right unit.

```
checkMonad_runit :: m a → Bool
checkMonad_runit x = x ≫= return ≡ x
```

Bind composition.

```
checkMonad_comp :: m a → (a → m b) → (b → m c) → Bool
checkMonad_comp f g h = f ≫= (λx → g x ≫= h) ≡ (f ≫= g) ≫= h
```

When we take the Monad declaration from a predefined file which describes the built-in types and classes of Haskell, we almost have everything necessary to generate a theory of State Monad. The type State was declared as a type synonym of a function type $s \to (a, s)$, it can be redeclared as another type synonym in HOL.

```
types ('a,' s) State = "'s => 'a \<times> 's"
```

By the type signatures of function $\gg=$ and return, we get the monomorphic type of $\gg=$ and return defined on State. Observing the State type is not recursive and it has just one pattern we don't need define $\gg=$ and return as recursive functions. Renaming the $\gg=$ to a function name in text:

```
constdefs
```

```
return_state :: "'a => ('a,'s) State"
  "return_state x == (%y. (x,y))"
constdefs
  bind_state :: "('a,'s) State =>
    ('a => ('b,'s) State ) => ('b,'s) State"
  "bind_state st f == (%x. let(a,st')
    = st x in (let m' = f a in m' st'))"
```

The nested let expression in bind_state needs to be addressed. It was translated from the sequence of **let** expressions.

After translating the type and functions, generating proof obligations for monad laws is straightforward. The theorems or lemmas come from the specifications explicitly provided above. The procedure involves a sequence of function name substitution and without using specifications signatures (which sometimes can be ignored).

```
lemma checkMonad_lunit_state[simp]:
  "bind_state (return_state x) f = f x"
lemma checkMonad_runit_state:
  "bind_state f return_state = f"
lemma checkMonad_comp_state:
  "bind_state f (%x. bind_state (g x) h ) =
    bind_state (bind_state f g) h"
```

The complete proof refers to appendix. The appendix also include translations and proofs of Monadplus on some basic types. The example of MonadT is worth some description, the polymorphic function lift (m a → t m a)lifts one Monad to another. The proofs verify two laws which were suggested by ShengLing, Paul Hudak and Mark Jones [LHJ95]

$$lift \circ unit_m = unit_{tm}$$

$$lift\ (m\ 'bind'_m\ k) = lift\ m\ 'bind'_{tm}\ (lift \circ k)$$

The example lifts a Maybe monad to a State Monad, but not List, because we cannot define a list monad transformer [LHJ95].

The examples given so far illustrate how to translate type instance for classes to HOL, and how to generate HOL proof obligations for laws of type classes specified in Haskell.

Chapter 5

Conclusions and Future work

Type classes are a unique feature of Haskell programming language. Using classes, a Haskell programmer can overload functions over a set of types rather than just one. It provides Haskell programmers a way to define general programming interfaces. Type classes associate some invariants which should be satisfied when a type was defined to be a member of the types. To guarantee the correctness of type instances of a class, validation and verification must be done. In this thesis we discuss two methods to test and verify type instances, one is QuickCheck which is a testing tool for Haskell, and another is Isabelle which is a formal method tool used to prove theorems.

5.1 Contributions

The first method adopted is QuickCheck, which is an automatic testing tool for Haskell programs. It was used to check the properties of Haskell functions. One of the attractions of this tool is that properties could be expressed by Haskell language itself. By giving some examples, it shows how to express type classes' invariants in QuickCheck, and provide library of QuickCheck tests for prelude class instances.

The second method used is Isabelle/HOL. In Isabelle/HOL properties could be expressed as theorems or lemmas. Isabelle/HOL can prove them in a formal way. To prove laws of type classes of Haskell in Isabelle/HOL, the functions and specifications of Haskell programs need to be translated to Isabelle. Syntax differences require translating from Haskell to Isabelle/HOL. This thesis suggests a set of translating rules to guide function translation, datatype translation and specification translation while preserving the semantics. Based on the translating rules a simple automatic translator is implemented to ease the Haskell programmers who want to verify their type instances in Isabelle/HOL. By giving some example, this thesis also shows how

the types and functions are translated; how the laws of type classes are presented in Isabelle/HOL, and how the theorems are proved. Appendix B includes the proving scripts for the type instances in the examples.

5.2 Conclusions

QuickCheck is suitable for testing type instances; new types could be easily defined as testable and writing properties is straightforward.

Type instances could be verified in Isabelle/HOL while the functions, types and specifications are translated to Isabelle/HOL. For the most part, this translation was purely syntactical and straightforward with regular expression matching. While the new datatype, functions and specifications are defined within the range that the translation rules support, automatic translating is applicable.

5.3 Future work

The work done in this thesis could be extended in two aspects.

Firstly, when checking the properties in QuickCheck, type information has to be given explicitly. The QuickCheck procedure could be improved by existential type of GADTs; it allows properties to be checked on a list of testable types automatically.

Secondly, the translator needs to be extended to cover more Haskell and Isabelle/HOL features, for example, translating functions to well-founded recursions which is generally recursive and supporting full pattern matchings. In the future some theories of built-in types of Haskell are required to make the translation easier.

Bibliography

[Bir98] Richard Bird. *Introduction to functional programming using Haskell.* Prentice Hall Europe, University of Oxford, 1998.

[BW99] Stefan Berghofer and Markus Wenzel. Inductive datatypes in HOL - lessons learned in formal-logic engineering. In *Theorem Proving in Higher Order Logics*, pages 19–36, 1999.

[CH00] Koen Claessen and John Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In *ICFP '00: Proceedings of the fifth ACM SIGPLAN International Conference on Functional Programming*, pages 268–279, New York, NY, USA, 2000. ACM Press.

[CH02] Koen Claessen and John Hughes. Testing Monadic Code with QuickCheck. *ACM SIGPLAN Notices*, 37(12), December 2002.

[CH06a] Koen Claessen and John Hughes. Smallcheck: another lightweight testing library in haskell, November 2006.

[CH06b] Koen Claessen and John Hughes. SmallCheck: another lightweight testing library in Haskell. http://www.cs.york.ac.uk/fp/darcs/smallcheck, Nov. 2006.

[CW85] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Comput. Surv.*, 17(4):471–523, 1985.

[DM82] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *POPL '82: Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 207–212, New York, NY, USA, 1982. ACM Press.

[Hal03] Thomas Hallgren. Haskell tools from the programatica project. In *Haskell '03: Proceedings of the 2003 ACM SIGPLAN workshop on Haskell*, pages 103–106, New York, NY, USA, 2003. ACM Press.

[Hin69] R. Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60, Dec 1969.

[HJW+92] Paul Hudak, Simon Peyton Jones, Philip Wadler, Brian Boutel, Jon Fairbairn, Joseph Fasel, María M. Guzmán, Kevin Hammond, John Hughes, Thomas Johnsson, Dick Kieburtz, Rishiyur Nikhil, Will Partain, and John Peterson. Report on the programming language Haskell: a non-strict, purely functional language version 1.2. *SIGPLAN Not.*, 27(5):1–164, 1992.

[HK05] William L. Harrison and Richard B. Kieburtz. The logic of demand in haskell. *J. Funct. Program.*, 15(6):837–891, 2005.

[HPF99] Paul Hudak, John Peterson, and Joseph Fasel. A Gentle Introduction To Haskell 98. http://www.haskell.org/tutorial/, 1999.

[JJ06] Patrik Jansson and Johan Jeuring. Testing properties of generic functions. Technical report, Institute of Information and Computing Sciences Utrecht University, 2006.

[Jon95] Mark P. Jones. Functional programming with overloading and higher-order polymorphism. In *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques — Tutorial Text*, pages 97–136, London, UK, 1995. Springer-Verlag.

[Jon97] Mark P. Jones. First-class polymorphism with type inference. In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 483–496, New York, NY, USA, 1997. ACM Press.

[Kie02] Richard B. Kieburtz. P-logic: property verification for Haskell programs. www.cse.ogi.edu/PacSoft/projects/programatica/plogic.pdf, February 2002.

[LHJ95] Sheng Liang, Paul Hudak, and Mark Jones. Monad transformers and modular interpreters. In *POPL '95: Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 333–343, New York, NY, USA, 1995. ACM Press.

[MML07] Till Mossakowski, Christian Maeder, and Klaus Lttich. *The Heterogeneous Tool Set*, volume 4424. Springer-Verlag Heidelberg, 2007.

[MTM97] Robin Milner, Mads Tofte, and David Macqueen. *The Definition of Standard ML*. MIT Press, Cambridge, MA, USA, 1997.

[Nay06] Matthew Naylor. SparseCheck a logic programming library for test-data generation, 2006.

[NP93] Tobias Nipkow and Christian Prehofer. Type checking type classes. In *POPL '93: Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 409–418, New York, NY, USA, 1993. ACM Press.

[NPW02] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[NPW05] T. Nipkow, L. Paulson, and M. Wenzel. Isabelle's logics: HOL. http://isabelle.in.tum.de/doc/logics-HOL.pdf, October 2005.

[Pau89] Lawrence C. Paulson. The foundation of a generic theorem prover. *Journal of Automated Reasoning*, 5(3):363–397, 1989.

[Pau90a] L. C. Paulson. A formulation of the simple theory of types (for isabelle). In *COLOG-88: Proceedings of the international conference on Computer logic*, pages 246–274, New York, NY, USA, 1990. Springer-Verlag New York, Inc.

[Pau90b] Lawrence C. Paulson. Isabelle: The next 700 theorem provers. *Logic and Computer Science*, pages 361–385, 1990.

[PJ+03] Simon Peyton Jones et al. *The Revised Haskell 98 Report.* Cambridge University Press, 2003. Also on http://haskell.org/.

[Tho99] Simon Thompson. *Haskell: The Craft of Functional Programming, Second Edition.* Addison-Wesley Longman, 1999.

[Tur90] David Turner. *An overview of Miranda 1.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1990.

[Utt94] Mark Utting. A Haskell implementation of Z data types. http://citeseer.ist.psu.edu/utting94haskell.html, 1994.

[Wad92] Philip Wadler. The essence of functional programming. In *Conference Record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–14, Albequerque, New Mexico, 1992.

Appendix A

A simple translator from Haskell to Isabelle

entrf is the main entry of the program. It reads source code from a file and parses it using the Haskell parser.

```
entrf :: FilePath → IO ()
entrf fp ty = do x ← readFile fp
  writeFile ty
  ("imports thy \n\n" ++ (prettyPrint $ trimok $
  parseModule (unlines (docu (lines x))))
     ++ "\n\n" ++ (entr2 x))


entr1 :: FilePath → IO ()
entr1 fp = do x ← readFile fp
  putStrLn $
  "\n\\begin{isabelle}" ++ "\n\n"
   ++ unlines (fdecl (extdecl $ trimok
   $ parseModule (unlines (docu (lines x))))
  []) ++ "\\end{isabelle}"
entr2 :: String → String
entr2 x = "\\begin{isabelle}" ++ "\n\n"
   ++ unlines (scnimpt ta) ++ "\n"
   ++ unlines (fdecl ta ta)
   ++ "\\end{isabelle}"
  where ta = (extdecl ∘ trimok ∘
    parseModule ∘ unlines ∘ docu ∘ lines) x
```

Function trimok analyzes the Haskell module. After calling this function we already have a HsModule data structure.

```
trimok :: ParseResult HsModule → HsModule
trimok modu = case modu of
  (ParseOk x) → x
  (ParseFailed loc str) → error "r"
```

Function extdecl extracts the declaration part of from the Haskell program.

```
extdecl :: HsModule → [HsDecl]
extdecl (HsModule _ _ _ _ xs) = xs
```

Function scnimpt takes first round of scan to find what theories need to be imported. The information could be obtained from contexts of definitions of classes, instances, or functions.

```
scnimpt :: [HsDecl] → [String]
scnimpt [] = []
scnimpt (n : ns) = case n of
  (HsTypeDecl sl x xs ht) →
    scnimpt ns
  (HsDataDecl sl hc hn xs ys zs) →
    rmdup (cxt hc) (scnimpt ns)
  (HsInfixDecl sl ha i xs) →
    scnimpt ns
  (HsNewTypeDecl sl hc hn xs hd ys) →
    rmdup (cxt hc) (scnimpt ns)
  (HsClassDecl sl hc x xs ys) →
    rmdup (cxt hc) (scnimpt ns)
  (HsInstDecl sl hc hq xs ys) →
    rmdup (cxt hc) (scnimpt ns)
  (HsDefaultDecl sl xs) →
    scnimpt ns
  (HsTypeSig sl xs hq) →
    let (HsQualType hc ht) = hq
      in (rmdup (cxt hc) (scnimpt ns))
  (HsFunBind xs) →
    scnimpt ns
  (HsPatBind sl hp hsr xs) →
    scnimpt ns
```

In the first round scan of the Haskell source code, function cxt checks the context of **data**, **newtype**, **class**, **instance**, and type signature clauses to find what theories of Isabelle will be used in the current theory and generates the imports lines for it. You should be aware that in this way, you need to avoid importing the same theory more than once, for example if (Eq a) ⇒ appears in two different type signatures. Another thing you need to avoid is to import the current theory. This could happen because your defined classes were referenced as context by some other object in the current module. We keep a list of theory names that we need to import and generate

new by cxt if the one of more results of cxt are included in the list then do nothing otherwise add them into the list.

```
cxt :: HsContext → [String]
cxt [] = []
cxt (x : xs) = ("imports " ++ rmrt (unlines y)
   ++ ".thy") : cxt xs
  where y = let (hq, zs) = x
    in dataqnf hq
rmdup :: [String] → [String] → [String]
rmdup [] xs = xs
rmdup (x : xs) ys = if (elem x ys) then rmdup xs ys
  else x : rmdup xs ys
```

rmrt removes the all returns in a string.

```
rmrt :: String → String
rmrt [] = []
rmrt (x : xs) = if x ≡ '\n' then rmrt xs
  else x : rmrt xs
```

Function fdecl handles the declarations.

```
fdecl :: [HsDecl] → [HsDecl] → [String]
fdecl [] x = [""]
fdecl (n : ns) env = case n of
  (HsTypeDecl sl x xs ht) →
    rmrt (unlines (typedecl x xs ht)) : fdecl ns env
  (HsDataDecl sl hc hn xs ys zs) →
    (datadecl hn xs ys zs) ++ fdecl ns env
  (HsInfixDecl sl ha i xs) →
    fdecl ns env
  (HsNewTypeDecl sl hc hn xs hd ys) →
    fdecl ns env
  (HsClassDecl sl hc z xs ys) →
    classdecl sl hc z xs ys env ++ fdecl ns env
  (HsInstDecl sl hc hq xs ys) →
    instdecl hc hq xs ys env ++ fdecl ns env
  (HsDefaultDecl sl xs) → error "HsDeaultDecl"
  (HsTypeSig sl xs hq) →
    (tpsigdecl xs hq) : fdecl ns env
```

```
  (HsFunBind xs) →
    (funcdecl xs) ++ ["\n"] ++ fdecl ns env
  (HsPatBind sl hp hsr xs) → error "HsPatBind"
```

Function datadecl handles the datatype declarations of the Haskell module to generate the data declarations in Isabelle.

```
datadecl :: HsName → [HsName] → [HsConDecl]
  → [HsQName] → [String]
datadecl hn xs ys zs =
  lines (rmrt (unlines (lines (rmrt (bfeq ++
    "=" ++ "  ")) ++ cstrf ys))) ++ ["\n"]
  where xnf :: [HsName] → String
    xnf [] = []
    xnf (f : fs) = let (HsIdent n2) = f
      in "'" ++ n2 ++ "  " ++ xnf fs
    cstrf :: [HsConDecl] → [String]
    cstrf [] = []
    cstrf (h : hs) = if (length hs ≡ 0) then
      lines (rmrt (gcondecl h))
      else (rmrt ((gcondecl h) ++ "  \n"
       ++ "| "
       ++ "  ")) : cstrf hs
    bfeq = "datatype  " ++ xnf xs
      ++ "  " ++ datanf hn ++ "  "
    blkspc n = if (n > 0) then "  " ++ blkspc (n − 1)
      else ""
```

Function gcondecl generates the constructor parts of a datatype in Isabelle relating to the original Haskell data declaration.

datanf changes the HsName to a String by removing the type constructor of HsIdent.

```
gcondecl :: HsConDecl → String
gcondecl hcd = case hcd of
  (HsConDecl srl hsn hbs) → datanf hsn ++ (hbtype hbs)
  (HsRecDecl srl hsn [(hsn2, hbt)]) → datanf hsn
datanf :: HsName → String
datanf (HsIdent n1) = n1
```

dataqnf changes the qualified name to a list of strings.

```
dataqnf :: HsQName → [String]
dataqnf (Qual m n) = lines (datanf n)
dataqnf (UnQual n) = lines (datanf n)
dataqnf (Special h) = case h of
  HsUnitCon → ["()"]
  HsListCon → ["List"]
  HsFunCon → ["->"]
  HsTupleCon n → ["n"]
  HsCons    → [":"]
hbtype :: [HsBangType] → String
hbtype [] = ""
hbtype (x : xs) = "  " ++ typ ++ hbtype xs
  where typ = case x of
    (HsBangedTy ht) → hstype ht
    (HsUnBangedTy ht) → hstype ht
```

hstype changes the type of the form HsType to a string.

```
hstype :: HsType → String
hstype (HsTyFun x y) = (hstype x) ++
  " => " ++ hstype y
hstype (HsTyTuple (x : xs)) =
  "(" ++ hstype x ++ (tts xs)
  where tts :: [HsType] → String
    tts [] = ")"
    tts (z : zs) = "," ++ hstype z ++ (tts zs)
hstype (HsTyApp x y) = "" " ++ hstype y          ++ "
" ++ hstype x ++ " ""
hstype (HsTyVar x) = "'" ++ datanf x
hstype (HsTyCon x) = unlines (dataqnf x)
```

Function typedecl translates the type synonyms of Haskell program to Isabelle type synonyms.

```
typedecl :: HsName → [HsName]
   → HsType → [String]
typedecl x xs t = ["types   "]
   ++ lines (datanf x)
   ++ [" = "] ++ lines (hstype t)
   ++ ["\n"]
```

Function funcdecl extracts the function bindings from source code, Its output is a list of strings with a head of function name.

Most function definition will appear in an Isabelle theory. But the functions that define the specification will be transfered to proof obligations such as lemma or theorem so their original version will be hiden. I define such specification with a name prefixed by "check"; we use the function hidcheck to pick them out.

```
funcdecl :: [HsMatch] → [String]
funcdecl [] = []
funcdecl (x : xs) =
  if (hidcheck x "check") then funcdecl xs
  else ("primrec\n" ++
    (prettyPrint x)) : funcdecl xs
hidcheck :: HsMatch → String → Bool
hidcheck (HsMatch sl hn xs y zs) pt =
if (pt ≡ (take (length pt) (datanf hn))) then True
else False
```

Function tpsigdecl (typesig) was used to construct the general type signature in a Haskell program.

```
tpsigdecl :: [HsName] → HsQualType → String
tpsigdecl [] qtp = []
tpsigdecl (x : xs) qtp = if (take 5 (datanf x) ≡ "check")
  then tpsigdecl xs qtp
  else ("consts\n   " ++ (datanf x) ++ "  ::  \""
    ++ (tpsigdecl2 qtp) ++ "\"") ++ tpsigdecl xs qtp
tpsigdecl2 :: HsQualType → String
tpsigdecl2 (HsQualType hc ht) = hstype ht
```

Function instdecl finds out all instance declarations and translates them into Isabelle syntax. As output, three things will be generated: The first is an Isabelle consts declaration; the second is a function definition; the third is a list of lemmas which were derived from specification and type signature.

```
instdecl :: HsContext → HsQName → [HsType] →
[HsDecl] → [HsDecl] → [String]
instdecl hc hq xs ys env = ["consts"]
  ++ consts hq xs ys env
  ++ ["primrec"] ++ primrec hq xs ys env ++ ["\n"]
  ++ lemma hq xs ys env ++ ["\n"]
```

Function const helps to generate the const declaration of Isabelle from it's counterpart in the Haskell module. hq is the class name xs is the types that will be instanced. ys is the declarations. cls contains a class declaration.

tdecl picks all class declarations from the Haskell module.

Functions cdecl and bdecl together select the specific class declaration corresponding to the current instance from the return of tdecl

```
consts :: HsQName → [HsType] →
[HsDecl] → [HsDecl] → [String]
consts hq xs ys env = iconsts xs cls ++ ["\n"]
  where cls = cdecl hq (tdecl env)
iconsts :: [HsType] → HsDecl → [String]
iconsts [] _ = []
iconsts (x : xs) y =
    repclassdef (hstype x) (takclstype y) y
takclstype :: HsDecl → String
takclstype (HsClassDecl sl hc z xs ys) = tnm xs
  where tnm :: [HsName] → String
    tnm [] = ""
    tnm (y : ys) = datanf y
takclstype _ = ""
repclassdef :: String → String → HsDecl → [String]
repclassdef x y (HsClassDecl sl hc z xs ys) =
  repdecl x y ys
repclassdef _ _ _ = []
repdecl _ _ [] = []
repdecl t1 t2 (x : xs) =
    (repsig t1 t2 x) : (repdecl t1 t2 xs)
repsig :: String → String → HsDecl → String
repsig t1 t2 (HsTypeSig _ xs y) =
  rmrt ("    " ++ (datanf (head xs))
   ++ "_" ++ t1
   ++ "  ::  \""
   ++ pptypesig t1 t2 (etypesig y)
   ++ "\"")
repsig _ _ _ = ""
etypesig (HsQualType x y) = y
pptypesig :: String → String → HsType → String
```

```
pptypesig t1 t2 (HsTyFun x y) = (ptypesig t1 t2 x)
  ++ " => " ++ (ptypesig t1 t2 y)
pptypesig t1 t2 x = ptypesig t1 t2 x

ptypesig :: String → String → HsType → String
ptypesig t1 t2 (HsTyFun x y) = "(" ++ (ptypesig t1 t2 x)
  ++ " => " ++ (ptypesig t1 t2 y) ++ ")"
ptypesig t1 t2 (HsTyTuple (x : xs)) =
  "(" ++ ptypesig t1 t2 x ++ (tts xs)
  where tts :: [HsType] → String
    tts [] = ")"
    tts (z : zs) = "," ++ ptypesig t1 t2 z ++ (tts zs)
ptypesig t1 t2 (HsTyApp x y) = (ptypesig t1 t2 y)
  ++ " " ++ (ptypesig t1 t2 x)
ptypesig t1 t2 (HsTyVar x) = if (t2 ≡ datanf x)
  then t1
  else rmrt ("'" ++ datanf x)
ptypesig t1 t2 (HsTyCon x) = unlines (dataqnf x)

cdecl :: HsQName → [HsDecl] → HsDecl
cdecl _ [] = error "No proper class"
cdecl hq (x : xs) = if (bdecl hq x) then x
  else cdecl hq xs

bdecl :: HsQName → HsDecl → Bool
bdecl hq x = if ((head (dataqnf hq)) ≡ (cname x))
  then True
  else False

cname (HsClassDecl _ _ n _ _) = datanf n

tdecl :: [HsDecl] → [HsDecl]
tdecl [] = []
tdecl (x : xs) = case x of
  (HsClassDecl _ _ _ _ _) → x : tdecl xs
  _ → tdecl xs
```

Function primrec generates the primrec section for an isabelle theory. Functions following it help to combine a new consts function name which show it is an instance of the type.

```
primrec :: HsQName → [HsType] →
[HsDecl] → [HsDecl] → [String]
primrec hq [] ys env = []
primrec hq (x : xs) ys env = sbtprc ys (hstype x)
```

```
    ++ primrec hq xs ys env
sbtprc :: [HsDecl] → String → [String]
sbtprc [] _ = []
sbtprc (x : xs) pt = (sbtdecl x pt) ++ sbtprc xs pt
sbtdecl :: HsDecl → String → [String]
sbtdecl hd [] = [prettyPrint hd]
sbtdecl (HsFunBind xs) y = sbtfb xs y
sbtdecl _ y = []
sbtfb :: [HsMatch] → String → [String]
sbtfb [] _ = []
sbtfb (x : xs) [] = (prettyPrint x) : sbtfb xs []
sbtfb (x : xs) hd = ("    " ++ (sbtmtch x hd)) : sbtfb xs hd
```

sbtmtch substitutes the polymorphic function with a monomorphic function within a primrec definition.

Function lemma generates a lemma in Isabelle from the source code and specification in Haskell.

```
lemma :: HsQName → [HsType] →
[HsDecl] → [HsDecl] → [String]
lemma hq xs ys env =
lemmas nn hq xs ys (lespec hq env)
  where nn = "check" ++ head (dataqnf hq)
lemmas :: String → HsQName → [HsType]
   → [HsDecl] → [HsDecl] → [String]
lemmas nn hq [] _ env = []
lemmas nn hq _ [] env = []
lemmas nn hq _ _ [] = []
lemmas nn hq xs (z : zs) ys =
  lemmass nn hq (head xs) z ys ++ lemmas nn hq xs zs ys
lemmass :: String → HsQName → HsType
   → HsDecl → [HsDecl] → [String]
lemmass nn hq _ _ [] = []
lemmass nn hq x z (y : ys) =
  rmrt ("lemma " ++ ispec nn y
   ++ "_" ++ hstype x ++ " : "
   ++ "\"" ++ stblemma y (funame z) (hstype x) ++ "\"")
   : lemmass nn hq x z ys
stblemma :: HsDecl → String → String → String
```

```
stblemma decl t1 t2 = stbrhs (stbmt (stbfb decl)) t1 t2
stbfb (HsFunBind (x : xs)) = x
stbmt (HsMatch _ _ _ x _) = x
stbrhs :: HsRhs → String → String → String
stbrhs (HsUnGuardedRhs e) t1 t2 = stbexp e t1 t2
stbrhs (HsGuardedRhss (e : es)) t1 t2 = ""
stbexp :: HsExp → String → String → String
stbexp (HsVar x) t1 t2 = if (head (dataqnf x) ≡ t1)
  then rmrt (head (dataqnf x) ++ "_" ++ t2)
  else head (dataqnf x)
stbexp (HsCon x) t1 t2 = if (head (dataqnf x) ≡ t1)
  then rmrt (head (dataqnf x) ++ "_" ++ t2)
  else head (dataqnf x)
stbexp (HsLit x) t1 t2 = prettyPrint x
stbexp (HsInfixApp x y z) t1 t2 =
  rmrt ((stbexp x t1 t2)
  ++ " " ++ prettyPrint y
  ++ " " ++ (stbexp z t1 t2))
stbexp (HsApp x y) t1 t2 = (stbexp x t1 t2) ++ " "
  ++ (stbexp y t1 t2)
stbexp (HsNegApp x) t1 t2 = "!" ++ stbexp x t1 t2
stbexp (HsLambda x ys z) t1 t2 = ""
stbexp (HsLet xs y) t1 t2 = ""
stbexp (HsIf x y z) t1 t2 =
  "if (" ++ stbexp x t1 t2 ++ ")"
  ++ " then " ++ stbexp y t1 t2
  ++ " else " ++ stbexp z t1 t2
stbexp (HsCase x ys) t1 t2 = ""
stbexp (HsDo xs) t1 t2 = ""
stbexp (HsTuple xs) t1 t2 = "(" ++ tupexp xs
  where tupexp :: [HsExp] → String
        tupexp [] = []
        tupexp (y : []) = stbexp y t1 t2 ++ ")"
        tupexp (y : ys) = stbexp y t1 t2 ++ "," ++ tupexp ys
stbexp (HsList xs) t1 t2 = "[" ++ lstexp xs
  where lstexp :: [HsExp] → String
        lstexp [] = []
        lstexp (y : []) = stbexp y t1 t2 ++ "]"
        lstexp (y : ys) = stbexp y t1 t2 ++ "," ++ lstexp ys
```

```
stbexp (HsParen x) t1 t2 = "(" ++ stbexp x t1 t2 ++ ")"
stbexp (HsLeftSection x y) t1 t2 = "( "
  ++ stbexp x t1 t2 ++ "  "
  ++ prettyPrint y ++ ")"
stbexp (HsRightSection x y) t1 t2 = "( "
  ++ prettyPrint x ++ "  " ++ stbexp y t1 t2 ++ ")"
stbexp (HsRecConstr x ys) t1 t2 = ""
stbexp (HsRecUpdate x ys) t1 t2 = ""
stbexp (HsEnumFrom x) t1 t2 = "[" ++ stbexp x t1 t2 ++ "..]"
stbexp (HsEnumFromTo s y) t1 t2 = "[" ++ stbexp s t1 t2
  ++ ".." ++ stbexp y t1 t2 ++ "]"
stbexp (HsEnumFromThen x y) t1 t2 = ""
stbexp (HsEnumFromThenTo x y z) t1 t2 = ""
stbexp (HsListComp x ys) t1 t2 = ""
stbexp (HsExpTypeSig x y z) t1 t2 = ""
stbexp (HsAsPat x y) t1 t2 = ""
stbexp HsWildCard t1 t2 = "  _  "
stbexp (HslrrPat x) t1 t2 = ""
funame :: HsDecl → String
funame (HsFunBind (x : xs)) =
  let (HsMatch _ ss _ _ _) = x
  in datanf ss
funame _ = []
```

To build lemma function, lespec takes class name and the whole AST of programs to pick out the specifications (rules) of a type class by matching the names.

```
lespec _ [] = []
lespec hq (x : xs) = if (kk ≢ "")
  then x : lespec hq xs
  else lespec hq xs
  where nn = "check" ++ head (dataqnf hq)
    kk = ispec nn x
```

ispec test every object to find whether it is a function bind; if so, it returns the function name; if not, it returns the empty string.

```
ispec :: String → HsDecl → String
ispec y (HsFunBind (x : xs)) =
  let (HsMatch sl hn zs hrhs hs) = x
  in if (y ≡ take (length y) (datanf hn))
```

```
        then datanf hn
        else ""
ispec y (HsFunBind []) = ""
ispec y _ = ""


classdecl :: SrcLoc → HsContext → HsName → [HsName] →
[HsDecl] → [HsDecl] → [String]
classdecl sl hc hn xs ys env = []
```

Appendix B

Isabelle Proofs for Selected Standard Instances

theory *aclass*
imports *Main Fun List*

begin
datatype *'a maybe = Nothing | Just 'a*

consts
fmap-maybe :: ('a => 'b) => ('a maybe => 'b maybe)

primrec
fmap-maybe f Nothing = Nothing
fmap-maybe f (Just x) = Just (f x)

lemma *checkFunctor-id-maybe: fmap-maybe id fa = fa*
apply(*induct-tac fa*)
apply(*auto*)
done

constdefs
nn :: nat => nat
nn x == let y = x+x; z = x+x in y + z

lemma *checkFunctor-comp-maybe: fmap-maybe g (fmap-maybe f fa) = fmap-maybe (g ∘ f) fa*
apply(*induct-tac fa*)
apply(*auto*)
done

consts
bind-maybe :: 'a maybe => ('a => 'b maybe) => 'b maybe

primrec

bind-maybe (Just x) f = f x
bind-maybe Nothing f = Nothing

constdefs *return-maybe :: 'a => 'a maybe*
return-maybe x == Just x

lemma *checkMonad-lUnit-maybe[simp]: bind-maybe (return-maybe x) f = f x*
apply(*simp only: return-maybe-def*)
apply(*auto*)
done

lemma *checkMonad-runit-maybe: bind-maybe f return-maybe = f*
apply(*induct-tac f*)
apply(*auto*)
apply(*simp only: return-maybe-def*)
done

lemma *checkMonad-comp-maybe: bind-maybe f (%x. bind-maybe (g x) h) = bind-maybe (bind-maybe f g) h*
apply(*induct-tac f*)
apply(*auto*)
done

constdefs *zero-maybe :: 'a maybe*
zero-maybe == Nothing

consts *plus-maybe :: 'a maybe => 'a maybe => 'a maybe*

primrec
plus-maybe Nothing x = x
plus-maybe (Just x) y = Just x

lemma *checkMonad-lzero-maybe: bind-maybe m (%x . zero-maybe) = zero-maybe*
apply(*induct-tac m*)
apply(*auto*)
apply(*simp only: zero-maybe-def*)
done

lemma *checkMonad-rzero-maybe: bind-maybe zero-maybe m = zero-maybe*

apply(*simp only*: *zero-maybe-def*)
apply(*auto*)
done

lemma *checkMonad-plusr-maybe* : *plus-maybe m zero-maybe* = *m*
apply(*induct-tac m*)
apply(*auto*)
apply(*simp only*: *zero-maybe-def*)
done

lemma *checkMonad-plusl-maybe*: *plus-maybe zero-maybe m* = *m*
apply(*induct-tac m*)
apply(*simp only*: *zero-maybe-def*)
apply(*auto*)
apply(*simp only*: *zero-maybe-def*)
apply(*auto*)
done

consts
bind-list :: *'a list* => (*'a* => *'b list*) => *'b list*

primrec
bind-list [] *f* = []
bind-list (*x* # *xs*) *f* = (*f x*) @ (*bind-list xs f*)

lemma *bindConcatlist*[*simp*]: !*ys* . *bind-list* (*xs* @ *ys*) *f* = *bind-list xs f* @ *bind-list ys f*
apply(*induct-tac xs*)
apply(*auto*)
done

constdefs *return-list* :: *'a* => *'a list*
return-list x == *x* # []

constdefs *zero-list* :: *'a list*
zero-list == []

consts *monplus* :: *'a list* => *'a list* => *'a list*

primrec
monplus [] *x* = *x*

monplus $(x\#xs)\ y = \quad x\ \#\ (xs\ @\ y)$

lemma *checkMonad-lunit-list*[*simp*]: *bind-list* (*return-list x*)` *f* = *f x*
apply(*simp only*: *return-list-def*)
apply(*auto*)
done

lemma *checkMonad-runit-list*: *bind-list f return-list* = *f*
apply(*induct-tac f*)
apply(*auto*)
apply(*simp only*: *return-list-def*)
done

lemma *checkMonad-comp-list*: *bind-list* (*bind-list f g*) *h* = *bind-list f* (%*x*. *bind-list* (*g x*) *h*)
apply(*induct-tac f*)
apply(*simp*)
apply(*auto*)
done

lemma *checkMonad-rzero-list*: *bind-list m* (%*x* . *zero-list*) = *zero-list*
apply(*induct-tac m*)
apply(*auto*)
apply(*simp only*: *zero-list-def*)
apply(*simp only*: *zero-list-def*)
done

lemma *checkMonad-lzero-list*: *bind-list zero-list m* = *zero-list*
apply(*simp only*: *zero-list-def*)
apply(*auto*)
done

lemma *checkMonad-plusr-list* : *monplus m zero-list* = *m*
apply(*induct-tac m*)
apply(*auto*)
apply(*simp only*: *zero-list-def*)
apply(*simp only*: *zero-list-def*)
done

lemma *checkMonad-plusl-list*: *monplus zero-list m* = *m*
apply(*induct-tac m*)

apply(*simp only*: *zero-list-def*)
apply(*auto*)
apply(*simp only*: *zero-list-def*)
apply(*auto*)
done

datatype ('*a*,'*b*) *either* = *Left* '*a* | *Right* '*b*

consts
fmap-either :: ('*a* => '*b*) => (('*c*,'*a*) *either* => ('*c*,'*b*) *either*)

primrec
fmap-either f (*Left x*) = *Left x*
fmap-either f (*Right y*) = *Right* (*f y*)

lemma *checkFunctor-id-either*: *fmap-either id fa* = *fa*
apply(*induct-tac fa*)
apply(*auto*)
done

lemma *checkFunctor-comp-either* : *fmap-either g* (*fmap-either f fa*) = *fmap-either* (*g* ∘ *f*) *fa*
apply(*induct-tac fa*)
apply(*auto*)
done

consts
bind-either :: ('*c*,'*a*) *either* => ('*a* => ('*c*,'*b*) *either*) => ('*c*,'*b*) *either*

primrec
bind-either (*Left x*) *f* = *Left x*

bind-either (*Right y*) *f* = *f y*

constdefs *return-either* :: '*a* => ('*b*,'*a*) *either*
return-either x == *Right x*

lemma *checkMonad-lunit-either*[*simp*]: *bind-either* (*return-either x*) *f* = *f x*
apply(*simp only*: *return-either-def*)
apply(*auto*)
done

lemma *checkMonad-runit-either*: *bind-either f return-either = f*
apply(*induct-tac f*)
apply(*auto*)
apply(*simp only*: *return-either-def*)
done

lemma *checkMonadcomp-either*: *bind-either f (%x. bind-either (g x) h) = bind-either (bind-either f g) h*
apply(*induct-tac f*)
apply(*auto*)
done

types *('a,'s) State = 's => 'a × 's*

constdefs
fmap-state :: ('a => 'b) => (('a,'c) State => ('b,'c) State)
fmap-state f st == (%s. (let (x,s') = st s in (f x, s')))

constdefs *return-state :: 'a => ('a,'s) State*
return-state x == (%y. (x,y))

constdefs
bind-state :: ('a,'s) State => ('a => ('b,'s) State) => ('b,'s) State
bind-state st f == (%x. let (a,st') = st x in (let m' = f a in m' st'))

lemma *checkFunctor-id-state*: *fmap-state id fa = fa*
apply(*simp only*: *fmap-state-def*)
apply(*simp only*: *Let-def*)
apply(*simp add*: *split-def*)
done

lemma *checkFunctor-comp-state*: *fmap-state g (fmap-state f fa) = fmap-state (g ∘ f) fa*

apply(*simp only*: *fmap-state-def*)
apply(*auto*)
apply(*simp add*: *Let-def*)
apply(*simp only*: *split-def*)
apply(*auto*)
done

lemma *checkMonad-lunit-state*[*simp*]: *bind-state* (*return-state x*) *f* = *f x*
apply(*simp only*: *return-state-def*)
apply(*simp only*: *bind-state-def*)
apply(*auto*)
apply(*simp add*: *Let-def*)
done

lemma *checkMonad-runit-state*: *bind-state f return-state* = *f*
apply(*simp only*: *bind-state-def*)
apply(*simp only*: *return-state-def*)
apply(*simp only*: *Let-def*)
apply(*simp only*: *split-def*)
apply(*auto*)
done

lemma *checkMonad-comp-state*: *bind-state f* (%*x*. *bind-state* (*g x*) *h*) = *bind-state* (*bind-state f g*) *h*
apply(*simp only*: *bind-state-def*)
apply(*simp only*: *Let-def*)
apply(*simp only*: *split-def*)
done

types ('*a*,'*s*) *StateT* = '*s* => ('*s* ×'*a*) *maybe*

constdefs
returnT :: '*a* => ('*a*,'*s*) *StateT*
returnT x == (%*k*. *return-maybe* (*k*,*x*))

constdefs
bindT :: ('*a*,'*s*) *StateT* => ('*a* => ('*b*,'*s*) *StateT*) => ('*b*,'*s*) *StateT*
bindT m f == %*p*. (*bind-maybe* (*m p*) (%(*q*,*a*). ((*f a*) *q*)))

constdefs
lift :: '*a maybe* => ('*a*,'*s*) *StateT*

lift m == %p. (bind-maybe m (%q. return-maybe (p,q)))

lemma *Monadtrans-liftunit*: *(lift ∘ return-maybe) x = returnT x*
apply(*simp add*: *comp-def*)
apply(*simp add*: *return-maybe-def*)
apply(*simp add*: *lift-def*)
apply(*simp add*: *return-maybe-def*)
apply(*simp add*: *returnT-def*)
apply(*simp add*: *return-maybe-def*)
done

lemma *Monadtrans-liftbind*: *lift (bind-maybe m k) = bindT (lift m) (lift ∘ k)*
apply(*simp add*: *lift-def*)
apply(*simp add*: *return-maybe-def*)
apply(*simp add*: *bindT-def*)
apply(*simp add*: *lift-def*)
apply(*simp add*: *return-maybe-def*)
apply(*induct-tac m*)
apply(*auto*)
done

end