Deriving Pretty-printing for Haskell

Yi Zhen

Master of Science
School of Informatics
University of Edinburgh

2016

Abstract

Print information of data type values can be used to help programmers understand the nature and the way the structure of instances of a certain data type is generated. This work aims to provide an interface wrapper which includes a pre-designed indentation format for printing arbitrary data types, called a pretty printer. It can be seen as an extension of the Show library which converts the value of the data type to a printable string in Haskell. This report describes the design, implementation and evaluation of such a pretty-printing library. The overall result is a pretty printer intended to be available, easy-to-use, and interesting for programmers to print the data type value in a visually appealing way.

Acknowledgements

Many thanks to my supervisor Prof. Philip Wadler for the guidance and numerous explanations; and to my parents, for food and a place to live.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Yi Zhen*)

Table of Contents

List of Figures

Listings

Chapter 1

Introduction

1.1 Introduction and Purpose

This project set out to create a generic, deriving Haskell pretty printer that is an interface wrapper between a collection of pretty printer combinators and user-defined data types, where generic means that ”it is a form of abstraction that allows defining functions that can operate on a large class of datatypes” (Leather, 2012) and ”the deriving mechanism supports automatic generation of instances for a number of functions” (Magalhes, 2010). Such a printer provides a way to print out the value of data type in a consistent format. It is in a form of Haskell library based on the interfaces in the paper, ’A Prettier Printer’ by Wadler (2003). The form of generic feature used is based on that introduced in the paper, ’A Generic Deriving Mechanism for Haskell’ by (Magalhes, 2010).

The functionality of this pretty printer can be seen as a reasonable extension of the ’Show’ library which converts the data type value to a printable string. It looks intuitive and more appealing to print the output of the value of string data type onto the screen under typesetting rather than printing the result directly. The pretty printer provides an interface wrapper which includes a pre-designed indentation format for arbitrary data types. This make it easier to read and clearer to convey the structure of the data type. Reading a paper which is written with LaTex would be much more comfortable than reading in plain-text.

The following code is a demonstration that compares the output of a print function to the pretty printer (pprint), where the print function converts values to strings for output using the Show operation and adds a newline. They both print one tree data type, but generate noticeably different results.

```
Prelude Text.PPrinter> print tree
Node ”aaa” [Node ”bbbbb” [Node ”ccc” [],Node ”dd” []],Node ”eee” [],
Node ”ffff” [Node ”gg” [],Node ”hhh” [],Node ”ii” []]]
Prelude Text.PPrinter> pprint tree
Node ”aaa”
     [Node ”bbbbb”
           [Node ”ccc” [],
            Node ”dd” []],
      Node ”eee” [],
      Node ”ffff”
           [Node ”gg” [],
            Node ”hhh” [],
            Node ”ii” []]]
```

Listing 1.1: Pretty printing example

According to the demonstration, the tree data type which is printed by the pretty printer looks clearer. Users may find it is much easier to understand the data from the pretty printer than from the result of Show.

The goal of this project is to extend the Haskell library with a way to derive a function automatically to pretty print the value of a given data type.

Up to now, there have already been some pretty-printing libraries implemented for Haskell. None of them combine generic mechanism and Wadler's combinators together, whereas this current study does. In the remaining chapters and sections, the following things are introduced: (i) Instructions in using a pretty printer. (ii) Related works. (iii) Background of the preliminary knowledge. (iv) Design of the pretty printer. (v) Implementation. (vi) Evaluation. Finally, the conclusion is included at the end of the paper.

1.2 How to Use Pretty-printing in Haskell

This library is user-friendly. The Haskell programming language can derive implementations of certain common tasks because of a 'deriving' facility. For example, to show a data type declaration which converts the value of that data type to a string could be done by adding '(deriving Show)' at end of the declaration. It automatically derives a function instead of making users implement it themselves. With the deriving mechanism facility of type class in Haskell, it is possible for the programmer to specify how to derive different functions.

Users should use the pragma '-DeriveGeneric' and import the library first, and then define the data type and derive Generic and Show type classes. Then users only need to declare the instance rather than define it. Finally, the value of data type should be defined for printing.

```
{-# LANGUAGE DeriveGeneric #-}
import Text.PPrinter

data Tree = Node String [Tree] deriving (Generic, Show)
instance Pretty (Tree)
tree = Node "aaa" [
             [Node "bbbbb"
                   [Node "ccc" [],
                    Node "dd" []],
              Node "eee" [],
              Node "ffff"
                   [Node "gg" [],
                    Node "hhh" [],
                    Node "ii" []]]
```

Listing 1.2: Definition of tree data type

For printing the above value out, users can use the pprint function directly as mentioned before, or use pprintLen to customise the maximum length of width for each line. It accepts two arguments: the first is an integer of the length and the second is the value of the data type. The following example shows the effects of different length parameters. The longer the length, the more characters can be put on each line. Actually, the pprint function has a default length of width(40). For more customisation details, see Chapter 3.

```
Text.PPrinter> pprintLen 40 tree
Node "aaa"
     [Node "bbbbb"
```

```
           [Node "ccc" [],
            Node "dd" []],
      Node "eee" [],
      Node "ffff"
           [Node "gg" [],
            Node "hhh" [],
            Node "ii" []]]
Text.PPrinter> pprintLen 60 tree
Node "aaa"
     [Node "bbbbb" [Node "ccc" [], Node "dd" []],
      Node "eee" [],
      Node "ffff"
           [Node "gg" [], Node "hhh" [], Node "ii" []]]
Text.PPrinter> pprintLen 80 tree
Node "aaa"
     [Node "bbbbb" [Node "ccc" [], Node "dd" []],
      Node "eee" [],
      Node "ffff" [Node "gg" [], Node "hhh" [], Node "ii" []]]
```

Listing 1.3: Outputs of pretty printer

On the other hand, programmers do not need to implement anything of the instance because this is derived by the compiler automatically. The following code is a good example that demonstrates the benefit of a generic deriving mechanism in this study's pretty printer.

```
data Trees a = Leaf a | Nod (Trees a) (Trees a)
    deriving (Generic, Show)

instance (Pretty a) => Pretty (Trees a)
```

Listing 1.4: Comparison of definition of the instances - 1

Programmers can also implement the instance by themselves. However, they have to implement all the necessary methods if they do not use a deriving mechanism. This substantially increases the workload. In fact, the performance of the two instances is the same.

```
data Trees a = Leaf a | Nod (Trees a) (Trees a)
    deriving (Show)
```

```
instance (Pretty a) => Pretty (Trees a) where
        ppPrec d (Leaf m) = rep $ wrapParens (d > appPrec) $
            text ”Leaf␣” : [nest (constrLen + parenLen)
            (ppPrec (appPrec + 1) m)]
          where appPrec = 10
                constrLen = 5
                parenLen = if(d > appPrec) then 1 else 0

        ppPrec d (Nod u v) = rep $ wrapParens (d > appPrec) $
            text ”Nod␣” :
            nest (constrLen + parenLen) (ppPrec (appPrec + 1) u) :
            [nest (constrLen + parenLen) line <>
            (ppPrec (appPrec + 1) v)]
          where appPrec = 10
                constrLen = 4
                parenLen = if(d > appPrec) then 1 else 0

-- helper function for wrapping the parenthesis
wrapParens :: Bool      -- add parens or not
              -> [DOC]
              -> [DOC]
wrapParens _ [] = []
wrapParens False s = s
wrapParens True (x:xs) = lpar <> x : wrapParens2 xs
  where
    wrapParens2 = foldr (:) [rpar]
```

Listing 1.5: Comparison of definition of the instances - 2

Therefore, it is easier to use a generic mechanism than to implement the instances by hand.

1.3 Related Work

Ranca (2012) has implemented a Haskell library called GenericPretty based on another pretty printer combinators maintained by Terei (2001). This printer uses Hughes's model which is a different design to the one in this study. The latter has a simpler pretty-printing model designed by Wadler. His work is a good example that can be used as a reference. This MSc project builds on a different existing library of combinators than Razvan Ranca's for deriving pretty-printing for Haskell.

Chapter 2

Background

Print information of data type values provides a common way for programmers to know the nature and see how the structure of any instances of a certain data type is generated. The two main advantages of pretty printing such information are: (i) it allows the programmer to analyse and understand the running state of a program dynamically and clearly, (ii) it is produced by some specific interfaces which simplify the steps of using side-effect functions in a pure, functional programming language such as Haskell. The pretty printer is not only applied for functional programming language, but here we do not consider other sorts of language.

Haskell provides a sound type system so that type errors do not occur in a well-typed Haskell program. Thus, a generic pretty printer for Haskell could guarantee that it can work well for all the arbitrary data types under normal circumstances. Hutton (2007) also claims that such a type system is more powerful than most modern programming languages.

However, it is hard to prove the soundness of the pretty printer since the property of ’pretty’ is ambiguous. This notwithstanding, the pretty printer is qualified to provide a robust, user-friendly way of printing such information.

2.1 Haskell

2.1.1 Introduction

Hudak (2007) states that most research languages are usually employed less after a year of use, or five years if it is a successful and widely-used research language. However, Haskell, a purely functional programming language with static, strong typing and non-strict semantics, has been used for more than fifteen years since it was released in 1990 (Hudak, 2007).

Nowadays, programmers can develop software in a wide variety of application domains with the help of Haskell since it is general-purpose. It is being widely used both in academia and industry. According to Done (2014) computer scientist Dijkstra has suggested that universities teach students Haskell. In the last few years, a considerable number of computer science schools set Haskell programming as a compulsory course at a undergraduate level.

Haskell has also influenced many other programming languages that have been inspired by its advanced features of Haskell such as its type system. For example, Agda by Norell (2007), a dependently typed functional programming language, is an interactive theorem prover which is written in Haskell. It has the Haskell-like syntax and a very powerful type system. Another good example is Scala (2004) which has the functional programming features of purity and laziness (Binstock, 2011).

2.1.2 Data Type in Haskell

Everything in Haskell has a type. According to its static type system, the type of data is determined entirely when it has been compiled rather than at runtime (such as with Python). It is better to identify type errors before the program crashes. Besides this, Haskell also supports many other advanced features. This section does not introduce the theory of type but gives a quick, brief overview of data types in Haskell.

First, Haskell provides primitive types just like most of programming languages do. The programmer can declare data with the type: Float, Integer and Char, etc. This kind of data type is always a fundamental component of a type system.

Second, it is limited that the language only supports primitive types. Programmers should define the type themselves when they need to. Thus, algebraic data type may contribute to helping developers define a new type themselves.

```
data Bool  = False | True
data Ords  = EQ | LT | GT
```

Listing 2.1: Definition of data type

The type should be denoted in the left of '='. Then users can define the value constructors to the right of '='. The two examples above define Bool and Ords. Bool is a Boolean type with two nullary constructors, False and True. To follow the example of Bool, it costs three constructors to define an Ords type.

To define a record in Haskell, programmers may use selectors instead of constructors. It is a clearer way to do the same thing by record syntax in Haskell.

```
data Person = Person { firstName :: String  , lastName :: String }
```

Listing 2.2: Selectors

Third, Haskell also incorporates polymorphic types. These types are universally qualified in some ways over all types. For instance, (forall z) (z, z) is the family of types consisting of, for every type z, the type of pair 'z' (Hudak, 2000). The Identifier 'z' is a type variable. The following function head has the type of ([a] $-$ $>$ a). It works on any list and returns the first element of the list. This is known as parametric polymorphism whereby the type of a value contains one or more type variables.

```
head :: [a] -> a
head (x:xs)    =  x
-- head [1, 2, 3] = 3
-- head ['a', 'b', 'c'] = 'a'
```

Listing 2.3: Type variable

Finally, the user can define a parameterised type and recursive data structure. Polymorphism is a useful feature that saves the user time in defining a collection of types which are in the same mode. The user does not need to define one certain type with type variable(s) one by one.

```
data Tree a = Leaf a | Node (Tree a) (Tree a)
-- The definition of List in the generalized syntax:
data List a where
  Cons ::   a -> List a -> List a
  Nil  ::   List a
```

Listing 2.4: Data structures

Programmers may find it helpful to define a tree type once and then this type can be used in many ways with a specific type variable such as (Tree Int), the tree data structure of integers and (Tree Char), the tree data structure of characters. The (List a) is similar to (Tree a).

2.1.3 Type Classes and Polymorphism

In addition to purity and laziness, one of the most important language features which will be remembered even when Haskell is dust is type classes, which are the most unusual feature of Haskell's type system. The form of classes in Haskell is similar to those used in other object-oriented languages such as Python and C++, but class of types is totally different from class of objects since type is not an object.

For example, we declare a class of type called Ord. It provides some necessary methods so that Haskell can determine the order of comparable data types.

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

Listing 2.5: Type class example

Operators $(<), (<=), (>=), (>)$ can be applied to arguments of many different types. This is called operator overloading in object-oriented programming, better known as ad hoc polymorphism.

```
quicksort  ::  (Ord a) => [a] -> [a]
```

Listing 2.6: Quicksort

The above code which from 'A Gentle Introduction to Haskell' by Hudak (2000) shows one instance of the use of Ord, where the typing of a quick sort function is defined. Here the type 'a' must be an instance of the class Ord. (Ord a) is not a type expression but is called a context. It expresses a constraint on a type. This should be read as 'For every type a that is an instance of the class Ord, quicksort has type [a]− >[a]' (Hudak, 2000).

Finally, compared to defining families of types by universally quantifying over all types (parametric polymorphism which is mentioned), Haskell's type classes provide a way for overloading (ad hoc polymorphism). The relationship between them is that the parametric polymorphism can be considered as a kind of overloading as well, but the overloading of parametric polymorphism occurs absolutely over all types instead of a constrained set of types which is quantified through a structured method provided by type classes (Hudak, 2000).

2.2 A Prettier Printer

The interfaces of the collection of pretty printer combinators used are based on those introduced in the paper: 'A Prettier Printer', Wadler (2003). This is not the only collection of combinators in existence. Another example is Hughes's pretty printer combinators, (Hughes, 1995). Compared to Hughes's library, Wadler (2003) points out that "the new library is based on a single way to concatenate documents, which is associative and has a left and right unit." He goes on to say that "Hughes's library has two distinct ways to concatenate documents, horizontal and vertical, with horizontal composition possessing a right unit but no left unit, and vertical composition possessing neither unit." The new library of Wadler's is 30% shorter and runs 30% faster than Hughess.

```
data DOC      = NIL
              | DOC :<> DOC
              | NEST Int DOC
              | TEXT String
              | LINE
              | DOC :<|> DOC
```

Listing 2.7: Definition of DOC

The DOC data type is defined recursively as the above shows. Any other value of data type is converted to DOC type before being printed by the pretty printer. The meaning of such data type is not important to users because this is not visible to them. Furthermore, the following list states the most important interfaces which are used in the implementation. Here, the ’document’ means the value of data type DOC in the library.

- <>, the associative operation that concatenates two documents together
- **text**, the function converts a string to the corresponding DOC type
- **nest**, the function that adds indentation characters to a document
- **group**, the function that ”returns the set with one new element added, representing the layout in which everything is compressed on one line” (Wadler, 2003)
- **line**, the interface that denotes a line break character

In general, using these combinators to implement a pretty printer for a certain data type can be divided into four steps:

1. Using the text function generates the original document text.
2. Inserting a line break by line and <> for document texts.
3. Inserting the indentation tag by nest function.
4. Using the group function to wrap everthing at the end.

The following code shows a function named prettyPrint which accepts a value and returns the result in a form that shows the value twice, and the second one indents eight spaces.

```
prettyPrint s = group $ (text $ show s) <>
                          nest 8 (line <> (text $ show s)) <>
                          line
{-
(prettyPrint ”this is a string example”) generates the result:
”this is a string example”
        ”this is a string example”
-}
```

Listing 2.8: Example of using combinators

Chapter 3

Design

3.1 Printing Style

The style of indentation format is not unique. Different pretty printers may perform in several styles because of the design and use of combinators. Wadlers library provides a flexible solution to help design the pretty printer. In later sections, we introduce the style in two kinds of data type.

3.1.1 Primitive Data Type

As mentioned in Chapter 2, the primitive data types are the basic elements of Haskell. They have the property of being atomic. There is no need to add line breaks in most of the value of primitive data types. Thus, all of them should be printed directly excpet lists and tuples.

```
Text.PPrinter> pprint 10
10
Text.PPrinter> pprint ”a␣string”
”a␣string”
Text.PPrinter> pprintLen 10 (1,(2,3),4,(5,6))
(1,
 (2, 3),
 4,
 (5, 6))
```

Listing 3.1: Style of primitive data type

For lists and tuples, we add a line break at the end of each element except the last one. We also need to add white spaces to align the elements vertically. We can design a style for left justification or right justification. Left justification was chosen here because the order of printing is from left to right, so it is convenient to implement.

3.1.2 User-defined Data Type

A data type which is user-defined has three forms of fixity, namely, prefix, infix and record. For record data types, the style is to align all selectors vertically with commas next to the values. For prefix and infix data types, because they are usually used to define recursive data types, such as tree, the elements should align to the others which are at the same level. One example of tree data structure is where nodes at level five should only align to level five rather than any other levels. The following code is the definition of algebraic data types by user.

```
data Trees a = Leaf a | Nod (Trees a) (Trees a)
    deriving (Generic, Show)
data Person = Person { firstName :: String,
                       lastName :: String,
                       age :: Int,
                       height :: Float,
                       addr :: String,
                       occup :: String,
                       gender :: Bool,
                       nationality :: String}
                       deriving (Generic, Show)

instance (Pretty a) => Pretty (Trees a)
instance Pretty (Person)

pers = Person ”Arthur” ”Lee” 20 (−1.75) ”Edinburgh␣UK”
              ”Student” True ”Japan”

tree1 :: Trees Int
tree1 = Nod (Nod (Leaf 333)
```

```
                 (Leaf 5555))
            (Nod (Nod(Nod(Leaf 8888)
                         (Leaf 5757))
                  (Leaf 14414))
             (Leaf 32777))

-- Example from GHC.Show
infixr 5 :^:
data Tree2 a =  Leaf2 a  |  Tree2 a :^: Tree2 a
    deriving (Generic, Show)

instance (Pretty a) => Pretty (Tree2 a)
tree2 :: Tree2 Int
tree2 = (Leaf2 89) :^: ((((Leaf2 1324324) :^:
        (Leaf2 1341)) :^: (Leaf2 (-22))) :^: (Leaf2 99))
```

Listing 3.2: Style of user-defined data type

The following demonstration shows the results of above two fixity and record types. The two tree data types have different fixity. Here, the ’tree1’ has two printable prefix constructors: ’Nod’ and ’Leaf’. Thus, the style of ’tree1’ is that each constructor should align to others in the same level. For example, the first two lines of ’tree1’ have ’(Leaf 333)’ and ’(Leaf 5555)’ and they are both in the second level. On the other hand, ’tree2’ has one infix constructor and one prefix constructor. The infix one is designed to align to the data of the left sub-tree. Finally, the value of record type ’pers’ has a style that every selectors should align vertically.

```
> pprintLen 20 tree1
Nod (Nod (Leaf 333)
         (Leaf 5555))
    (Nod (Nod (Nod (Leaf 8888)
                   (Leaf 5757))
              (Leaf 14414))
         (Leaf 32777))
> pprintLen 60 tree2
Leaf2 89 :^:
      (((Leaf2 1324324 :^: Leaf2 1341) :^: Leaf2 (-22)) :^:
       Leaf2 99)
> pprintLen 20 pers
Person {firstName = "Arthur",
```

```
        lastName = "Lee",
        age = 20,
        height = −1.75,
        addr = "Edinburgh␣UK",
        occup = "Student",
        gender = True,
        nationality = "Japan"}
```

Listing 3.3: Demonstration of style

3.1.3 Customisation

Similarly, the pretty printer has one function called pprint which is a function that can be used by programmers. For the historical reasons, there is a function called pretty in the Wadler's library. Thus, I give this function the name pprint.

The type declaration of that function could be:

```
pprint :: a -> IO()
```

Listing 3.4: Interface of pretty printer - 1

We have mentioned another function, pprintLen, which receives two arguments. The first one is the maximum width of each line. Thus, programmers can control the width. The indentation may be different if the programmer uses a different value. The second one is the target which will be printed, and it is declared with a type variable.

```
pprintLen :: Int -> a -> IO()
```

Listing 3.5: Interface of pretty printer - 2

Specifically, programmers have three choices in total. Apart from customising width, users can also customise the mode of printing style and use such style with the interface, pprintStyle. Due to time constraints, I only implement two modes: (i) ManyLineMode, the default mode of the pretty printer. The outputs of pprint and pprintLen both use this mode. (ii) OneLineMode, the result of this mode is similar to Show.

There are two selectors which should be specified. One is mode, which is chosen by users. Another is the lineLen which represents the maximum width of each line. If

OneLineMode is chosen, the selector of lineLen will not work.

```
data Mode = ManyLineMode | OneLineMode
-- | A rendering style
data Style = Style { mode    :: Mode,  -- ^ The redering mode
                     lineLen :: Int    -- ^ Length of line
                   }

pprintStyle :: Style -> a -> IO()
```

Listing 3.6: Customisation

Here is an example of customising the style of output, the values style1 and style2 specify the certain styles: The first printer of 'ManyLineMode' mode outputs the default style and each line can contain forty characters. The second printer of 'OneLineMode' mode just outputs the result in one line without any line breaks.

```
-- the type of tree is defined in Chapter 1
tree = Node "aaa" [Node "bbbbb" [Node "ccc" [], Node "dd" []],
       Node "eee" [], Node "ffff" [Node "gg" [], Node "hhh" [],
       Node "ii" []]]
style1 = Style {mode = ManyLineMode, lineLen = 40}
style2 = Style {mode = OneLineMode, lineLen = 80}

> pprintStyle style1 tree
Node "aaa"
     [Node "bbbbb"
           [Node "ccc" [],
            Node "dd" []],
      Node "eee" [],
      Node "ffff"
           [Node "gg" [],
            Node "hhh" [],
            Node "ii" []]]
> pprintStyle style2 tree
Node "aaa" [Node "bbbbb" [Node "ccc" [], Node "dd" []], Node "eee"
[], Node "ffff" [Node "gg" [], Node "hhh" [], Node "ii" []]]
```

Listing 3.7: Customisation

Above all, programmers can customise easily while using the pretty printer.

3.2 Interfaces and Type Classes

3.2.1 Pretty Printer

The first consideration in designing a generic pretty printer is how to ensure programmers use the pretty printer without touching any combinators from Wadler's pretty printer combinators. The answer is obviously that the interfaces from Wadler's library should be encapsulated into an interface wrapper.

First of all, because the user-defined data type is based on primitive data types, they should be supported by the printer directly so that Haskell can print the composite type.

Data abstraction of these kinds of types should be done. Obviously, they have a general character that could be printed by the pretty printer. Moreover, they are all constrained in the same way. Therefore, a type class 'Pretty a' is defined for solving this problem.

In this earlier design, the class, "Pretty", only has two methods:

```
pp : a -> DOC
ppList : [a] -> DOC
```

Listing 3.8: Type class: Pretty a

They can convert arbitrary data types to DOC type.

Finally, some original instances of 'Pretty' should be defined, such as (Pretty Int), (Pretty String) and so on. Therefore, the programmer can print the primitive data type without extra effort. It is also necessary to design a generic pretty printer.

3.2.2 Pretty Printer and Generic Programming

Since Glasgow Haskell Compiler(GHC) 7.2, there has been "improved support for datatype-generic programming through two features, enabled with two flags: DeriveGeneric and DefaultSignatures" (Leather, 2011). The one used here is DeriveGeneric. The compiler used for doing this project is GHC 7.10.

Generic programming supported in GHC allows defining classes with methods that do not need a user specification when instantiating: the method body is automatically derived by GHC. One example is already demonstrated in Chapter 1.

In this section, the pretty printer is extended to include the deriving mechanism. Typically, the pretty printer can be enhanced through generic representation type. That is, the programmer can represent most Haskell data types by using only the following primitive types (Leather, 2011):

```haskell
-- | Unit: used for constructors without arguments
data U1 p = U1
-- | Constants, additional parameters and recursion of kind *
newtype K1 i c p = K1 { unK1 :: c }
-- | Meta-information (constructor names, etc.)
newtype M1 i c f p = M1 { unM1 :: f p }
-- | Sums: encode choice between constructors
infixr 5 :+:
data (:+:) f g p = L1 (f p) | R1 (g p)
-- | Products: encode multiple arguments to constructors
infixr 6 :*:
data (:*:) f g p = f p :*: g p
```

Listing 3.9: Representation types

Thus, all I have to do is to tell GHC how to generate a pretty DOC type with each of these individual primitive types.

The best way to do this is to design a new helper type class. The design of this part is similar to the work of Razvan Ranca's (Ranca, 2012), but some details are different. For example, the style of indentation is a little bit different and fewer functions are used to implement the methods for adding indentation tags(white space). Because both works are based on the same paper: 'A Generic Deriving Mechanism for Haskell' by Magalhes (2010), the most different aspect is the use of combinators.

3.3 Model Comparison

Compared to Razvan Ranca's work, the number of helper functions in my pretty printer is fewer. I believe this is an improvement. On the other hand, Razvan Ranca's pretty printer is based on the combinators designed by Hughes (1995). My pretty printer is based on the one designed by Wadler (2003) so the implementation is simpler and the running speed is quicker than Razvan Ranca's.

Chapter 4

Implementation

4.1 Pretty Printer

To illustrate the implementation of the pretty printer, we should first define the type class Pretty.

```
class Pretty a where
  pp :: a -> DOC
  ppList :: [a] -> DOC
```

Listing 4.1: Type class: Pretty a

The functions pp and ppList return the value of the DOC data type. Thus, we use the existing combinators from Wadler's pretty printer as mentioned in Chapter 2. From my perspective, the function of class Pretty can be seen as the entrance of the pretty printer. The three most important interfaces namely, nest, group and line are used to build the class.

The first target is to implement all the instances of primitive data types. Haskell has a finite number of these types, so it is possible to deal with all of them. Specifically, whatever the type it is, the value should first be converted to a printable string. Then it is converted to a DOC value.

To convert the value of a data type to string, one way is to design several helper functions for conversion. For instance, an 'int' function could accept an integer value and return a DOC value. Therefore, we need to implement all kinds of functions of every primitive data type in Haskell.

However, a better way to implement it is to use the ”show” function directly. It is much simpler if we use ”show” instead of implementing every individual helper function. Because ”show” provides parametric polymorphism, it is suitable to help us to simplify the complexity of implementation. Thus, the idea to implement the instances is the same. First, they all need to call show and get a string value. Then passing the value to the function text obtains the DOC value. Finally, the DOC value should deal with the interfaces nest, line and group, if needed. The following example shows the definition of an instance of Pretty Bool. We do not need to use other interfaces here because the structure of primitive data type is atomic, as usual.

```
instance Pretty Bool where
  pp b = text (show b)
```

Listing 4.2: Example of an instance of Pretty Bool

Because the list data type is the very common in Haskell, it is worth defining an instance of Pretty a => Pretty [a]. This will call the function ppList so that we can control the output style of a list.

```
instance Pretty a => Pretty [a] where
  pp = ppList
```

Listing 4.3: Example of list

The implementation of ppList consists of nest, group and line functions. Each list is wrapped with a pair of square parentheses. The comma separates two adjacent elements of the list. A new line should be added after the comma notation. The indentation is inserted from the first element to the end by nest. To produce valid DOC data, we also need to use the group function.

```
 -- helper function for generating a DOC list
 genList :: [a] -> DOC
 genList [] = nil
 genList (x:xs) = text "," <>
                  line <> whiteSpace <>
```

```
                 nest indent (pp x) <>
                 genList xs

-- | ’ppList’ is the equivalent of ’Prelude.showList’
--
ppList :: [a] -> DOC
ppList []     = text ”[]”
ppList (x:xs) = group (
                text ”[” <>
                nest indent (pp x) <> genList xs <>
                text ”]”)
```

Listing 4.4: Implementation of ppList

The implementation of all the instances is trivial except Pretty String. The main problem is that String in Haskell is a synonym. Specifically, it is same as [Char]. If we want to print a value of String type, Haskell uses the instance Pretty a => Pretty [a]. This will give an unexpected result such as ”['a', 'b', 'c']” rather than ”abc” corresponding to the input ”abc”, but we hope the result looks like an individual string not a list. Thus, I refer to the implementation of Show String.

All that is needed is to define a new function, ppList, in the instance Pretty Char. That function has the same name of class Pretty. When the type of value is String, Haskell calls the ppList of instance Pretty Char, not class Pretty.

```
ppList str = text $ show str
```

Listing 4.5: Pretty printing example

For more details of the full implementation code, see Appendix A.

4.2 Generic Pretty Printer

In the last section, we talked about the definition of class Pretty. At this stage, if programmers want to use it, they have to define the instances by themselves. The following is a good example from 'A prettier printer' by Wadler (2003). Users not only need to define data type but also need to define the rules about how to print the result.

Specifically, it is desirable that users do not touch the combinators and even do not need to implement the instance. From the Chapter 3, we know that the Glasgow Haskell Compiler has the feature of generics, therefore, we can do generic programming in Haskell. We should define the class generically. Hence, when users try to define an instance of the type class, the compiler finishes the work instead of being done by users themselves.

```haskell
class GPretty f where
  -- 'gpp' is the (*->*) kind equivalent of 'pp'
  gpp     :: Type     -- The type of fixity. Record, Infix or Prefix.
             -> Int   -- The operator precedence
             -> Bool  -- Flag that marks if the constructors was
                      -- wrapped in parens
             -> f a
             -> [DOC] -- The result.

  -- 'nullary' marks nullary constructors
  nullary :: f x -> Bool
```

Listing 4.6: Helper type class GPretty f

We define a helper class GPretty, with gpp and nullary as the methods. Because 'gpp' is the $(*- > *)$ kind, we cannot define it in class Pretty. From the design chapter, we know that Haskell has some primitive representation types. For each representation type, there is an instance of gpp. We also need to record some necessary information, such as the current operator precedence, and the flag that marks whether the constructors was wrapped in parentheses. Moreover, the Infix data type has a different indentation method from prefix type and record types, so we need to consider this for the product operator.

In conclusion, gpp methods have four arguments: (i) Type of multiplication. (2) The operator precedence. (iii) A flag (iv) The sum of products representation of the user-defined type. The nullary method is explained later. We treat the return type as a list which is convenient when inputting the new line and white spaces for indentation.

Now, we talk about the implementation of each representation type. We use the order whereby the complexity of implementation will gradually increases.

4.2.1 Unit Type

Because the parentheses do not need to be put around a unit type, we do nothing and return an empty list. On the other hand, when we meet a unit, it means that there is a constructor with no arguments. Thus, the nullary method should return True here.

```haskell
instance GPretty U1 where
  gpp _ _ _ _ = []
  nullary _   = True
```

Listing 4.7: GPretty U1

4.2.2 K1 tag - Additional Information

This kind of type always has a tag, K1. It saves the constant, additional parameters and recursion of the kind * (Leather, 2011). However, the tagging is useless; we just ignore it. The remaining aspect has a concrete type, so we pass this to ppPrec. Clearly, the nullary function should return False this time.

```haskell
instance (Pretty a) => GPretty (K1 i a) where
  gpp _ n _ (K1 x) = [ppPrec n x]
  nullary _        = False
```

Listing 4.8: GPretty (K1 i a)

4.2.3 Sums

Sums encode choice between constructors. Thus, ignoring the tagging is the only thing we need to deal with. As we cannot determine if the constructor has arguments or not, we call the nullary function recursively.

```haskell
instance (GPretty a, GPretty b) => GPretty (a :+: b) where
  gpp t d b (L1 x) = gpp t d b x
  gpp t d b (R1 x) = gpp t d b x
  nullary (L1 x) = nullary x
  nullary (R1 x) = nullary x
```

Listing 4.9: GPretty (a :+: b)

4.2.4 Products

Products encode multiple arguments to constructors because the form of data type can be infix, prefix and record. We need to define three implementations of each kind of type. The style of each type is explained in the design chapter. Here we only introduce the Haskell code.

The implementation of record type and prefix type is trivial. The only difference is that we use comma notation to separate each arguments. These are inserted in a new line between arguments and repeated recursively.

Infix type is harder to be defined because we need to determine how many white spaces need to be added by the nest function. One possible way is to count the length of characters before the first left parenthesis (parens) and the length of non-space characters after the first left parenthesis (white). Then we can use these two values to compute the indentation length.

```
instance (GPretty a, GPretty b) => GPretty (a :*: b) where
  gpp t1@Recordt d flag (a :*: b) = gppa ++ [comma, line] ++ gppb
    where
      gppa = gpp t1 d flag a
      gppb = gpp t1 d flag b

  gpp t1@Prefixt d flag (a :*: b) = gppa ++ [line] ++ gppb
    where
      gppa = gpp t1 d flag a
      gppb = gpp t1 d flag b

  gpp t1@(Infixt s) d flag (a :*: b) = init gppa ++
                                       [last gppa <+> text s] ++
                                       addWhitespace gppb
    where
      gppa = gpp t1 d flag a
      gppb = gpp t1 d flag b

```

```
      addWhitespace :: [DOC] -> [DOC]
      addWhitespace [] = []
      addWhitespace m@(x:xs)
        | paren == 0 = if flag then map (nest 1) (line : m) else
                       line : m
        | otherwise = map (nest $ white + 1 +
                      (if flag then 1 else 0)) (line :  m)
        where
          sa = Prelude.filter (\x -> x /= '\n') $ pretty layout 1 x
          sb = Prelude.filter (\x -> x /= '\n') $ pretty layout 1
               (head gppa)
          paren = length $ takeWhile (== '(') sa
          white = length $ takeWhile( /= ' ') (dropWhile(== '(') sb)

  nullary _ = False
```

Listing 4.10: GPretty (a :*: b)

If the value of parens is equal to zero, we only put one more white space before a new line while the constructor is wrapped in parentheses. Otherwise, we put a single new line here.

If the value of parens does not equal zero, we should also consider the length of non-space characters (white) here.

Finally, the nullary function should return False because of the arguments.

4.2.5 Meta-information

To illustrate the use of selector and constructor labels, I refer to the implementation of generic Show in the paper: 'A Generic Deriving Mechanism for Haskell' by Magalhes (2010). For one thing, we should ignore the M1 tagging of the instance Datatype and do nothing with it.

```
instance (GPretty a, Datatype c) => GPretty (M1 D c a) where
  gpp t d b (M1 x) = gpp t d b x
  nullary (M1 x)   = nullary x
```

Listing 4.11: GPretty (M1 D c a)

The most interesting implementation of instances is for the meta-information of a constructor and a selector of a generic pretty printer. For a selector, we print the label of selector as long as it is not empty, which is followed by an equality notation and its value. We also should ignore the M1 tag to check if it is a nullary value.

```haskell
instance (GPretty f, Selector c) => GPretty (M1 S c f) where
  gpp t d b s@(M1 a)
               | null selector = gpp t d b a
               | otherwise = (text selector <+>  char '=' <>
               whiteSpace) :
               map (nest $ length selector + 2) (gpp t 0 b a)
      where
          selector = selName s

  nullary (M1 x) = nullary x
```

Listing 4.12: GPretty (M1 S c f)

For a constructor, two things should be determined here. One is whether the parentheses are wrapped or not. Another is how many white spaces should be added by the nest function. For simplicity, we do a classified discussion of possible fixities and record.

```haskell
instance (GPretty f, Constructor c) => GPretty (M1 C c f) where
  gpp _ d b c@(M1 a) =
    case conFixity c of
      Prefix -> wrapParens checkIfWrap $
        text (conName c) <> whiteSpace
        : addWhitespace checkIfWrap (wrapRecord (gpp t 11 b a))
      Infix _ l ->
        wrapParens (d > l) $ gpp t (l + 1) (d > l) a
      where
        t = if conIsRecord c then Recordt else
            case conFixity c of
              Prefix    -> Prefixt
              Infix _ _ -> Infixt (conName c)

        checkIfWrap = not (nullary a) && (d > 10)

        -- add whitespace
        addWhitespace :: Bool      -- check if wrap parens
                       -> [DOC]
```

```
                 -> [DOC]
      addWhitespace _ [] = []
      addWhitespace b s | conIsRecord c = s
                        | otherwise = map
           (nest $ length (conName c) + if b then 2 else 1) s

      -- add braces for record
      wrapRecord :: [DOC] -> [DOC]
      wrapRecord [] = []
      wrapRecord s | conIsRecord c = wrapNest s
                   | otherwise = s
               where
                wrapNest2      = foldr
              (\x -> (++) [nest (length (conName c) + 2) x])
              [text "}"]
                wrapNest (x:xs) = nest
                (length (conName c) + 1) (text "{" <> x) :
                wrapNest2 xs

      -- add Parens
      wrapParens :: Bool       -- add parens or not
               -> [DOC]
               -> [DOC]
      wrapParens _ [] = []
      wrapParens False s = s
      wrapParens True (x:xs) = lpar <> x : wrapParens2 xs
             where
              wrapParens2 = foldr (:) [rpar]

  nullary (M1 x) = nullary x
```

Listing 4.13: GPretty (M1 C c f)

- **Record**, We should wrap curly braces for record and add whitespaces.
- **Prefix**, There is nothing special to be done for this kind of type. We just add the constructor name, nest the result and possibly put it in parentheses.
- **Infix**, The only thing is possibly to be put in parentheses.

Here the real type and parenthesis flag are set and propagated forward via t and checkIfWrap, so the precedence factor is updated. For prefix type, we always place parentheses around a constructor except a nullary one. For infix type, we wrap parentheses if the previous constructor's precedence is bigger than the current one's.

4.2.6 Generic Default Method

Finally, we provide the default and a new method ppPrec in class ”Pretty” which saves one more integer type. This integer type can save the operator precedence of the enclosing context.

```
class Pretty a where
  ppPrec :: Int -> a -> DOC
  default ppPrec :: (Generic a, GPretty (Rep a)) => Int -> a -> DOC
  ppPrec n x = rep $ gpp Prefixt n False (from x)

  pp      :: a -> DOC
  default pp :: (Generic a, GPretty (Rep a)) => a -> DOC
  pp x = rep $ gpp Prefixt 0 False (from x)
```

Listing 4.14: Adding default keywords

Now, we implement the complete pretty printer. Please see Appendix A for further details.

Chapter 5

Evaluation

The project was evaluated from two aspects. One is a user study for the ability evaluation. Another uses the testing framework that contained the API to design a tester and tested a certain number of test cases.

5.1 Method

5.1.1 Method of Ability Evaluation

As mentioned in the Chapter 2, it is quite hard to measure whether the pretty printer outputs a 'pretty' result or not. Allowing users to assess quantitatively how pretty the result is may lead to confusion.

However, a solution for this problem exists. The reason why people need a prettier printer in Haskell is that the pretty printer can help to improve the efficiency in analysing and/or understanding the result. Thus, the proper way to evaluate the pretty property could be to interview the users and ask them 'Do you think it is pretty?' directly.

An alternative is to conduct a user study to determine whether the output of the pretty printer is easier to read than the output of Show. The results of the study should be collected. This might be tested by asking users to look at output from each and answer questions. If the answers are returned quicker or more correct for pretty rather than Show, then the evaluation demonstrates that the pretty printer is an improvement over

Show. Therefore, the ability of a generic pretty printer can be evaluated.

5.1.2 Testing Method and Acquiring Test Cases

Another property of a pretty printer is that the original data type value should be the same as the one generated by Show in Haskell. In other words, the results of the pretty printer should be the same as Show's if all the indentation characters and new lines were eliminated. The testing plan is simply to compare the results between the pretty printer and Show, introduced later.

To get data types for testing, either make some up or to take them from a repository of Haskell programs or both. In this project, 51 different data types are tested in this way. It covers almost all kinds of data types in Haskell.

5.2 Ability Evaluation

A user study for evaluation needs at least three steps. First, the form of the study is designed for getting the feedback from the user. This is a key link in the whole progress. Second, the evaluation is run successfully. Finally, the result of evaluation is analysed.

5.2.1 Design

The best form of the evaluation is an interview, because an interview is convenient to measure the time (efficiency evaluation) and to provide instructions face to face. Therefore, a standard questionnaire survey was designed.

The reason to design the questionnaire is because it can help to analyse the data collected from it. The following are those simple questions which enable the study to find the most important elements of a pretty printer which demonstrates that pretty is an improvement over Show in the evaluation. Each question has an explanation for its design.

The questionnaire here is incomplete. It only includes several of the most important questions which cover all the points of evaluation discovered in the method section.

A complete version of the questionnaire can be found at `http://goo.gl/forms/4vwwuUBVdlSk9ZKv1`. The following questions are divided into three parts: basic information, user experience and efficiency.

(i) Part 1 - Basic Information

(1) Have you tried pretty printer in Haskell before? (Yes/No)

This basic information can help us to determine if they have a relevant background.

(2) Do you think print datatypes out is helpful during development? Why you need that? (Long-answer text)

This question can help to find out if people like checking the result of data types or not.

(ii) Part 2 User Experience of Pretty Printer

(1) If you have tired implementing some complex instances in Haskell. How difficult do you think it is if you implement the instances by your own? (Vote 1 to 5 here; 1 is the easiest, 5 is the hardest)

This question can help to find out how easy it is to implement the instance by users themselves.

(2) If you have tried deriving feature in Haskell. How difficult do you think it is if you implement the instances with the deriving feature of Haskell? (Vote 1 to 5 here; 1 is the easiest, 5 is the hardest)

This question can help to find out how easy it is to implement the instances with the deriving facility in Haskell.

(3) In Generic Pretty Printer, the default function 'printer' has the only parameter name of data type. Do you think it is a good design and why? (Long-answer text)

This question can help to find how user-friendly it is to use the generic pretty printer.

(iii) Part 3 Efficiency Evaluation of Pretty Printer

There are three questions designed in part 3, each has three sub-questions. Only one of them will be introduced here, because they are under the same mode. The aim for this part is to judge that if the answers for each questions are quicker found out for pretty printer rather than show.

Question: Compare the following two print output of a binary tree which are the same result presented in different ways. Please find out the height of each tree and estimate the time you use.

(If you do not familiar with the concept height, please refer to `https://en.wikipedia.org/wiki/Binary_tree`)

```
Definition: data Trees a = Leaf a | Node (Trees a) (Trees a)
                 deriving (Generic, Show)

Tree A:
Node (Node (Leaf 333) (Leaf (-5555))) (Node (Node (Node (Leaf 8888)
(Leaf 5757)) (Leaf (-14414))) (Leaf 32777))

Tree B:
Node (Node (Leaf (333),
            Leaf ((-5555))),
        Node (Node (Node (Leaf (8888),
                                             Leaf (5757)),
                           Leaf ((-14414))),
              Leaf (32777)))
```

Listing 5.1: Code of questionnaire

(1) How much time you spend when you count the height of tree A?

(2) How much time you spend when you count the height of tree B?

(3) Which one is more prettier? (A/B/Both)

5.2.2 Conduct the Evaluation

The interviews were conducted with users face to face. I personally timed the users in part three of the questionnaire so the accuracy of each single result can be guaranteed.

5.2.3 Analysing the Results of the Ability Evaluation

The results are based on analysing statistical data to conclude which factors are more appropriate for most users of pretty printer. However, the survey did not always get an ideal result because of human factors. Not all the people think a generic pretty printer is for common use. The notwithstanding, the result is sufficient to generate a conclusion.

Figure 5.1 is the statistical result of the question: Have you tried the pretty printer in Haskell before? The figure shows that only half the interviewees have tried the pretty printer before.

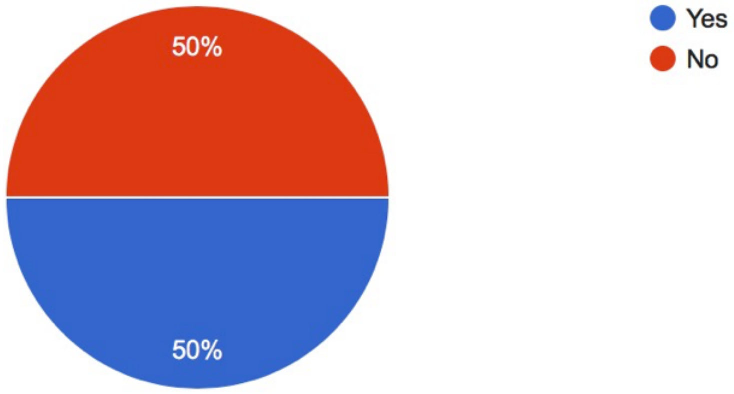
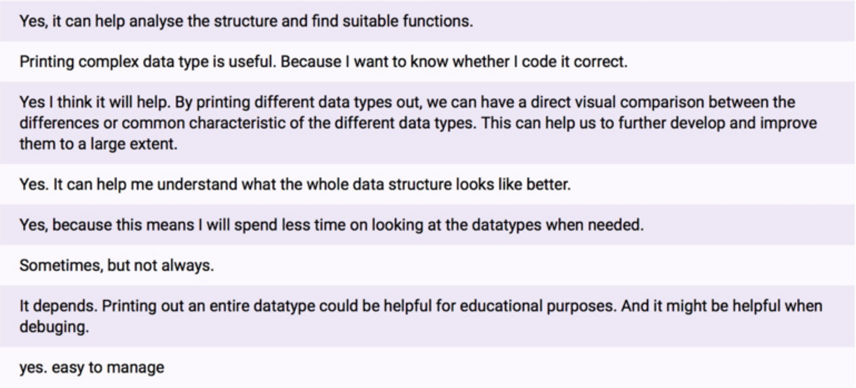


Figure 5.1: Verbatim answers to Part1, question 1.

Figure 5.2 is a collection of answers to the question: Do you think print datatypes out is helpful during development? Why you need that? And all the interviewees gave the positive answers and most were satisfied with their user-experience of pretty printer. This demonstrates that the pretty printer is an improvement.



Yes, it can help analyse the structure and find suitable functions.

Printing complex data type is useful. Because I want to know whether I code it correct.

Yes I think it will help. By printing different data types out, we can have a direct visual comparison between the differences or common characteristic of the different data types. This can help us to further develop and improve them to a large extent.

Yes. It can help me understand what the whole data structure looks like better.

Yes, because this means I will spend less time on looking at the datatypes when needed.

Sometimes, but not always.

It depends. Printing out an entire datatype could be helpful for educational purposes. And it might be helpful when debuging.

yes. easy to manage

Figure 5.2: Verbatim answers to Part1, question 2.

By timing the average speed of answering the question in part 3, I noticed that interviewees spent much more time finding out the result of Tree A than the result of Tree

B. Figure 5.3 shows that most people think Tree B is prettier than Tree A.

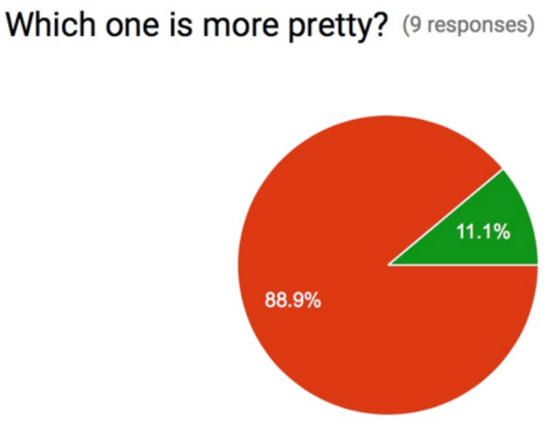


Figure 5.3: Proportion of interviewee preference for Tree A and Tree B

To sum up, the generic pretty printer performs better than Show in output. The result demonstrates that the pretty printer is an improvement over Show in Haskell. In addition, the pretty printer also provides user-friendly interfaces to programmers. Therefore, the ability of pretty printer is worth being spread.

5.3 Testing

Unlike an ability evaluation, testing is a rigorous way to evaluate whether the value of the result of the pretty printer is reasonable. The method of testing has been defined in the method section. The aim of testing is to help fix any bugs of the pretty printer. The tester cannot guarantee to test all possible cases since the number of user-defined data types is infinite. Due to the time constraints of the project, only the most common test cases are tested.

5.3.1 Design

The architecture of the tester is that it enumerates test cases as much as possible and then uses Test.QuickCheck, a library in Haskell, to generate random values of a given data type. Then, Show and the pretty printer are checked to see whether they produce identical strings, not counting whitespace. Finally, the pretty printer is checked to see that it produces strings no wider than the given width.

A key problem of the tester is comparing two values. It is easy to use (==) to compare two integers, but difficulties arose when implementing one test unit that compared two lists of String.

```
[”a\na”,
 ”b”,
 ”c”]
```

Listing 5.2: A test case

The result of function, show does not include any new line at the end of a comma. Thus, testers should normalise the white space (turn a sequence of spaces and newlines to one space). However, there is a better a better solution, which is to read back the printed data and check if it yields the original data. Furthermore, a parser could be implemented when the tester tries to compare two algebraic data types.

However, the tester cannot work out whether the result is printed in the correct format. One solution to this problem is to let other people try the pretty printer and report the bugs.

5.3.2 Implementation of Tester

Hspec is a good testing framework for Haskell maintained by Spangler (2011). The main function of then tester was implemented with it and Test.QuickCheck developed by Koen Classen (2006). The parser of one test unit is implemented with the library Text.Parsec developed by Leijen (2006). Now it owns 51 test cases. It almost guarantees that the pretty printer can work correctly. Some of the test cases are made up by myself. Others are obtained from the repository of Haskell programs.

(i) Some test cases

The following data type is one example from Text.Show maintained by Ross Paterson (2009).

```
infixr 5 :^:
data Tree2 a = Leaf2 a  |  Tree2 a :^: Tree2 a
     deriving (Generic, Show)
```

Listing 5.3: Data type of tree

The recursive tree data type (Tree2 a) has an infix constructor (:^:). Because of the deriving facility, the instance of a pretty printer of (Tree2 a) does not need to be implemented.

```
instance (Pretty a) => Pretty (Tree2 a)
```

Listing 5.4: Instance

Here, tree2 is a test case for testing the result of pretty printer printing a data type with an infix constructor.

```
tree2 :: Tree2 Int
tree2 = (Leaf2 89) :^: ((((Leaf2 1324324) :^: (Leaf2 1341)) :^:
        (Leaf2 (−22))) :^: (Leaf2 99))
```

Listing 5.5: Value of data type

To make up some primitive data types, such as integer, the tester can call the property function from QuickCheck that generates random data directly.

(ii) Test units of primitive data type

Hspec has a friendly DSL(domain specific language) for defining tests. The main advantages of Hspec can be found at the official website. The following list is a part of the main function with two test units.

```
main :: IO ()
main = hspec $ do
  describe ”Primitive␣Data␣Type␣Testing” $ do
    it ”Unit” $ property $
    \x −> omitNewline (pShow x ()) ‘shouldBe‘ show ()
    it ”Num␣:␣test␣positive␣integer” $
    omitNewline (pShow 10 (10 ::Int)) ‘shouldBe‘ show 10
```

Listing 5.6: Main function of tester

To use this DSL for defining the test case, several position should be changed. First, the string after the describe function is the title of the test and the string after its function is the subtitle of each test case. Using the property function and lambda calculus together can make it test some random data, or the test unit can be defined without the property function. The shouldBe function is the same as (==). Using shouldBe here is only a syntactic sugar of this DSL.

Where pShow is a utility function defined as follows, it returns a value of String type. This matches the type of return value of Show.

```
pShow w x = pretty layout w (pp x <> line)
```

Listing 5.7: Utility function

In my tester, I defined more than 50 test cases including the test units of Unit, Number (Float, Double, Integer), Char and String etc. The full code of the tester can be found in the Appendix.

(iii) A test unit of algebraic data type

The following test unit is an example for testing a list of integers. The test cases are wrapped with a function, testList. This is a parser which can parse all the elements of the list. The tester can compare the result which is produced by the parser to determine if they are the same.

```
it "List : [Int]" $ property $
  \x y -> testList (pShow x (y :: [Int])) `shouldBe`
          testList (show y)
```

Listing 5.8: Test unit

I use a library called Text.Parsec. It provides many useful interfaces and functions and the syntax of defining a parser with it is also clear. The following code shows that how to use a parser. The listParser is a function that defines a specific parser.

```
testList = parse listParser ""
```

Listing 5.9: Parser example

The following is the implementation of stringParser which can parse a string value which is wrapped by quotation marks. The manyTill function means that the parsing stops upon meeting some rules. The rules here are defined as functions, such as a letter function which means parsing the letter. (<|>) is syntax sugar that for matching one of multiple rules.

```
stringParser :: Parser String
stringParser = do
          char '\”'
␣␣␣␣␣␣␣␣␣␣manyTill␣(letter␣<|>␣digit␣<|>␣space)
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣(char␣'\”')␣<|>␣string␣”\\\””
```

Listing 5.10: Parser

With the help of Parsec, any parser can be defined for parsing the value of data type. In my tester, I defined two different tree parsers, infix constructor parser, Map parser, list parser and record parser. The full code of the parser is included in the tester which can be found in the Appendix.

5.3.3 Running Tests and Inspecting the Result of Testing

The following is the result of running the test suites. After inspecting the result of testing, I fixed a bug in printing Map. The tester reported a bug in Map whereby the pretty printer did not output the string of fromList. This error is because of the implementation instance (Pretty a, Pretty b) => Pretty (Map a b) in the pretty printer. There are also some other small bugs reported by the tester, for instance, the pretty printer threw an exception when it tested a list. These were fixed and now works well.

Finally, I found the cause of some other errors and fixed the implementation of pretty printer. These are:

- **Map**, one instance of type class Pretty.
- **Just**, one instance of type class Pretty.
- **Printing style**, the implementation of conducting the constructor and selector

The result of testing is shown in the following table.

Name of Test	Test Content	Result
Unit	()	Pass
Integer	positive integer test	Pass
Integer	negative integer test	Pass
Integer	big integer test	Pass

Integer	random integer test	Pass
Float	random float number test	Pass
Double	random double number test	Pass
Char	random character test	Pass
String	random string test	Pass
Bool	random Boolean test	Pass
Map	map test 1	Pass
Map	map test 2	Pass
Map	map test 3	Pass
Ordering	EQ, LT, GT test	Pass
Maybe	Nothing	Pass
Maybe	Maybe Bool	Pass
Maybe	Maybe Int	Pass
Maybe	Maybe Char	Pass
Maybe	Maybe String	Pass
Maybe	Maybe Float	Pass
Maybe	Maybe Double	Pass
Maybe	Maybe Ordering	Pass
Either	Left Int	Pass
Either	Right Integer	Pass
Either	Left (Maybe Int)	Pass
Either	Right Bool	Pass
Either	Left Char	Pass
Either	Right String	Pass
Either	Left Float	Pass
Either	Right Double	Pass
Pair	(a, b)	Pass
Triple	(a, b, c)	Pass
Tuple	(a, b, c, d)	Pass
Tuple	(a, b, c, d, e)	Pass
Tuple	(a, b, c, d, e, f)	Pass
Tuple	(a, b, c, d, e, f, g)	Pass
Tuple	(a, b, c, d, e, f, g, h)	Pass
Tuple	(a, b, c, d, e, f, g, h, i)	Pass
Tuple	(a, b, c, d, e, f, g, h, i, j)	Pass

Tuple	(a, b, c, d, e, f, g, h, i, j, k)	Pass
Tuple	(a, b, c, d, e, f, g, h, i, j, k, l)	Pass
List	[Int]	Pass
List	[Float]	Pass
List	[Double]	Pass
List	String	Pass
List	[String]	Pass
List	[Bool]	Pass
Record	test 1	Pass
Tree	Int	Pass
Infix stype	data Foo a b = a :**: b	Pass
Tree	example from GHC.Show	Pass

Table 5.1: Testing Result

Chapter 6

Conclusion

Pretty-printing is certainly a useful function. The results show that my pretty printer works as required. However, it still has many aspects that need to be improved. For example, it is possible to design a powerful customisation mechanism for this pretty printer in the future. Due to limitations, we cannot derive generic instances for: Datatypes with a context; existentially-quantified datatypes; GADTs (Leather, 2011).

Finally, this project provides a general solution for deriving generic pretty-printing for Haskell, and an automatic tester for evaluation is developed.

Appendix A

Code of Generic Pretty Printer

```haskell
-- This library is also available at https://hackage.haskell.org/package/PPrinter

{-# LANGUAGE DeriveGeneric, TypeOperators, FlexibleInstances, FlexibleContexts, DefaultSignatures #-}

module Text.PPrinter (
  Pretty(..),
  Style(..),

  -- Instances for Pretty: (), Bool, Ordering, Int, Integer, Char, String, Float, Double

  -- Pretty support code
  pprint, pprintLen, pprintStyle,
  Generic
  ) where

import Data.Map hiding (showTree, map, null, foldr)
import GHC.Generics
import Data.List (null)
import Data.Char

infixr 5    :<|>
infixr 6    :<>
infixr 6    <>
infixr 6    <+>
infixr 6    <->

data DOC      = NIL
              | DOC :<> DOC
              | NEST Int DOC
              | TEXT String
              | LINE
              | DOC :<|> DOC
                deriving(Show)

data Doc      = Nil
              | String `Text` Doc
              | Int `Line` Doc
                deriving(Show)

-- interface

nil           = NIL
x <> y        = x :<> y
x <+> y       = x <> whiteSpace <> y
nest          = NEST
text          = TEXT
line          = LINE

```

```
lpar         = text ”(”
rpar         = text ”)”
comma        = text ”,”
whiteSpace   = text ”␣”
parens s     = lpar <> s <> rpar

group x      = flatten x :<|> x

indent       = 1

-- implementation

flatten NIL           = NIL
flatten (x :<> y)     = flatten x :<> flatten y
flatten (NEST i x)    = NEST i (flatten x)
flatten (TEXT s)      = TEXT s
flatten LINE          = TEXT ”␣”
flatten (x :<|> y)    = flatten x


layout Nil            = ””
layout (s ‘Text‘ x)   = s ++ layout x
layout (i ‘Line‘ x)   = ’\n’ : copy i ’ ’ ++ layout x

-- interfaces for oneLineMode
oneLayout Nil            = ””
oneLayout (s ‘Text‘ x)   = s ++ oneLayout x
oneLayout (i ‘Line‘ x)   = ’ ’ : oneLayout x

copy i x              = [ x | _ <- [1 .. i] ]

best w k x            = be w k [(0, x)]

be w k []                 = Nil
be w k ((i,NIL):z)        = be w k z
be w k ((i,x :<> y):z)    = be w k ((i,x):(i,y):z)
be w k ((i,NEST j x):z)   = be w k ((i+j,x):z)
be w k ((i,TEXT s):z)     = s ‘Text‘ be w (k+length s) z
be w k ((i,LINE):z)       = i ‘Line‘ be w i z
be w k ((i,x :<|> y):z)   = better w k (be w k ((i,x):z))
                                       (be w k ((i,y):z))

better w k x y            = if fits (w-k) x then x else y

fits w x | w < 0          = False
fits w Nil                = True
fits w (s ‘Text‘ x)       = fits (w − length s) x
fits w (i ‘Line‘ x)       = True


-- class GPretty

data Type = Infixt String | Prefixt | Recordt

class GPretty f where

  -- ’gpp’ is the (*->*) kind equivalent of ’pp’
  gpp    :: Type      -- The type of fixity. Record, Infix or Prefix.
            -> Int    -- The operator precedence
            -> Bool   -- Flag that marks if the constructors was wrapped in parens
            -> f a
            -> [DOC]  -- The result.

  -- ’nullary’ marks nullary constructors
  nullary :: f x -> Bool

instance GPretty U1 where
  gpp _ _ _ _ = []
  nullary _   = True

-- ignore tagging
-- K1 : Constants, additional parameters and recursion of kind *
```

```haskell
instance (Pretty a) => GPretty (K1 i a) where
  gpp _ n _ (K1 x) = [ppPrec n x]
  nullary _        = False

instance (GPretty a, GPretty b) => GPretty (a :+: b) where
  gpp t d b (L1 x) = gpp t d b x
  gpp t d b (R1 x) = gpp t d b x
  nullary (L1 x) = nullary x
  nullary (R1 x) = nullary x

instance (GPretty a, GPretty b) => GPretty (a :*: b) where
  gpp t1@Recordt d flag (a :*: b) = gppa ++ [comma, line] ++ gppb
    where
      gppa = gpp t1 d flag a
      gppb = gpp t1 d flag b

  gpp t1@Prefixt d flag (a :*: b) = gppa ++ [line] ++ gppb
    where
      gppa = gpp t1 d flag a
      gppb = gpp t1 d flag b

  gpp t1@(Infixt s) d flag (a :*: b) = init gppa ++ [last gppa <+> text s] ++ addWhitespace gppb
    where
      gppa = gpp t1 d flag a
      gppb = gpp t1 d flag b

      -- add whitespace
      addWhitespace :: [DOC] -> [DOC]
      addWhitespace [] = []
      addWhitespace m@(x:xs)
        | paren == 0 = if flag then map (nest 1) (line : m) else line : m
        | otherwise = map (nest $ white + 1 + (if flag then 1 else 0)) (line :  m)
        where
          sa = Prelude.filter (\x -> x /= '\n') $ pretty layout 1 x
          sb = Prelude.filter (\x -> x /= '\n') $ pretty layout 1 (head gppa)
          paren = length $ takeWhile (== '(') sa
          white = length $ takeWhile( /= ' ') (dropWhile(== '(') sb)

  nullary _ = False

-- ignore datatype meta-information
-- data D : Tag for M1: datatype
instance (GPretty a, Datatype c) => GPretty (M1 D c a) where
  gpp t d b (M1 x) = gpp t d b x
  nullary (M1 x)   = nullary x

-- selector, display the name of it
-- data S : Tag for M1: record selector
instance (GPretty f, Selector c) => GPretty (M1 S c f) where
  gpp t d b s@(M1 a)
               | null selector = gpp t d b a
               | otherwise = (text selector <+>  char '=' <> whiteSpace) : map (nest $ length selector + 2) (gpp t 0 b a)
      where
          selector = selName s

  nullary (M1 x) = nullary x

-- constructor, show prefix operators
-- data C : Tag for M1: constructor
instance (GPretty f, Constructor c) => GPretty (M1 C c f) where
  gpp _ d b c@(M1 a) =
    case conFixity c of
      Prefix -> wrapParens checkIfWrap $
        text (conName c) <> whiteSpace
        : addWhitespace checkIfWrap (wrapRecord (gpp t 11 b a))
      Infix _ l ->
        wrapParens (d > l) $ gpp t (l + 1) (d > l) a
      where
        t = if conIsRecord c then Recordt else
              case conFixity c of
                Prefix   -> Prefixt
                Infix _ _ -> Infixt (conName c)
```

```
        checkIfWrap = not (nullary a) && (d > 10)

        -- add whitespace
        addWhitespace :: Bool      -- check if wrap parens
                      -> [DOC]
                      -> [DOC]
        addWhitespace _ [] = []
        addWhitespace b s | conIsRecord c = s
                          | otherwise = map (nest $ length (conName c) + if b then 2 else 1) s

        -- add braces for record
        wrapRecord :: [DOC] -> [DOC]
        wrapRecord [] = []
        wrapRecord s | conIsRecord c = wrapNest s
                     | otherwise = s
                     where
                       wrapNest2        = foldr (\x -> (++) [nest (length (conName c) + 2) x]) [text "}"]
                       wrapNest (x:xs) = nest (length (conName c) + 1) (text "{" <> x) : wrapNest2 xs

        -- add Parens
        wrapParens :: Bool        -- add parens or not
                   -> [DOC]
                   -> [DOC]
        wrapParens _ [] = []
        wrapParens False s = s
        wrapParens True (x:xs) = lpar <> x : wrapParens2 xs
                   where
                     wrapParens2 = foldr (:) [rpar]

  nullary (M1 x) = nullary x


class Pretty a where

  -- | 'ppPrec' converts a value to a pretty printable DOC.
  --
  ppPrec :: Int    -- ^ the operator precedence of the enclosing context
         -> a      -- ^ the value to be converted to a 'String'
         -> DOC -- ^ the result
  default ppPrec :: (Generic a, GPretty (Rep a)) => Int -> a -> DOC
  ppPrec n x = rep $ gpp Prefixt n False (from x)

  -- | 'pp' is the equivalent of 'Prelude.show'
  --
  pp      :: a -> DOC
  default pp :: (Generic a, GPretty (Rep a)) => a -> DOC
  pp x = rep $ gpp Prefixt 0 False (from x)

  -- helper function for generating a DOC list
  genList :: [a] -> DOC
  genList [] = nil
  genList (x:xs) = comma <>
                   line <> whiteSpace <>
                   nest indent (pp x) <>
                   genList xs

  -- | 'ppList' is the equivalent of 'Prelude.showList'
  --
  ppList :: [a] -> DOC
  ppList []      = text "[]"
  ppList (x:xs) = group $
                  text "[" <>
                  nest indent (pp x) <> genList xs <>
                  text "]"
  {-# MINIMAL ppPrec | pp #-}


instance Pretty () where
  pp () = text "()"
  ppPrec _ = pp
```

```
instance Pretty Bool where
  pp b = text $ show b
  ppPrec _ = pp

instance Pretty Ordering where
  pp o = text $ show o
  ppPrec _ = pp

instance Pretty Int where
  ppPrec n x
    | n /= 0 && x < 0 = parens (text $ show x)
    | otherwise = text $ show x
  pp = ppPrec 0

instance Pretty Integer where
  ppPrec n x
    | n /= 0 && x < 0 = parens (text $ show x)
    | otherwise = text $ show x
  pp = ppPrec 0

instance Pretty Float where
  ppPrec n x
    | n /= 0 && x < 0 = parens (text $ show x)
    | otherwise = text $ show x
  pp = ppPrec 0

instance Pretty Double where
  ppPrec n x
    | n /= 0 && x < 0 = parens (text $ show x)
    | otherwise = text $ show x
  pp = ppPrec 0

instance Pretty Char where
  pp char = text $ show char
  ppPrec _ = pp
  -- instance Pretty String where , as below
  ppList str = text $ show str

-- doc ([1,3,7] :: [Int])
instance Pretty a => Pretty [a] where
  pp = ppList
  ppPrec _ = pp

instance (Pretty a, Pretty b) => Pretty (Map a b) where
  pp m =  group $ ”fromList” <-> pp (toList m)
  ppPrec _ = pp

instance Pretty a => Pretty (Maybe a) where
  ppPrec n Nothing = text ”Nothing”
  ppPrec n (Just x)
    | n /= 0 = parens s
    | otherwise = s
      where
        s = ”Just” <-> ppPrec 10 x
  pp = ppPrec 0

instance (Pretty a, Pretty b) => Pretty (Either a b) where
  ppPrec n (Left x)
    | n /= 0 = parens s
    | otherwise = s
      where
        s = ”Left” <-> ppPrec 10 x
  ppPrec n (Right x)
    | n /= 0 = parens s
    | otherwise = s
      where
        s = ”Right” <-> ppPrec 10 x
  pp = ppPrec 0

-- instances for the first few tuples

instance (Pretty a, Pretty b) => Pretty (a, b) where
```

```
  pp (a, b) = group (parens $ sep [pp a <> comma, pp b])
  ppPrec _ = pp

instance (Pretty a, Pretty b, Pretty c) => Pretty (a, b, c) where
  pp (a, b, c) = group (parens $ sep [pp a <> comma, pp b <> comma, pp c])
  ppPrec _ = pp

instance (Pretty a, Pretty b, Pretty c, Pretty d) => Pretty (a, b, c, d) where
  pp (a, b, c, d) = group (parens $ sep [pp a <> comma, pp b <> comma, pp c <> comma, pp d])
  ppPrec _ = pp

instance (Pretty a, Pretty b, Pretty c, Pretty d, Pretty e) => Pretty (a, b, c, d, e) where
  pp (a, b, c, d, e) = group (parens $ sep [pp a <> comma, pp b <> comma,
                                            pp c <> comma, pp d <> comma, pp e])
  ppPrec _ = pp

instance (Pretty a, Pretty b, Pretty c, Pretty d, Pretty e, Pretty f) => Pretty (a, b, c, d, e, f) where
  pp (a, b, c, d, e, f) = group (parens $ sep [pp a <> comma, pp b <> comma,
                                               pp c <> comma, pp d <> comma,
                                               pp e <> comma, pp f])
  ppPrec _ = pp

instance (Pretty a, Pretty b, Pretty c, Pretty d, Pretty e, Pretty f, Pretty g)
        => Pretty (a, b, c, d, e, f, g) where
  pp (a, b, c, d, e, f, g) = group (parens $ sep [pp a <> comma, pp b <> comma, pp c <> comma,
                                                  pp d <> comma, pp e <> comma, pp f <> comma,
                                                  pp g])
  ppPrec _ = pp

instance (Pretty a, Pretty b, Pretty c, Pretty d, Pretty e, Pretty f, Pretty g, Pretty h)
        => Pretty (a, b, c, d, e, f, g, h) where
  pp (a, b, c, d, e, f, g, h) = group (parens $ sep [pp a <> comma, pp b <> comma, pp c <> comma,
                                                     pp d <> comma, pp e <> comma, pp f <> comma,
                                                     pp g <> comma, pp h])
  ppPrec _ = pp

instance (Pretty a, Pretty b, Pretty c, Pretty d, Pretty e, Pretty f, Pretty g, Pretty h, Pretty i)
        => Pretty (a, b, c, d, e, f, g, h, i) where
  pp (a, b, c, d, e, f, g, h, i) = group (parens $ sep [pp a <> comma, pp b <> comma, pp c <> comma,
                                                        pp d <> comma, pp e <> comma, pp f <> comma,
                                                        pp g <> comma, pp h <> comma, pp i])
  ppPrec _ = pp

instance (Pretty a, Pretty b, Pretty c, Pretty d, Pretty e, Pretty f, Pretty g, Pretty h, Pretty i,
         Pretty j)
        => Pretty (a, b, c, d, e, f, g, h, i, j) where
  pp (a, b, c, d, e, f, g, h, i, j)
      = group (parens $ sep [pp a <> comma, pp b <> comma, pp c <> comma, pp d <> comma,
                             pp e <> comma, pp f <> comma, pp g <> comma, pp h <> comma,
                             pp i <> comma, pp j])
  ppPrec _ = pp

instance (Pretty a, Pretty b, Pretty c, Pretty d, Pretty e, Pretty f, Pretty g, Pretty h, Pretty i,
         Pretty j, Pretty k)
        => Pretty (a, b, c, d, e, f, g, h, i, j, k) where
  pp (a, b, c, d, e, f, g, h, i, j, k)
      = group (parens $ sep [pp a <> comma, pp b <> comma, pp c <> comma, pp d <> comma,
                             pp e <> comma, pp f <> comma, pp g <> comma, pp h <> comma,
                             pp i <> comma, pp j <> comma, pp k])
  ppPrec _ = pp

instance (Pretty a, Pretty b, Pretty c, Pretty d, Pretty e, Pretty f, Pretty g, Pretty h, Pretty i,
         Pretty j, Pretty k, Pretty l)
        => Pretty (a, b, c, d, e, f, g, h, i, j, k, l) where
  pp (a, b, c, d, e, f, g, h, i, j, k, l)
      = group (parens $ sep [pp a <> comma, pp b <> comma, pp c <> comma, pp d <> comma,
                             pp e <> comma, pp f <> comma, pp g <> comma, pp h <> comma,
                             pp i <> comma, pp j <> comma, pp k <> comma, pp l])
  ppPrec _ = pp

--------------------------------------------------------------------------------
-- Support code for Pretty
```

```haskell
--------------------------------------------------------------------

-- helper function that get the value from char type to DOC
char :: Char -> DOC
char chr = text [chr]

-- helper functions for instance Pretty Pair and List
-- generate n spaces
text' :: Int -> String
text' n | n == 0 = ""
        | otherwise = "␣" ++ text' (n - 1)

-- helper function for docList
pp' :: Pretty a => a -> DOC
pp' x = nest indent (line <> pp x)

-- helper function for reproducing the [DOC] to DOC
rep :: [DOC] -> DOC
rep []     = nil
rep (x:xs) = group $ Prelude.foldl (<>) nil (x:xs)

sep :: [DOC] -> DOC
sep []     = nil
sep (x:xs) = nest indent x
             <> foldr1 (\l r -> l <> nil <> r) (map (\x -> nest indent (line <> x)) xs)

x <-> y = text x <+> nest (length x + 1) y

pretty :: (Doc -> String) -> Int -> DOC -> String
pretty f w x  = f (best w 0 x)

pshow :: Pretty a => (Doc -> String) -> Int -> a -> String
pshow f w x = pretty f w (pp x <> line)

pprinter :: Pretty a => Int -> a -> IO()
pprinter w x = putStr (pshow layout w x)

--------------------------------------------------------------------
-- Pretty Printer
--------------------------------------------------------------------

data Mode = ManyLineMode | OneLineMode

-- | A rendering style
data Style = Style { mode    :: Mode,  -- ^ The redering mode
                     lineLen :: Int    -- ^ Length of line
                   }

styleMode :: Style -> Mode
styleMode (Style mode length) = mode

styleLen  :: Style -> Int
styleLen (Style mode length) = length

-- | The default 'Style'
style :: Style
style = Style {mode = ManyLineMode, lineLen = 40}

render     :: Show a => Pretty a => a -> String
fullRender :: Show a => Pretty a =>
              Mode
              -> Int
              -> a
              -> String
fullRender ManyLineMode w x = pshow layout w x
fullRender OneLineMode w x = pshow oneLayout w x

-- use default style
render = fullRender (styleMode style) (styleLen style)

pprint :: Show a => Pretty a => a -> IO()
pprint x = putStr (render x)
```

```

pprintLen :: Show a => Pretty a => Int -> a -> IO()
pprintLen = pprinter

-- | The default Pretty Printer
pprintStyle :: Show a => Pretty a => Style -> a -> IO()
pprintStyle s x = putStr $ fullRender (styleMode s) (styleLen s) x
```

Listing A.1: Code of Generic Pretty Printer

Appendix B

Code of Tester for Pretty Printer

```haskell
{-# LANGUAGE DeriveGeneric #-}

module Tester where

import Text.PPrinter hiding (char, (<|>))
import Test.Hspec
import Test.QuickCheck

import Data.Map hiding (showTree, map, null)
import Data.List (null)
import Data.Char
import Control.Exception (evaluate)

import Control.Applicative hiding (many, (<|>))
import Text.Parsec
import Text.Parsec.String
import Text.Parsec.Expr
import Text.Parsec.Token
import Text.Parsec.Language

--------------------------------------------------------------------------------
-- Parser
--------------------------------------------------------------------------------

-- separator
sepParser :: Parser ()
sepParser = spaces >> char ’,’ >> spaces


-- literal string
stringParser :: Parser String
stringParser = do
              char ’\”’
␣␣␣␣␣␣␣␣␣␣␣␣␣␣manyTill␣(letter␣<|>␣digit␣<|>␣space)␣(char␣’\”’)␣<|>␣string␣”\\\””


numParser :: Parser String
numParser = many (char ’-’ <|> digit <|> char ’.’ <|> char ’e’)

strParser :: Parser String
strParser = stringParser <|> string ”True” <|> string ”False” <|> numParser

-- (String, Int)
pairParser :: Parser (String, String)
pairParser = do
            char ’(’
            a <- stringParser
            spaces
```

```
           char ’,’
           spaces
           b <- many1 digit
           char ’)’
           return (a, b)

-- list of pair
listParser :: Parser [(String, String)]
listParser = sepBy pairParser sepParser

listParser2’ :: Parser [String]
listParser2’ = sepBy strParser sepParser

listParser2 :: Parser [String]
listParser2 = do
            char ’[’
            e <- listParser2’
            char ’]’
            return e

mapParser :: Parser [(String, String)]
mapParser = do
          string ”fromList”
          spaces
          char ’[’
          e <- listParser
          char ’]’
          return e

tree1Parser :: Parser String
tree1Parser = (string ”Nod” >>
              spaces >> char ’(’ >> tree1Parser >>= \res1 -> char ’)’ >>
              spaces >> char ’(’ >> tree1Parser >>= \res2 -> char ’)’ >>
              spaces >> return (res1 ++ res2)) <|>
             (string ”Leaf” >> spaces >> strParser)

tree2Parser :: Parser String
tree2Parser =(char ’(’ >> tree2Parser >>= \res1 -> char ’)’ >> spaces >> string ”:^:” >>
              spaces >> char ’(’ >> tree2Parser >>= \res2 -> char ’)’ >>
              return (res1 ++ res2)) <|>
             (string ”Leaf2” >> spaces >> strParser)

infixParser :: Parser String
infixParser = numParser >>= \res1 -> spaces >> string ”:**:” >> spaces >>
              strParser >>= \res2 -> return (res1 ++ res2)


recParser :: Parser String
recParser = string ”Person” >> spaces >>
            string ”{firstName␣=␣” >> strParser >>= \res1 -> char ’,’ >> spaces >>
            string ”lastName␣=␣” >> strParser >>= \res2 -> char ’,’ >> spaces >>
            string ”age␣=␣” >> strParser >>= \res3 -> char ’,’ >> spaces >>
            string ”height␣=␣” >> strParser >>= \res4 -> char ’,’ >> spaces >>
            string ”addr␣=␣” >> strParser >>= \res5 -> char ’,’ >> spaces >>
            string ”occup␣=␣” >> strParser >>= \res6 -> char ’,’ >> spaces >>
            string ”gender␣=␣” >> strParser >>= \res7 -> char ’,’ >> spaces >>
            string ”nationality␣=␣” >> strParser >>= \res8 ->
            return (res1 ++ res2 ++ res3 ++ res4 ++ res5 ++ res6 ++ res7 ++ res8)


testRec = parse recParser ””

testInfix = parse infixParser ””

-- the interface of testing Trees
testTree1 = parse tree1Parser ””

testTree2 = parse tree2Parser ””

-- the interface of testing Map
testMap = parse mapParser ””
```

```haskell
testList = parse listParser ""

--------------------------------------------------------------------
-- Utility functions
--------------------------------------------------------------------

-- omit the white space and new line
-- For example, omitNewline "a b c \n" will return "abc"
omitNewline :: String -> String
omitNewline [] = []
omitNewline (x:xs) = Prelude.filter (/= '\n') (x:xs)

omitWhite :: String -> String
omitWhite [] = []
omitWhite (x:xs) = Prelude.filter (\x -> x /= ' ' && x /= '\n') (x:xs)

pShow :: Pretty a => Int -> a -> String
pShow w x = pretty layout w (pp x <> line)

--------------------------------------------------------------------
-- Main function
--------------------------------------------------------------------

main :: IO ()
main = hspec $ do

--------------------------------------------------------------------
-- Primitive data types
--------------------------------------------------------------------

  describe "Primitive␣Data␣Type␣Testing" $ do

    -- Unit

    it "Unit" $ property $
      \x -> omitNewline (pShow x ()) `shouldBe` show ()

    -- Number test

    it "Num␣:␣test␣positive␣integer" $
      omitNewline (pShow 10 (10 ::Int)) `shouldBe` show 10

    it "Num␣:␣test␣negative␣integer" $
      omitNewline (pShow 10 ((-999) ::Int)) `shouldBe` show (-999)

    it "Num␣:␣test␣big␣integer" $
      omitNewline (pShow 10 (999999999999999999999999 ::Integer)) `shouldBe`
                  show 999999999999999999999999

    it "Num␣:␣random␣integer␣test" $ property $
       \x y -> omitNewline (pShow y (x :: Int)) `shouldBe` show x

    it "Num␣:␣random␣float␣test" $ property $
       \x y -> omitNewline (pShow y (x :: Float)) `shouldBe` show x

    it "Num␣:␣random␣double␣test" $ property $
       \x y -> omitNewline (pShow y (x :: Double)) `shouldBe` show x

    -- Char and String test

    it "Char␣:␣random␣character" $ property $
       \x y -> init (pShow y (x :: Char)) `shouldBe` show x

    it "String␣:␣random␣string" $ property $
       \x y -> init (pShow y (x :: String)) `shouldBe` show x

    -- Bool test

    it "Bool␣:␣random␣boolean" $ property $
       \x y -> omitNewline (pShow y (x :: Bool)) `shouldBe` show x

    -- Map test
```

```haskell

    it "Map  : test 1" $ property $
       \x -> testMap (pShow x ml) `shouldBe` testMap (show ml)

    it "Map  : test 2" $ property $
       \x -> testMap (pShow x ml) `shouldBe` testMap (show ml)

    it "Map  : test 3" $ property $
       \x -> testMap (pShow x ml) `shouldBe` testMap (show ml)

    -- Ordering
    it "Ordering : EQ | LT | GT" $ property $
       \x y -> omitNewline (pShow x (y :: Ordering)) `shouldBe` show y

    -- Maybe
    it "Maybe : Nothing" $ property $
       \x -> omitNewline (pShow x (Nothing :: Maybe Int)) `shouldBe`
             show (Nothing :: Maybe Int)

    it "Maybe : Maybe Bool" $ property $
       \x -> omitNewline (pShow x (Just True :: Maybe Bool)) `shouldBe`
             omitNewline (show (Just True :: Maybe Bool))

    it "Maybe : Maybe Int" $ property $
       \x y -> omitNewline (pShow x (Just y :: Maybe Int)) `shouldBe`
               omitNewline (show (Just y :: Maybe Int))

    it "Maybe : Maybe Char" $ property $
       \x y -> omitNewline (pShow x (Just y :: Maybe Char)) `shouldBe`
               omitNewline (show (Just y :: Maybe Char))

    it "Maybe : Maybe String" $ property $
       \x y -> omitNewline (pShow x (Just y :: Maybe String)) `shouldBe`
               omitNewline (show (Just y :: Maybe String))

    it "Maybe : Maybe Float" $ property $
       \x y -> omitNewline (pShow x (Just y :: Maybe Float)) `shouldBe`
               omitNewline (show (Just y :: Maybe Float))

    it "Maybe : Maybe Double" $ property $
       \x y -> omitNewline (pShow x (Just y :: Maybe Double)) `shouldBe`
               omitNewline (show (Just y :: Maybe Double))

    it "Maybe : Maybe Ordering" $ property $
       \x y -> omitNewline (pShow x (Just y :: Maybe Ordering)) `shouldBe`
               omitNewline (show (Just y :: Maybe Ordering))

    -- Either

    it "Either : Left Int" $ property $
       \x y -> omitNewline (pShow x (Left y :: Either Int Int)) `shouldBe`
               omitNewline (show (Left y :: Either Int Int))

    it "Either : Right Integer" $ property $
       \x y -> omitNewline (pShow x (Right y :: Either Int Integer)) `shouldBe`
               omitNewline (show (Right y :: Either Int Integer))

    it "Either : Left (Maybe Int)" $ property $
       \x y -> omitNewline (pShow x (Left y :: Either (Maybe Int) Int)) `shouldBe`
               omitNewline (show (Left y :: Either (Maybe Int) Int))

    it "Either : Right Bool" $ property $
       \x y -> omitNewline (pShow x (Right y :: Either Int Bool)) `shouldBe`
               omitNewline (show (Right y :: Either Int Bool))

    it "Either : Left Char" $ property $
       \x y -> omitNewline (pShow x (Left y :: Either Char Int)) `shouldBe`
               omitNewline (show (Left y :: Either Char Int))

    it "Either : Right String" $ property $
       \x y -> omitNewline (pShow x (Right y :: Either Int String)) `shouldBe`
               omitNewline (show (Right y :: Either Int String))
```

```

    it "Either␣:␣Left␣Float" $ property $
      \x y -> omitNewline (pShow x (Left y :: Either Float Int)) `shouldBe`
              omitNewline (show (Left y :: Either Float Int))

    it "Either␣:␣Right␣Double" $ property $
      \x y -> omitNewline (pShow x (Right y :: Either Int Double)) `shouldBe`
              omitNewline (show (Right y :: Either Int Double))

    -- Pair

    it "Pair␣:␣(a,␣b)" $ property $
      \x y -> omitWhite (pShow x (y :: (Int, Int))) `shouldBe`
               omitWhite (show (y :: (Int, Int)))

    it   Triple : (a, b, c)"␣$␣property␣$
␣␣␣␣␣␣␣\x␣y␣->␣omitWhite␣(pShow␣x␣(y␣::␣(Int,␣Int,␣Bool)))␣`shouldBe`
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣omitWhite␣(show␣(y␣::␣(Int,␣Int,␣Bool)))

␣␣␣␣it␣  Tuple␣:␣(a,␣b,␣c,␣d)" $ property $
      \x y -> omitWhite (pShow x (y :: (Int, Int, Bool, Float))) `shouldBe`
               omitWhite (show (y :: (Int, Int, Bool, Float)))

    it "Tuple␣:␣(a,␣b,␣c,␣d,␣e)" $ property $
      \x y -> omitWhite (pShow x (y :: (Int, Int, Bool, Double, Char))) `shouldBe`
               omitWhite (show (y :: (Int, Int, Bool, Double, Char)))

    it "Tuple␣:␣(a,␣b,␣c,␣d,␣e,␣f)" $ property $
      \x   -> omitWhite (pShow x ((True, False, True, False, True, False)
                                   :: (Bool, Bool, Bool,Bool, Bool, Bool)))
              `shouldBe`
               omitWhite (show  ((True, False, True, False, True, False)
                                   :: (Bool, Bool, Bool,Bool, Bool, Bool)))

    it "Tuple␣:␣(a,␣b,␣c,␣d,␣e,␣f,␣g)" $ property $
      \x   -> omitWhite (pShow x ((True, False, True, False, True, False, True, False)
                                   :: (Bool, Bool, Bool,Bool, Bool, Bool,Bool, Bool)))
              `shouldBe`
               omitWhite (show  ((True, False, True, False, True, False, True, False)
                                   :: (Bool, Bool, Bool,Bool, Bool, Bool,Bool, Bool)))

    it "Tuple␣:␣(a,␣b,␣c,␣d,␣e,␣f,␣g,␣h)" $ property $
      \x   -> omitWhite (pShow x ((True, False, True, False, True, False, True, False)
                                   :: (Bool, Bool, Bool,Bool, Bool, Bool,Bool, Bool)))
              `shouldBe`
               omitWhite (show  ((True, False, True, False, True, False, True, False)
                                   :: (Bool, Bool, Bool,Bool, Bool, Bool,Bool, Bool)))

    it "Tuple␣:␣(a,␣b,␣c,␣d,␣e,␣f,␣g,␣h,␣i)" $ property $
      \x   -> omitWhite (pShow x ((True, False, True, False, True, False, True, False, True)
                                   :: (Bool, Bool, Bool,Bool, Bool, Bool,Bool, Bool, Bool)))
              `shouldBe`
               omitWhite (show  ((True, False, True, False, True, False, True, False, True)
                                   :: (Bool, Bool, Bool,Bool, Bool, Bool,Bool, Bool, Bool)))

    it "Tuple␣:␣(a,␣b,␣c,␣d,␣e,␣f,␣g,␣h,␣i,␣j)" $ property $
      \x   -> omitWhite (pShow x ((True, False, True, False, True, False, True, False, True, False)
                                   :: (Bool, Bool, Bool,Bool, Bool, Bool,Bool, Bool, Bool,Bool)))
              `shouldBe`
               omitWhite (show  ((True, False, True, False, True, False, True, False, True, False)
                                   :: (Bool, Bool, Bool,Bool, Bool, Bool,Bool, Bool, Bool,Bool)))

    it "Tuple␣:␣(a,␣b,␣c,␣d,␣e,␣f,␣g,␣h,␣i,␣j,␣k)" $ property $
      \x   -> omitWhite (pShow x ((True, False, True, False, True, False, True, False, True, False, True)
                                   :: (Bool, Bool, Bool,Bool, Bool, Bool,Bool, Bool, Bool,Bool, Bool)))
              `shouldBe`
               omitWhite (show  ((True, False, True, False, True, False, True, False, True, False, True)
                                   :: (Bool, Bool, Bool,Bool, Bool, Bool,Bool, Bool, Bool,Bool, Bool)))

    it "Tuple␣:␣(a,␣b,␣c,␣d,␣e,␣f,␣g,␣h,␣i,␣j,␣k,␣l)" $ property $
      \x   -> omitWhite (pShow x ((True, False, True, False, True, False, True, False, True, False, True, False)
                                   :: (Bool, Bool, Bool,Bool, Bool, Bool,Bool, Bool, Bool,Bool, Bool, Bool)))
```

```
              `shouldBe`
               omitWhite (show  ((True, False, True, False, True, False, True, False, True, False, True, False)
                                 :: (Bool, Bool, Bool,Bool, Bool, Bool,Bool, Bool, Bool,Bool, Bool, Bool)))


    -- List
    it "List␣:␣[Int]" $ property $
      \x y -> testList (pShow x (y :: [Int])) `shouldBe` testList (show y)

    it "List␣:␣[Float]" $ property $
      \x y -> testList (pShow x ((0 : y) :: [Float])) `shouldBe` testList (show (0 : y))

    it "List␣:␣[Double]" $ property $
      \x y -> testList (pShow x ((1.0 : y) :: [Double])) `shouldBe` testList (show (1.0 : y))

    it "List␣:␣String" $ property $
      \x y -> testList (pShow x ((' ' : y) :: String)) `shouldBe` testList (show (' ' : y))

    it "List␣:␣[String]" $ property $
      \x y -> testList (pShow x (("␣" : y) :: [String])) `shouldBe` testList (show ("␣" : y))

    it "List␣:␣[Bool]" $ property $
      \x y -> testList (pShow x ((True : y) :: [Bool])) `shouldBe` testList (show (True : y))

{- It does not work

    it "List : Infinite [Int]" $ property $
      \x y -> (omitNewline $ pShow x ([y ..] :: [Int])) `shouldBe` show [y ..]

-}

------------------------------------------------------------------
-- Algebraic data type
------------------------------------------------------------------

  describe "\nAlgebraic␣Data␣Type␣Testing" $ do

    -- Record

    it "Record␣:␣test␣1" $ property $
      \x -> testRec (pShow x pers) `shouldBe`
            testRec (show pers)

    -- Tree

    it "Trees␣Int" $ property $
      \x -> testTree1 (pShow x tree1) `shouldBe`
            testTree1 (show tree1)

    -- Infix notation

    it "Infix␣style␣:␣data␣Foo␣a␣b␣=␣a␣:**:␣b␣" $ property $
      \x -> testInfix (pShow x test1) `shouldBe`
            testInfix (show test1)


------------------------------------------------------------------
-- Data types from Repository of Haskell Programs
------------------------------------------------------------------

    it "Tree␣example␣from␣GHC.Show" $ property $
      \x -> testTree2 (pShow x tree2) `shouldBe`
            testTree2 (show tree2)


------------------------------------------------------------------
-- Definition for algebraic data type testing
------------------------------------------------------------------

-- Maps

m1 :: Map String Int
```

```
m1 = fromList [(”ad”, 123), (”b”, 234), (”c”, 345), (”d”, 45)]

m2 :: Map String Int
m2 = singleton ”abc” 123

— tree example

data Tree = Node String [Tree] deriving (Generic, Show)

instance Pretty (Tree)

tree                 = Node ”aaa” [
                            Node ”bbbbb” [
                               Node ”ccc” [],
                               Node ”dd” []
                               ],
                            Node ”eee” [],
                            Node ”ffff” [
                              Node ”gg” [],
                              Node ”hhh” [],
                              Node ”ii” []
                              ]
                            ]

data Foo a b = a :**: b deriving (Generic, Show)

data Trees a = Leaf a | Nod (Trees a) (Trees a) deriving (Generic, Show)

data Person = Person { firstName :: String,
                       lastName :: String,
                       age :: Int,
                       height :: Float,
                       addr :: String,
                       occup :: String,
                       gender :: Bool,
                       nationality :: String
                     } deriving (Generic, Show)

instance (Pretty a, Pretty b) => Pretty (Foo a b)

instance (Pretty a) => Pretty (Trees a)

instance Pretty (Person)

pers = Person ”Arthur” ”Lee” 20 (−1.75) ”Edinburgh␣UK” ”Student” True ”Japan”

test1 = (2 :: Float) :**: ”cc”

tree1 :: Trees Int
tree1 = Nod (Nod (Leaf 333)
                 (Leaf 5555))
            (Nod (Nod(Nod(Leaf 8888)
                         (Leaf 5757))
                  (Leaf 14414))
             (Leaf 32777))

— Example from GHC.Show
infixr 5 :^:
data Tree2 a =  Leaf2 a  |  Tree2 a :^: Tree2 a deriving (Generic, Show)

instance (Pretty a) => Pretty (Tree2 a)
tree2 :: Tree2 Int
tree2 = (Leaf2 89) :^: ((((Leaf2 1324324) :^: (Leaf2 1341)) :^: (Leaf2 (−22))) :^: (Leaf2 99))
```

Listing B.1: Code of Tester for Pretty Printer

Bibliography

Binstock, A. (2011). Interview with Scala's Martin Odersky. `http://www.drdobbs.com/architecture-and-design/interview-with-scalas-martin-odersky/231001802`. Accessed: 2016-08-14.

Done, C. (2014). Dijkstra on Haskell and Java. `http://chrisdone.com/posts/dijkstra-haskell-java`. Accessed: 2016-08-14.

Hudak, Paul, J. H. S. P. J. P. W. (2007). A history of Haskell: being lazy with class. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, pages 12–1. ACM.

Hudak, Paul, J. P. J. F. (2000). A gentle introduction to Haskell. `https://www.haskell.org/tutorial/`. Accessed: 2016-08-14.

Hughes, J. (1995). The design of a pretty-printing library. In *International School on Advanced Functional Programming*, pages 53–96. Springer Berlin Heidelberg.

Hutton, G. (2007). *Programming in Haskell*. Cambridge University Press.

Koen Classen, J. H. (2006). QuickCheck: Automatic Testing of Haskell Programs. `https://hackage.haskell.org/package/QuickCheck`. Accessed: 2016-08-18.

Leather, S. (2011). GHC.Generics. `https://wiki.haskell.org/GHC.Generics`. Accessed: 2016-08-14.

Leather, S. (2012). Generics. `https://wiki.haskell.org/Generics`. Accessed: 2016-08-14.

Leijen, D. (2006). The parsec: Monadic parser combinators. `https://hackage.haskell.org/package/parsec-2.0`. Accessed: 2016-08-18.

Magalhes, Jos Pedro, A. D. J. J. A. L. (2010). A generic deriving mechanism for Haskell. *ACM*, 45(11):37–48.

Norell, U. (2007). *Towards a practical programming language based on dependent type theory*. PhD thesis, Chalmers University of Technology, Sweden.

Ranca, R. (2012). GenericPretty: A generic, derivable, Haskell pretty printer. `https://hackage.haskell.org/package/GenericPretty`. Accessed: 2016-08-14.

Ross Paterson, Herbert Valerio Riedel, I. L. (2009). Text.Show. `https://hackage.haskell.org/package/base-4.9.0.0/docs/Text-Show.html`. Accessed: 2016-08-18.

Spangler, T. (2011). Hspec: A Testing Framework for Haskell. `http://hspec.github.io`. Accessed: 2016-08-18.

Terei, D. (2001). Pretty-printing library. `https://hackage.haskell.org/package/pretty-1.1.3.4/docs/Text-PrettyPrint.html`. Accessed: 2016-08-14.

Wadler, P. (2003). A prettier printer. In *The Fun of Programming (Cornerstones of Computing)*, pages 223–243, UK. Palgrave Macmillan.