A Prettier Prelude

Matthew Marsland

4th Year Project Report
Computer Science
School of Informatics
University of Edinburgh
2021
Abstract

Two systems are presented, aimed at improving the experience of Haskell for Inf1A, a first-year Informatics course with 400 students. The first is the implementation of a library developed by Razvan Ranca called GenericPretty, to pretty-print complex data types in Haskell tutorials. GenericPretty was evaluated by students in the 2020 Inf1A course. The second is a modified standard library for Haskell called EdPrelude which has default pretty-printing, defaulted function types and only supports arbitrary-size integers. EdPrelude is packaged for ease of use by beginner Haskell students, and was tested on all nine Inf1A tutorial exercises.
Acknowledgements

Acknowledgements go here.
Table of Contents

1 Introduction 1

2 Background 4
  2.1 Haskell 4
     2.1.1 Values and Types 4
     2.1.2 Functions 5
     2.1.3 User-Defined Types 6
     2.1.4 Programming in Haskell and the REPL 7
     2.1.5 Further 8
     2.2 Pretty-Printing 8
        2.2.1 Hughes-PJ Pretty-Printing Operators 9
     2.3 Generic Programming 11
        2.3.1 Type Representation 13
        2.3.2 Example: Generic Equality 15
        2.3.3 Example: Generic Printing 16

3 Pretty-Printing For First-Years 18
  3.1 Trees in Tutorials 18
  3.2 GenericPretty For Commands 20
  3.3 Default Pretty-Printing 21
  3.4 Parenthetical Subtleties 22
  3.5 Results 26

4 EdPrelude 27
  4.1 Alternative Preludes 27
  4.2 Pretty-Printing By Default 28
  4.3 Removing Fixed-Size Integers 31
  4.4 Removing (Or Defaulting) Typeclasses 32
  4.5 Shortening Compiler Flags 33
     4.5.1 -interactive-print 34
     4.5.2 -ghci-script, -ignore-dot-ghci 34
     4.5.3 -i 34
     4.5.4 -XNoImplicitPrelude 34
     4.5.5 -XDeriveGeneric, -XDeriveAnyClass 35

5 Future Work 36
Parentheses, like wasps, make nests I dread. Printing fundamental types such as integers, strings, or characters which have self-evident and straightforward representations is easy. But complex user-defined types, such as the following datatype for manipulating images ...

data Picture
    = Img Image
    | Above Picture Picture
    | Beside Picture Picture
    | Over Picture Picture
    | FlipH Picture
    | FlipV Picture
    | Invert Picture

... are not so straightforward. Consider visually parsing even a relatively small example of this Picture type:

Main*> print emptyRow
Beside (Beside (Beside (Img (Image WhiteTile (50,50))) (Img (Image BlackTile (50,50)))) (Beside (Img (Image WhiteTile (50,50))) (Img (Image BlackTile (50,50)))) (Beside (Beside (Img (Image WhiteTile (50,50))) (Img (Image BlackTile (50,50)))) (Beside (Img (Image WhiteTile (50,50))) (Img (Image BlackTile (50,50))))

Parentheses fill the programmer’s field of view well past clear understanding.

Defining an intelligible print format for a given data type (a custom pretty-printer) is the common solution. For instance, we could create a printing function to render a Picture prettily, making use of the Text.PrettyPrint library for structuring text in Haskell:

pPrint :: Picture -> Doc
pPrint (Img a)     = parens (text "Img"     <+> parens (text (show a)))
pPrint (Above a b) = parens (text "Above"  <+> sep [nest 1 (pPrint a),
                                           nest 1 (pPrint b)])
Chapter 1. Introduction

pPrint (Beside a b) = parens (text "Beside" <+> sep [nest 1 (pPrint a),
                                          nest 1 (pPrint b)])

pPrint (Over a b) = parens (text "Over" <+> sep [nest 1 (pPrint a),
                                           nest 1 (pPrint b)])

pPrint (FlipH a) = parens (text "FlipH" <+> parens (text (show a)))

pPrint (FlipV a) = parens (text "FlipV" <+> parens (text (show a)))

Now, when we pretty-print that same example:

Main*> pPrint emptyRow
(Beside (Beside (Beside (Img (Image WhiteTile (50,50)))
           (Img (Image BlackTile (50,50))))
       (Beside (Img (Image WhiteTile (50,50)))
            (Img (Image BlackTile (50,50))))
    (Beside (Beside (Img (Image WhiteTile (50,50)))
             (Img (Image BlackTile (50,50))))
      (Beside (Img (Image WhiteTile (50,50)))
           (Img (Image BlackTile (50,50))))
   (Beside (Img (Image WhiteTile (50,50)))
        (Img (Image BlackTile (50,50))))))

A custom pretty-printer solves the parenthetical infestation but replaces it with a programmatic one. Now, the programmer has to spend time learning the nuances of a pretty-printing library (Doc, parens, text, <+>, sep, and nest are not standard Haskell) and yet further valuable time and space writing their printer. Mistakes in design might cause the printing to obscure information, introducing further murkiness. Worse still, all of that work and risk is repeated all over again when either the type in question is modified or a new one is introduced! If the programmer makes even a minor change to Picture, such as renaming Beside to NextTo, the pretty-printer will cry out for maintenance.

And so we have a paradox: pretty-printers are written for clarity but writing them impairs it: in scooping the silt out of the water, we introduce ripples that continue to obstruct our view. Programmers must either put in the effort to produce a pretty printer and hope that their investment will reap rewards, or make do without.

In 2011, Razvan Ranca developed a pretty-printing library for Haskell [Ran18], based on a decade’s worth of research into Generic Programming [Hin00], [HJL06], [RJJ+08], [MDJL10]. His library simplifies the workflow of pretty-printing dramatically. In contrast to the bulky and case-specific pPrint above, a pretty-printer can now be declared like so:

data Picture
  = ...  
  deriving (Generic, Out)

Which automatically derives a pretty-printer for the datatype. This is a reduction to 10% of the manual printer’s line count, and furthermore, it’s future-proof. If the datatype changes, the derivation will update to match. The paradox unravels, the waters clear, pretty-printing is easy! And it seems users agree: since his project was released, the library has reached 18,000 downloads [Ran18]. For comparison, the Text.PrettyPrint library that GenericPretty is built on has received 36,000 downloads
in a similar timeframe.

At the University of Edinburgh, the first programming course students in the School of Informatics take is Inf1A: Introduction to Computation. Inf1A introduces all 400 of its students to Haskell, a functional programming language. A 2015/2016 study [AI16] found 14 other entry-level courses teaching Haskell throughout European universities. Haskell is far less common than C/C++ or Java, which together make up 746 of the 1019 subjects examined, however it is the most common functional programming language used in higher education.

I aim to integrate GenericPretty with Inf1A tutorials, to help students debug complex datatypes with ease. In addition, I will explore the process of using an alternative Prelude, the Haskell standard library, with the goal of improving the beginner Haskell experience. The primary alteration is to make the usage of GenericPretty seamless: by including it in the standard library, pretty-printing becomes available by default. General simplifications to the library will address student struggles with overly abstracted functions. Finally, to match the integer arithmetic that beginner programmers will be familiar with from mathematics, I restrict the available integer types to arbitrary-size integers and update all function definitions to comply with this restriction. The final product is an extensible module customized for educational Haskell in Inf1A.

The main contributions of this report are the following.

- I demonstrate the process I used to apply Ranca’s GenericPretty library to complex datatypes in an Inf1A student tutorial to aid debugging.
- I report on the success of GenericPretty’s use in the 2020 Inf1A course, evaluated by students who made use of the implementation, showing that the implementation improves readability.
- I explain the design of a modified standard library I created, EdPrelude, for teaching Haskell without fixed-size integer types or typeclasses that go unused.
- I evaluate EdPrelude for performance and correctness on all Inf1A student tutorials, showing that performance remains unaffected by the modifications I made and that the tutorials work as normal under the new library.

This project was pursued in close coordination with Prof. Philip Wadler, my supervisor, who is a course organizer for the Inf1A course in the University of Edinburgh, and was therefore my main point of contact for determining what areas of the course this work could improve.

The code for EdPrelude is available at:

https://github.com/MatthewMarmalade/e-prelude
Chapter 2

Background

2.1 Haskell

Love and language share at least one key trait: you never forget your first. For many students of Informatics at Edinburgh, including the author, the first experience in university will be of the functional programming language Haskell. Haskell sports a static type system, lazy evaluation, pattern-matching, higher-order functions, list comprehensions, and more. Starkly separate from imperative languages, it forces its users to (re-)think about programming in a fundamentally more abstract and mathematical way. As the introduction to a popular Haskell textbook states, "In purely functional programming you don’t tell the computer what to do as such but rather you tell it what stuff is." [Lip11]. Even students that go on to use only imperative languages for the rest of their academic and professional careers will be affected by their time with Haskell. If nothing else, they will realize that the assumptions made by less functional languages are decisions that the designers made, not merely 'the way things are', and begin to reexamine those decisions for themselves. Throughout this report, many examples in Haskell are given and several features of the language are referenced. This section explains some of those features in further detail; readers familiar with Haskell’s basics could easily skip to section 2.2. In addition, examples in the report will often be accompanied by parenthetical explanations of a more specific point if necessary.

Often, examples will contain comments to serve as inline annotations. Comments in Haskell are either preceded by two dashes like so:

```haskell
-- This is a single-line comment!
```

for single-line commenting, or enclosed within curly braces with dashes like so:

```haskell
{- This is a multi-line comment! -}
```

for multi-line bounded commenting.
2.1.1 Values and Types

Computation in Haskell is achieved via expressions which evaluate to values (1, ‘a’, 3.14159265, and "hello world" are all values). Every value has a type, such as Integer for an integer value, Char for a single character, String for a string of characters, [Integer] for a list of Integers ([_] denotes a list of anything inside the brackets; for instance [[Integer]] represents a list of lists of Integers). Values are given types through type expressions, denoted by the has type operator (::). We can say that a value has a certain type as follows:

\[
\begin{align*}
1 & :: \text{Integer} \\
'\text{a}' & :: \text{Char} \\
"\text{hello world}" & :: \text{String} \\
[1,2,3,4,5,6] & :: [\text{Integer}] \\
\end{align*}
\]

2.1.2 Functions

An expression can be any of the above atomic values, but they can also be functions that perform computations on values and return results. Functions in Haskell are more mathematical than in imperative counterparts. They return the same output for the same input, and no state is mutated. Paralleling this mathematical approach, Haskell functions are often defined as a sequence of equations:

\[
\text{double } x = x + x
\]

Echoing the textbook quotation from earlier, this function definition can be thought of as explaining to the computer what 'doubling' means. In addition, functions are values the same way 1 and ‘a’ are; as such they also have types. The type of the double function above can be expressed like so:

\[
\text{double } :: \text{Integer} \rightarrow \text{Integer}
\]

Which states that double takes as input a value of type Integer and outputs a value of type Integer.

2.1.2.1 Pattern-Matching

The following recursive function calculates the factorial of a number:

\[
\begin{align*}
\text{factorial } & :: \text{Integer} \rightarrow \text{Integer} \\
\text{factorial } 0 & = 1 \quad \text{-- } 0! = 1, \quad \text{base case} \\
\text{factorial } n & = n * \text{factorial } (n-1) \quad \text{-- } n! = n * (n-1)! \quad \text{recursive case}
\end{align*}
\]

The factorial function above is declared in terms of equations that are pattern matched. Each equation will be examined in order, with the first matching option evaluated. This allows functions to be described in a piecewise manner, and makes recursive functions like factorial straightforward to define in terms of cases. Pattern matching can be even more sophisticated, such as the following:

\[
\begin{align*}
\text{sum } & :: [\text{Integer}] \rightarrow \text{Integer} \\
\text{sum } [] & = 0 \quad \text{-- } [] \text{ refers to the empty list}
\end{align*}
\]
Chapter 2. Background

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

The \((x:xs)\) will match any non-empty list, additionally assigning the first element to \(x\) and the remainder of the list to \(xs\) (The \((:)\) operator means 'list cons', or prepending an element to the beginning of a list: \(1 : [2, 3] \Rightarrow [1, 2, 3]\)). This is especially useful for recursive functions like the above, as it allows us to inductively examine every element of the list until we reach the base case of the empty list.

### 2.1.2.2 Polymorphic Types and Type Variables

The types of functions can also be polymorphic; defined over a universally quantified type variable. A type variable is represented by a lower-case type in a type declaration. For instance, \(a\) refers to any type, \([a]\) refers to any list type. As an example usage in a type declaration, the identity function can be defined as follows:

\[
\text{id :: a -> a} \\
\text{id x = x}
\]

This is far preferable to defining a separate identity function for every possible type.

### 2.1.2.3 Higher-Order Functions

Finally, functions are themselves values. A higher-order function is one that takes another function as its argument, such as the following:

\[
\text{twice :: (Integer -> Integer) -> Integer -> Integer} \\
\text{twice f x = f (f x)}
\]

This function applies the given function of type \(\text{Integer -> Integer}\) to a given value of type \(\text{Integer}\) twice, returning the result. Clearly,

\[
\text{twice double 1 => double (double 1) => double 2 => 4}
\]

Making use of both type variables and higher-order functions, we can define the well-known \text{map} function like so:

\[
\text{map :: (a -> b) -> [a] -> [b]} \\
\text{map f [] = []} \\
\text{map f (x:xs) = f x : (map f xs)} -- \text{apply f to the first element, then cons the output to the recursive call.}
\]

This extremely powerful function applies a given function of type \((a -> b)\) to every element of a list of type \([a]\), naturally returning a list of type \([b]\). By example:

\[
\text{map double [1, 2, 3, 4] => [2, 4, 6, 8]} \\
\text{map sum [[1, 2], [3]] => [3, 3]}
\]

### 2.1.3 User-Defined Types

Users can define their own datatypes by specifying the name of the type and the constructors. For instance, the boolean type can be expressed like so:
data Bool = False | True

A value of type Bool must be either equal to True or to False. Alongside the constructors, we can also include other types, for instance:

data Coordinate = Coord Integer Integer

Here Coord is a data constructor, whereas the pair of Integers are types. So we have:

(Coord 5 14) :: Coordinate

Coord 5 14 is a value of type Coordinate, just as True is a value of type Bool.

Types can also be recursive; a classic example is a tree:

data Tree = Leaf | Node Tree Tree

This definition is very intuitive; it states that trees are either leaves or nodes with two subtrees. A value of type Tree might be any of the following:

Leaf :: Tree
Node Leaf Leaf :: Tree
Node (Node Leaf Leaf) (Node (Node Leaf Leaf) Leaf) :: Tree

Finally, types can also be polymorphic, and be parameterized over a type variable as well. For example, we can extend the Tree type from before to represent trees of any other type:

data Tree a = Leaf a | Node (Tree a) (Tree a)

Leaf 1 :: Tree Integer
Node (Leaf "Hello") (Leaf "World") :: Tree String

2.1.4 Programming in Haskell and the REPL

Haskell files can be identified by a .hs extension, and are organized into modules. Modules define and export a collection of functions and types that can be imported into other modules. A Haskell library generally refers to a published module; one makes use of the library through an import statement at the top of a Haskell module like so:

import Text.PrettyPrint.GenericPretty

Often, the most straightforward way to interact with a Haskell program is through a REPL, or Read-Evaluate-Print-Loop, sometimes referred to as an interpreter. Statements entered into the REPL are Read, the statement is Evaluated, the output is Printed, and then entire process Loops.

A common REPL used in the Inf1A course and used for this report is called ghci, based on GHC (the Glasgow Haskell Compiler) [GHCb].

This REPL can be invoked like so:

$ ghci
Chapter 2. Background

Some examples of ghci usage:

```
Prelude> x = 1 + 1
Prelude> x
2
Prelude> f y = y * y
Prelude> f 3
9
Prelude> f 4
16
Prelude> f x
4
```

2.1.5 Further

For further information on beginning Haskell, I recommend the following:

- The approach in the brief sketch above is covered in further depth and detail in Paul Hudak’s ‘A Gentle Introduction to Haskell’ [HF92]
- Miran Lipovača’s ‘Learn You A Haskell for a Great Good’ [Lip11] is an excellent informal web-based textbook aimed at users with some programming experience. It has been used as a course textbook for Inf1A.
- Simon Thompson’s ‘Haskell: The Craft of Functional Programming’ [Tho11] is a more formal Haskell textbook. It has been used as a course textbook for Inf1A.

If the reader is interested in vaulting out of the stands and onto the Haskell playing field, they can do so by downloading the Haskell Platform.

2.2 Pretty-Printing

This report refers to pretty-printing as the automated addition of whitespace to a text representation of a value in order to reflect logical layout through visual layout. It does not consider coloring systems or the addition/removal of any non-whitespace characters.

‘Pretty’ code has become an integral part of programming. Many development environments will automatically format loops, conditionals, functions, and classes into neat blocks with scope conveyed visually. Reading XML formats with indentation to keep track of the nodes is a godsend. And Python literally relies on the programmer formatting their code prettily as meaningful syntax!

The specific pretty-printing application of interest to this project is outputting structured data generated and manipulated in the Haskell language. A common pretty-printing library used for this purpose by the Haskell community is the Hughes-PJ library [hug], which arose from a design by John Hughes as an example of an algebraic approach to constructing a library [Hug95]. The Hughes-PJ library constructs values of
type Doc, which can be rendered as a formatted string of text. These Docs can be composed together vertically and horizontally, capturing newlines and indentation. This library does not enforce how Docs should be constructed for values of given types, but it gives programmers consistent tools to create custom pretty-printers and a consistent rendering of their outputs.

Ranca’s GenericPretty library, when given a datatype, derives a pretty-printer based on the Hughes-PJ library. It automatically derives instances of the methods doc and docPrec, which coerce a value of the given datatype to a pretty-printable Doc [Ran18]. The fiendishly clever area of research underpinning this magic trick is Generic Programming, discussed in more detail in section 2.3.

As GenericPretty derives pretty-printers based on the Hughes-PJ library, a description of the basic operators of the library is now given with accompanying examples. Readers familiar with this library can skip to section 2.3.

### 2.2.1 Hughes-PJ Pretty-Printing Operators

First, we have atomic Docs:

- `empty :: Doc
  -- The empty document, the identity for document composition operators.

- `char :: Char \rightarrow Doc
  -- Creates a Doc containing a literal character.

- `text :: String \rightarrow Doc
  -- Creates a Doc containing a literal string.

Then, we have operators for augmenting existing Docs:

- `parens :: Doc \rightarrow Doc
  -- Wraps a document in parentheses. Applying parentheses in this manner ensures an uneven number of parentheses is not introduced.

- `nest :: Int \rightarrow Doc \rightarrow Doc
  -- Indents a document by the given amount.

Finally, we have operators for combining multiple Docs together in meaningful ways:

- `{<>() :: Doc \rightarrow Doc \rightarrow Doc
  -- Infix operator that concatenates two documents together horizontally.

- `{<+> () :: Doc \rightarrow Doc \rightarrow Doc
  -- Identical to `{<>() with an added space between the documents.

- `hsep :: [Doc] \rightarrow Doc
  -- List version of `{<+>()}. For a list of docs [d1, d2, ... , dn], returns d1 <+> d2 <+> ... <+> dn.
Chapter 2. Background

($$) :: Doc -> Doc -> Doc
-- Infix operator that concatenates two documents together vertically.

vcat :: [Doc] -> Doc
-- List version of ($$). For a list of docs [d1, d2, ... , dn], returns
d1 $$ d2 $$ ... $$ dn.

sep :: [Doc] -> Doc
-- Intelligently chooses between hsep or vcat. This maintains horizontal
composition where possible but will break lines if necessary to meet
line length restrictions.

Examples of usage are presented below. A function from the library is presented, and
what follows on the next line is what is printed.

empty =>

car 'a' =>
a

text "hello" =>
hello world

parens (car 'a') =>
(a)

nest 1 (text "hello") =>
  hello

(car 'a') <> (car 'b') =>
ab

(text "hello") <+> (text "world") =>
hello world

sep [text "hello", nest 1 (text "world")]} => --line length unrestricted
hello world

sep [text "hello", text "world"] => --line length <5 chars
hello
  world
2.3 Generic Programming

The general goal of Generic Programming is to move from a specific implementation of an operation to a general one by using abstractions to capture the semantics rather than the minutiae. Gibbons [Gib06] classifies interpretations of 'genericity' based on what is parameterized, i.e. what is not statically defined. For instance, 'genericity by value' will be familiar to anyone who has used a function argument before:

\[
\text{square } x = x \times x
\]

'Genericity by type' is also familiar territory. Rather than write out a length function manually for every list type, we can do the following:

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

Where \(a\) is a type variable ranging over every type. Give length a list of anything, and since it’s defined for all \(a\), it can tell you how many items are in that list.

However, Gibbons describes one further step, that of 'datatype-generic' programming. In the case of length, all we care about is that \(a\) is something - it’s as if we ignore everything but its existence. But for our pretty-printer, we can’t ignore the shape of the data we are dealing with. A tree should print differently to a list, even if they contain the same number of distinct elements.

Haskell already has a system in place for taking shape into account: the deriving mechanism, which can automatically infer instance methods for a datatype to become a member of a typeclass. A typeclass is a constraint on a datatype that requires certain methods to have implementations [WB89]. For instance, the \texttt{Eq} typeclass is concerned with equality. For a datatype to be a member of \texttt{Eq}, it has to have equality defined over values of its type. The example datatype \texttt{AB} below could become an instance of \texttt{Eq} like so:

\[
\text{data AB = A | B}
\]

\[
\text{instance Eq AB where}
\]
\[
\text{  (==) :: AB -> AB -> Bool}
\]
\[
\text{  (==) A A = True}
\]
\[
\text{  (==) B B = True}
\]
\[
\text{  (==) _ _ = False}
\]

Now, values of type \texttt{AB} can be compared for equality using \((==)\). The operator \((==)\) is datatype-generic, because equality comparison is going to care about how the data being compared is structured. But this datatype-generic operator is not generically programmed; we supplied the instance of \texttt{Eq} for our custom type \texttt{AB}. This is where the deriving mechanism comes in! We can specify typeclasses that we want our datatype to be an instance of, and the compiler will generate the requested instance declarations automatically:

\[
\text{data AB = A | B deriving (Eq)}
\]
And now we have \texttt{Eq AB} without needing to write any further code. We can even go a step further and derive a string representation of our data type:

\begin{verbatim}
data AB = A | B deriving (Eq, Show)
\end{verbatim}

Which will automatically derive the following instance of \texttt{Show} for us:

\begin{verbatim}
instance Show AB where
    show :: AB -> String
    show A = "A"
    show B = "B"
\end{verbatim}

Allowing us to print a value of type \texttt{Show} like so:

\begin{verbatim}
*Main> x = A
*Main> print x
A
\end{verbatim}

This is great! Regardless of the shape of the data structure we define, our derived \texttt{Eq} and \texttt{Show} instances will know how to navigate that shape and compare or print values with that structure. Unfortunately, this greatness is restricted. The compiler only knows how to do this code generation with a limited list of privileged typeclasses: \texttt{Eq}, \texttt{Ord}, \texttt{Enum}, \texttt{Num}, \texttt{Show}, and \texttt{Read}. Over time, additional typeclasses have been added to this list, but only through modification of the compiler itself; not exactly a scalable path.

Compiler-independent datatype-generic programming is unlocked by derivation of the \texttt{Generic} typeclass. First implemented in the Utrecht Haskell Compiler (UHC) [MDJL10] and then in 2011 in the Glasgow Haskell Compiler (GHC) [ghca], this mechanism enables derivation of a \textit{generic runtime representation} of a datatype. From that point, all the information the compiler normally uses to magically derive typeclasses like \texttt{Eq} or \texttt{Show}, such as the data constructors (\texttt{A} and \texttt{B} in the example above) and the relationship between them (a value of type \texttt{AB} is either \texttt{A} or \texttt{B}), is available to the Haskell programmer. So, if the compiler can be modified to derive \texttt{Generic} instances, and the programmer can specify how to use a \texttt{Generic} instance to derive any other typeclass, then by the transitive property the compiler can derive instances for any user-defined typeclass without requiring any additional compiler modification [MDJL10].

This method is now explained by a description of a simplified version of this system, along with multiple examples. To begin to treat datatypes generically, we need a unified representation of types within data. The representation should capture this tree type, for instance:

\begin{verbatim}
data Tree a = Leaf | Node a (Tree a) (Tree a)
\end{verbatim}

and allow us to \textit{reason} about the structure of the type.
2.3.1 Type Representation

The following representation components allow us to represent most Haskell datatypes:

- **Unit**: Constructor without arguments (Nullary constructor). For example, the Leaf constructor above.
  
  ```haskell
data Unit = U
```

- **Constants, Parameters, and Recursion**: This can be a type constant such as Integer, a type parameter like a in the Tree above, or a recursive type reference like the two Trees in the Node constructor above.
  
  ```haskell
data K i c = K c
```

  In this declaration, i represents the choice of constant, parameter, or recursion, while c represents the constant, parameterized, or recursive type itself (such as Integer, a, or Tree).

- **Meta-information**: For instance, the representation of a Leaf above would have meta-information keeping track of the fact that the name of that constructor is "Leaf".
  
  ```haskell
data Meta i c a = M c a
```

  In this declaration, i represents which meta-information is contained (constructor name or datatype name, for instance), while c represents the meta-information itself. The a parameter represents the data the meta-information is about.

- **Sum**: Encodes our choice between multiple constructors. Mirroring "|" in data declarations, this represents that a Tree can be either a Leaf or a Node.
  
  ```haskell
data (:+:) f g = Left f | Right g
```

- **Product**: Encodes multiple arguments within the same constructor. This captures Node's multiple arguments: the type variable a, and the two recursive Tree as.
  
  ```haskell
data (:*:) f g = f :*: g
```

This system can now be used to give a generic representation type of our Tree type:

```haskell
data Tree a = Leaf | Node a (Tree a) (Tree a)
```

```haskell
data Rep Tree a =
  Meta DataName TreeData {
    (Meta ConName LeafCon {
      Unit
    }:+:
    Meta ConName NodeCon {
      K Parameter a
    }::*:
    K Recursive (Tree a)
  }::*:
    K Recursive (Tree a))
```
We can see how the representation type for Trees mirrors the original Tree type declaration: A Tree can either be a Leaf or a Node. A Leaf has no content besides the constructor, so it is a Unit. A Node has a parameter matching the Tree’s type variable a, and two recursive sub-Trees.

**DataName** and **ConName** are empty datatypes signalling the sort of meta-information encoded. **LeafCon** and **NodeCon** are empty datatypes that are instances of the **Constructor** typeclass. The instance method of **Constructor** that those types need to implement is **conName**, which returns the constructor name as a String. Similarly, **TreeData** is an instance of the **Datatype** typeclass which requires a **dataName** method. This is a way to get around the restriction that only constructors and types may appear in a type declaration. By encoding the values "Tree", "Leaf", and "Node" in types, we can include them in this representation. But those types are merely stand-ins for specific Strings that can be accessed by the overloaded **conName** and **dataName** functions:

```haskell
data TreeData
data NodeCon
data LeafCon

instance Datatype TreeData where dataName = "Tree"
instance Constructor LeafCon where conName = "Leaf"
instance Constructor NodeCon where conName = "Node"

conName LeafCon => "Leaf"
conName NodeCon => "Node"
```

Importantly for the user, the code of a representation type like **Rep Tree** is generated automatically by the compiler when a datatype uses the deriving (Generic) mechanism. In addition, the compiler generates two methods **from** and **to** that map back and forth between a type and its representation.

```haskell
class Generic a where
  from :: a -> Rep a
  to :: Rep a -> a
```

For instance, if **Generic** is derived on Tree, then we can use the **from** method on a value of type **Tree** to get a value of type **Rep Tree**:

```haskell
from Leaf =>
  M TreeData ( -- Datatype Meta-information
    Left ( -- This is a Leaf, not a Node
      M LeafCon ( -- Constructor Meta-information
        U)))) -- Nullary constructor
```

We can then use **to** on the result to recover the original Leaf.

If we now define a function by explaining how to operate on a representation type such as **Rep Tree**, we will have defined a function that can operate on a **Tree** without ever specifically defining it over **Trees**: Generic Programming!
2.3.2 Example: Generic Equality

We have already shown how the Haskell compiler can derive `Eq` instances for us, but defining something as intuitive as equality in a generic fashion will illustrate the system nicely. The class `Eq` is specified like so:

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

To define `Eq` generically, we introduce a new typeclass, `GEq`:

```haskell
class GEq r where
  geq :: r a -> r a -> Bool
```

Instances of `Eq` are types; instances of `GEq` are representations of those types. By implementing equality on the representations, we implement equality of the types they represent. To do so, we create instances of `GEq` for each of the components of a representation:

- Two `Unit` types can only be both `U`, so they are always equal.

  ```haskell
  instance GEq Unit where
    geq U U = True
  ```

- If there is a choice between two constructors, then the choices must be the same for both values, and the data inside the constructors must be the same as well.

  ```haskell
  instance (GEq a, GEq b) => GEq (a :+: b) where
    geq (Left x) (Left y) = geq x y
    geq (Right x) (Right y) = geq x y
    geq (Left x) (Right y) = False
    geq (Right x) (Left y) = False
  ```

- Multiple arguments to the constructors must all be equal.

  ```haskell
  instance (GEq a, GEq b) => GEq (a :*: b) where
    geq (x :*: y) (x' :*: y') = (geq x x') && (geq y y')
  ```

- Equality depends on constructor choice but not on the constructor names. We can ignore any meta-information in this context.

  ```haskell
  instance GEq a => GEq (Meta i c a) where
    geq (M c x) (M c y) = geq x y
  ```

- Finally, we consider constants, parameters, and recursive types. Here we rely on primitive instances of `Eq` (not `GEq`) such as `Bool`, `Integer`, `Char` to perform the comparison.

  ```haskell
  instance Eq a => GEq (K i a) where
    geq (K x) (K y) = x == y
  ```

Finally, we add a default method to our typeclass definition of `Eq` that makes use of `GEq`. This means that a programmer can make their type an instance of `Eq` by providing an empty instance declaration; the compiler will then use the default method:
class Eq a where
    (==) :: a -> a -> Bool
    default (==) :: (Generic a, GEq (Rep a)) => a -> a -> Bool
    (==) x y = geq (from x) (from y)

This default (==) makes use of the from method described above. We have access to
it because of the type constraint Generic a. If we assume that various instances for
primitive types have already been provided for Eq, this completes the Generic defini-
tion of equality.

2.3.3 Example: Generic Printing

An example close to pretty-printing is a generic Show definition. Again, as before,
we define a helper class GShow, then give instances of GShow for all components of
representation types. Note that this definition is slightly different from the version of
Show present in the Haskell Prelude; this is done for simplicity. A Prelude-compliant
implementation of generic Show is present in [MDJL10].

- Nullary constructors, sums, and datatype meta-information are ignored.

    class Show a where
        show :: a -> String
        default show :: (Generic a, GShow (Rep a)) => a -> String
        show x = gshow (from x)

    class GShow r where
        gshow :: r a -> String

    instance GShow Unit where
        gshow U = "" -- empty string

    instance (GShow a, GShow b) => GShow (a :+: b) where
        gshow (Left x) = gshow x
        gshow (Right x) = gshow x

    instance GShow a => GShow (Meta DataName d a) where
        gshow (M d x) = gshow x

- Products are concatenated with a space between:

    instance (GShow a, GShow b) => GShow (a :*: b) where
        gshow (x :*: y) = (gshow x) ++ " " ++ (gshow y)

- Constructor meta-information is specifically printed:

    instance (GShow a, Constructor c) => (Meta ConName c a) where
        gshow (M c x) = "(" ++ (gshow (conName c))
                       ++ " " ++
                       (gshow x) ++ ")"
• We rely on primitive implementations of Show and use show recursively to complete the behaviour on constants:

\[
\text{instance } \text{Show } a \Rightarrow \text{GShow } (K \ i \ a) \text{ where } \\
gshow (K \ x) = \text{show } x
\]

Again, with simple definitions for primitive types, we have a generic definition of Show that we can apply to any Generic type. This is the magic behind GenericPretty. The library explains how to pretty-print a Generic instance by providing instances for each of the representation components. Rather than constructing simple strings like Show, Out produces Docs that accompany parentheses with newlines and indentation. Together with the compiler support for deriving Generics, this suddenly makes pretty-printing available at the drop of a deriving statement!

Note: Readers familiar with Generic Programming will see that the system presented here is simplified from the system provided by Magalhães et al. [MDJL10]. This is to strike a balance between understandable Generic Programming and a fully expressive representation, in favor of communicating the basic concepts effectively. Readers interested in pursuing the finer details of the implementation in GHC should consult section 2.1 of ’A Generic Deriving Mechanism for Haskell’ [MDJL10], or the more informal explanation given by the authors when Generic was introduced to GHC in version 7.2 [ghca].
Chapter 3

Pretty-Printing For First-Years

3.1 Trees in Tutorials

Informatics students at the University of Edinburgh have a compulsory first-year first-
semester course in Haskell. Over the course of the semester, they complete various
tutorials in Haskell, which take the form of a template Haskell file with undefined
functions that they fill out, according to instructions distributed separately.

One of these tutorials covers Turtle Graphics. To specify the path that a turtle should
take, the following data type is defined in an external module:

type Angle = Float
type Distance = Float
data Command = Go Distance |
               | Turn Angle |
               | Sit |
               | Command :#: Command |
               | Branch Command |
               | GrabPen Pen |
               deriving (Eq, Ord, Show)

Where Pen is another datatype defined in the same file which controls the color of the
line the turtle draws:

data Pen = Colour Float Float Float |
          | Inkless |
          deriving (Eq, Ord, Show)

For instance, this command:

Go 20.0 :#: Turn 120.0 :#: Go 20.0 :#: Turn 120.0 :#: Go 20.0

Corresponds to this image:
Later on in the tutorial, students are asked to define various functions that produce commands corresponding to various shapes, and finally define functions that produce complex fractals. Here is an example of a 3rd-generation Sierpinski Arrowhead fractal:

And here is the Command that created that example:

```
(((GrabPen (Colour 0.0 0.0 1.0) #: Go 10.0) #: (Turn (-60.0) #: ((GrabPen (Colour 1.0 0.0 0.0) #: Go 10.0) #: (Turn (-60.0) #: (GrabPen (Colour 0.0 0.0 1.0) #: Go 10.0)))) #: (Turn 60.0 #: ((GrabPen (Colour 0.0 0.0 1.0) #: Go 10.0) #: (Turn 60.0 #: (GrabPen (Colour 1.0 0.0 0.0) #: Go 10.0)))))) #: (Turn (-60.0) #: ((GrabPen (Colour 0.0 0.0 1.0) #: Go 10.0)) #: (Turn 60.0 #: ((GrabPen (Colour 0.0 0.0 1.0) #: Go 10.0)) #: (Turn (-60.0) #: (GrabPen (Colour 1.0 0.0 0.0) #: Go 10.0)))))) #: (Turn (-60.0) #: ((GrabPen (Colour 0.0 0.0 1.0) #: Go 10.0)) #: (Turn 60.0 #: ((GrabPen (Colour 0.0 0.0 1.0) #: Go 10.0)) #: (Turn (-60.0) #: (GrabPen (Colour 1.0 0.0 0.0) #: Go 10.0)))))) #: (Turn (60.0 #: ((GrabPen (Colour 0.0 0.0 1.0) #: Go 10.0) #: (Turn (-60.0) #: (GrabPen (Colour 1.0 0.0 0.0) #: Go 10.0))))))))) #: (Turn (60.0 #: ((GrabPen (Colour 0.0 0.0 1.0) #: Go 10.0) #: (Turn (-60.0) #: (GrabPen (Colour 1.0 0.0 0.0) #: Go 10.0))))))))) #: (Turn (60.0 #: ((GrabPen (Colour 0.0 0.0 1.0) #: Go 10.0) #: (Turn (-60.0) #: (GrabPen (Colour 1.0 0.0 0.0) #: Go 10.0))))))))) #: (Turn (-60.0) #: (GrabPen (Colour 1.0 0.0 0.0) #: Go 10.0) #: (Turn (-60.0) #: (GrabPen (Colour 1.0 0.0 0.0) #: Go 10.0))) #: (Turn 60.0 #: ((GrabPen (Colour 1.0 0.0 0.0) #: Go 10.0)) #: (Turn (60.0 #: (GrabPen (Colour 1.0 0.0 0.0) #: Go 10.0)) #: (Turn 60.0 #: ((GrabPen (Colour 1.0 0.0 0.0) #: Go 10.0)) #: (Turn (-60.0) #: (GrabPen (Colour 1.0 0.0 0.0) #: Go 10.0)) #: (Turn (-60.0) #: (GrabPen (Colour 0.0 0.0 1.0) #: Go 10.0)))))))))
```

This is a convoluted and cumbersome twenty-five full lines of text with a maximum of
twelve levels of nesting. Since the output is forced onto a single line, it is difficult for
the programmer to parse. This is a familiar situation for anyone who has defined their
own data type in Haskell and used it to store large amounts of structured information!
Debugging these programs is frustrating because the printed value is almost impossible
to read.

3.2 GenericPretty For Commands

To address this struggle, I introduced GenericPretty to this tutorial to derive a pretty-
printer for the Command datatype. The steps for doing so were straightforward:

1. GenericPretty was installed through cabal, and imported in the LSystem.hs file.

2. The Command datatype definition was changed to derive Generic and Out rather
   than Show, like so:
   
   ```haskell
data Command = Go Distance
   | Turn Angle
   | Sit
   | Command :#: Command
   | Branch Command
   | GrabPen Pen
   deriving (Eq, Ord, Generic, Out)
   ```

3. Most of the Command constructors already have Out instances available, however
   since Pen is also user-defined we need to derive Generic and Out instead of
   Show for Pen as well, like so:
   
   ```haskell
data Pen = Colour Float Float Float | Inkless
   deriving (Eq, Ord, Generic, Out)
   ```

4. The derivation mechanism ordinarily only allows certain privileged typeclasses
   (Eq, Ord, Bounded, Enum, Show and Read) to be automatically derived. To allow
   derivation of the Generic and Out instances for our datatypes, we need to enable
   the DeriveGeneric and DeriveAnyClass language pragmas like so:
   
   ```haskell
   {-# LANGUAGE DeriveGeneric, DeriveAnyClass #-}
   ```

We are done! All that remains is to use the alternative print command, `pp`, to print the
Commands.

The immediate result of the above steps, in comparison to an unpretty version: Printed
Using Show:

```
*Tutorial7> print (arrowhead 2)
((GrabPen (Colour 1.0 0.0 0.0) :#: Go 10.0) :#: (Turn 60.0 :#: ((GrabPen (Colour 0.0
0.0 1.0) :#: Go 10.0) :#: (Turn 60.0 :#: (GrabPen (Colour 1.0 0.0 0.0) :#: Go 10.0)))))
 :#: (Turn (-60.0) :#: (((GrabPen (Colour 0.0 0.0 1.0) :#: Go 10.0) :#: (Turn (-60.0) :#: ((GrabPen (Colour
0.0 0.0 1.0) :#: Go 10.0))) :#: (Turn (-60.0) :#: ((GrabPen (Colour 1.0 0.0 0.0) :#: Go 10.0) :#: (Turn
60.0 :#: (GrabPen (Colour 1.0 0.0 0.0) :#: Go 10.0)))))
```
Pretty-Printed Using `Out`:

*Tutorial7> pp (arrowhead 2)
  ((GrabPen (Colour 1.0 0.0 0.0) :#: Go 10.0) :#: (Turn 60.0 :#: ((GrabPen (Colour 0.0 0.0 1.0) :#: Go 10.0) :#: (Turn 60.0 :#: (GrabPen (Colour 1.0 0.0 0.0) :#: Go 10.0)))) :#: (Turn (-60.0) :#: ((GrabPen (Colour 0.0 0.0 1.0) :#: Go 10.0) :#: (Turn (-60.0) :#: ((GrabPen (Colour 1.0 0.0 0.0) :#: Go 10.0) :#: (Turn (-60.0) :#: ((GrabPen (Colour 0.0 0.0 1.0) :#: Go 10.0) :#: (Turn (-60.0) :#: ((GrabPen (Colour 1.0 0.0 0.0) :#: Go 10.0)))) :#: (Turn (-60.0) :#: ((GrabPen (Colour 1.0 0.0 0.0) :#: Go 10.0)))) :#: (Turn (-60.0) :#: ((GrabPen (Colour 1.0 0.0 0.0) :#: Go 10.0)))))))))))

### 3.3 Default Pretty-Printing

One issue I noted with the pretty-printer is that it requires the use of the `pp` (Command) function anytime you wish to pretty-print something at the REPL. Fortunately, `ghci` has a helpful flag for this explicit purpose; the `-interactive-print` flag. By running `ghci` as follows:

```
ghci -interactive-print=pp
```

The REPL will use `pp` instead of the default `Show-based System.Out.print` like so:

*Tutorial7> arrowhead 1
  (GrabPen (Colour 0.0 0.0 1.0) :#: Go 10.0) :#: (Turn (-60.0) :#: ((GrabPen (Colour 1.0 0.0 0.0) :#: Go 10.0) :#: (Turn (-60.0) :#: ((GrabPen (Colour 0.0 0.0 1.0) :#: Go 10.0))))))
3.4 Parenthetical Subtleties

As discussed, Commands can be stated in code like so:

Go 10.0 :#: Turn 20.0 :#: Go 10.0 :#: Turn 20.0

But since the :#: infix constructor combines only two Commands, the compiler needs to decide where to place the parentheses. In absence of explicit direction, it goes for the right-associative default:

rightAssoc = Go 10.0 :#: (Turn 20.0 :#: (Go 10.0 :#: Turn 20.0))

But the same output image can be represented like so:

balanced = (Go 10.0 :#: Turn 20.0) :#: (Go 10.0 :#: Turn 20.0)

The difference is insubstantial when rendered on a single line, but the clarity can differ when pretty-printed:

*Tutorial7> ppLen 30 rightAssoc
Go 10.0 :#: (Turn 20.0 :#: (Go 10.0 :#: Turn 20.0))

*Tutorial7> ppLen 30 balanced
(Go 10.0 :#: Turn 20.0) :#: (Go 10.0 :#: Turn 20.0)

So how is this relevant? Well, the fractal Commands are created via a function that takes as its argument the ‘generation’ of the fractal. The Command is assembled out of various components, some of which are recursive calls to the fractal functions with the generation decremented. This leads to the self-similarity of the fractals, drawing a delightful parallel between programmatic and visual recursion. The placement of parentheses within this function definition changes nothing about the eventual visual, but shapes the pretty-printed layout a great deal.

Here’s a Haskell implementation of the Hilbert Fractal, a space-filling curve of right angles.

hilbert :: Int -> Command
hilbert x = l x where
  l 0 = Sit
  l x = p :#: r (x-1) :#: f :#: n :#: l (x-1) :#: f :#: l (x-1) :#: n :#: f :#: r (x-1) :#: p
  r 0 = Sit
  r x = n :#: l (x-1) :#: f :#: p :#: r (x-1) :#: f :#: r (x-1) :#: p :#: f :#: l (x-1) :#: n
  f = Go 10
  n = Turn 90
  p = Turn (-90)
And here’s another version of this implementation, with nothing changed except the parentheses:

hilbertParen :: Int -> Command
hilbertParen x = l x where
  l 0 = Sit
  l x = (p :#: (r (x-1) :#: f)) :#: ((n :#: l (x-1)) :#: f :#: (l (x-1) :#: n)) :#: (f :#: r (x-1)) :#: p
  r 0 = Sit
  r x = (n :#: (l (x-1) :#: f)) :#: ((p :#: r (x-1)) :#: f :#: (r (x-1) :#: p)) :#: (f :#: l (x-1)) :#: n
  f = Go 10
  n = Turn 90
  p = Turn (-90)

The resulting Commands are semantically identical, but pretty-print in dramatically different ways. The first iteration of the Hilbert fractal without parentheses pretty-prints like this:

*Tutorial7> pp (hilbert 2)
Turn (-90.0) :#:
  ((Turn 90.0 :#:
    (Sit :#:
      (Go 10.0 :#:
        (Turn (-90.0) :#:
          (Sit :#:
            (Go 10.0 :#:
              (Sit :#:
                (Go 10.0 :#:
                  (Sit :#:
                    (Go 10.0 :#:
                      (Turn 90.0)))))
    ()))))
      )
  )
: #:
  (Go 10.0 :#:
    (Turn 90.0 :#:
      ((Turn (-90.0) :#:
        (Sit :#:
          (Go 10.0 :#:
            (Sit :#:
              (Go 10.0 :#:
                (Sit :#:
                  (Turn 90.0)))))
      )))
  )
: #:
: #:
: #:
Whereas with designed parentheses:

*Tutorial7> pp (hilbertParen 2)
(Turn (-90.0)))()))))))) :#
  (Go 10.0 :#
    ((Turn (-90.0) :#
      (Sit :#
        (Go 10.0 :#
          (Turn 90.0 :#
            (Sit :#
              (Go 10.0 :#
                (Sit :#
                  (Turn (-90.0)))))))))))) :#
  (Go 10.0 :#
    (Sit :#
      Turn (-90.0)))))))))))) :#
  (Turn 90.0 :#
    (Go 10.0 :#
      ((Turn 90.0 :#
        (Sit :#
          (Go 10.0 :#
            (Sit :#
              Turn (-90.0)))))) #)
  (-90.0) :#
  (Sit :#
    (Go 10.0 :#
      (Sit :#
        (Turn (-90.0) :#
          (Go 10.0 :#
            (Sit :#
              Turn 90.0)))))))))) :#
    Turn (-90.0)))))))))))

Whereas with designed parentheses:

*Tutorial7> pp (hilbertParen 2)
(Turn (-90.0)) :#
  (((Turn 90.0 :#: (Sit :#: Go 10.0)) :#
    (((Turn (-90.0) :#: Sit) :#
      (Go 10.0 :#: (Sit :#: Turn (-90.0))))))) :#:
The number of lines is reduced from over 35 (one per atomic Command) to 23. Given that Commands are trees, default right-association is essentially the least balanced tree possible, with depth linear in the number of elements rather than logarithmic. The depth correlates directly to the number of lines used, as it is conveyed through indentation following newlines.

The output of the pretty-printer for very large datatypes is actually not very pretty at all. Outputs of low-generation fractals stretch to 50 lines or more, because each atomic command needs an entire line devoted to it. To make things worse, the line is diagonal. At some point, the terminal window will introduce additional line breaks to avoid the line going off screen, and we end up with a messy zigzag. Yes, each atom is visually distinct and we have no trouble following the line downward, but it comes at the cost of consuming the entire window. It’s not worse than unprettily smashing the entire thing into a single line, but it’s not much better either.

This is a problem that pushes more at the limits of pretty-printing in general than it does at GenericPretty. At some point, there is simply no way to render a value above a certain size in a manner that is readable. Razvan’s library is producing exactly what we asked for, in line with the Hughes-PJ pretty-printing library it is based on. It is printing these trees correctly from an algebraic standpoint, but readability remains unsatisfyingly unfulfilled.

One potential readability improvement for our case would be to feed the Commands through a balancing algorithm, producing a balanced tree semantically identical to the original but minimizing the depth, but the key words ‘for our case’ hint that this would no longer be Generic. The :# operator is associative, making rebalancing harmless, but plenty of trees have infix operators that are not - take a datatype for exponential expressions, for instance:

```
((Go 10.0 :#: Sit) :#: Turn 90.0)) :#
  (Go 10.0)) :#
  (Go 10.0 :#: Sit) :#
    (Go 10.0 :#: Turn 90.0)) :#
  (Go 10.0 :#: Sit) :#
    (Go 10.0 :#: Turn (-90.0))) :#
(Turn 90.0)) :#
  (Go 10.0 :#: Sit) :#
    (Go 10.0 :#: Turn 90.0)) :#
  (Go 10.0 :#: Sit) :#
    (Go 10.0 :#: Turn (-90.0))) :#
(Turn (-90.0))
```
data Exp = Value Integer | Exp :^: Exp

The following value of type Exp:

(Value 2 :^: Value 3) :^: Value 2

Is not semantically equivalent to:

Value 2 :^: (Value 3 :^: Value 2)

Because $8^2 (64)$ does not equal $2^9 (512)$. Therefore, our only generic option to improve the balance of GenericPretty-printed trees is through parenthesizing of the Command-generating functions.

The preference of parenthesization and grouping in pretty-printing is not easy to comment on. Yes, the basic example shown occupies a large amount of space and is messily draped across the terminal window, but the second option sacrifices readability as well by forcing the programmer to read across the line before moving on. And for smaller examples, the diagonal ribbon of text is easily more readable. The intention is merely to illustrate an interesting relationship I discovered, and to provide further justification that pretty is not the same as perfect.

### 3.5 Results

In Autumn 2020, this modified tutorial was released to students as an optional challenge they could attempt. Feedback was collected from the students by way of a survey. Of the 400 students who received the tutorial exercises, 8 students both completed the optional challenge and responded to the survey. Notably, some pointed out that in several cases the many newlines and tab characters served to make the Command overwhelming, pushing the output far to the right and significantly impairing readability. However, the majority (6 of 8) commented that overall, readability was improved through the pretty-printer.
Chapter 4

EdPrelude

With GenericPretty implemented in tutorials, we began reviewing other potential areas for improvement in the First-Year Haskell Experience. The idea arose to look into a custom version of Haskell’s Standard Prelude, the standard library of Haskell types and functions that is implicitly imported in all Haskell files. This alternative Prelude (termed EdPrelude, both for Edinburgh Prelude and Educational Prelude) could address common concerns of first-years when they are still figuring out the language.

4.1 Alternative Preludes

As mentioned, the Standard Prelude is imported implicitly in all Haskell files. However, this implicit import can be disabled by either providing an explicit import statement with an empty import list, like so:

```haskell
import Prelude ()
```

Or through the `NoImplicitPrelude` language pragma:

```haskell
{-# LANGUAGE NoImplicitPrelude #-}
```

Or through the `-XNoImplicitPrelude` flag of the `ghci` command:

```
$ ghci -XNoImplicitPrelude <filename>
```

In order to determine what functions EdPrelude would need to implement, I combed through the latest version of each tutorial and its template solution, making note of every library function used. I also took note of the library functions students are allowed to use in the Inf1A exam. Finally, some functions come in important conceptual groups. `take` and `drop`, for instance, are similar list operations: `take` returns the first n elements of a list, `drop` returns the remainder after the first n elements are removed. If just `take` was present in tutorials, it would be foolish to not include `drop` in EdPrelude as well.

The Standard Prelude was initially defined in the Haskell Report [HPJW+92]. Since then the Prelude has expanded and evolved, but it has maintained a distributed module
structure, so the functions I sought to replace were spread across multiple modules. I considered two alternatives for defining EdPrelude:

Option 1. was to replace pieces from the ceiling down. The Haskell Standard Prelude, along with its supporting libraries, is contained in the base package \[\text{lib}\]. The file Prelude.hs imports and re-exports the entirety of Prelude. By starting from that file, functions could be changed by hiding them from the relevant import list, then providing new definitions to be re-exported. This would have the strength of ensuring a functional Prelude at all intermediary stages.

Option 2. was to build from the ground up. Beginning with an empty file, the export list could be filled out with the list of necessary functions for EdPrelude to implement. Then implementations could be provided for exactly those functions by either adding a specific import, or, if the function needed alteration, providing a definition within the file. This would have the strength of ensuring knowledge at each stage about what exactly was being imported and exported.

I experimented with both approaches, and eventually decided on the second option: building up from an empty file. This was because tracking every last import and export in code I did not structure was frustrating; I spent most of my time chasing the kind of bugs and arcane type signatures that I was trying to remove in the first place. My aim, to implement an extremely basic educational Prelude, was much better suited to starting from a blank file and steadily slotting pieces into place. In that way, I could gain much finer control over the project as it evolved, and prevent it from becoming overly bloated.

The following features of EdPrelude were explored: pretty-printing by default, no fixed-size integers, no unnecessary typeclasses, and short compiler flags. Each one will now be considered in turn.

### 4.2 Pretty-Printing By Default

The largest obstacle to pretty-printing has always been Show’s derivable advantage. But GenericPretty extends the deriving mechanism to Out as well. So, if we include GenericPretty within EdPrelude, we can expect all printing to be pretty without requiring the programmer do any work to specify pretty-printing instances.

Changing base Haskell to support printing through Out instead of Show would be a large step to take, requiring widespread awareness and support from the Haskell community. For the moment, removing any need to install a package or write setup code by changing underlying Haskell is infeasible. However, we model what it would be like to have default pretty-printing by making it available in the modified standard library:

First, EdPrelude imports GenericPretty and exports its relevant methods and typeclasses. This enables users to add deriving (Generic, Out) declarations to their datatypes as before. Next, building off of lessons learned from the tutorial implementation, the -interactive-print=pp flag is introduced to the alternative REPL command, making pretty-printing the default.
We now expect all code to derive `Generic` and `Out` rather than `Show`, but legacy code may still use `Show`. As a fallback, we can use `Show` to derive a trivial instance of `Out`. For example, consider the following datatype:

```haskell
data Tree = Leaf | Node Tree Tree deriving (Show)

exTree :: Tree
exTree = Node Leaf Leaf
```

Notably, `Show` is derived but `Out` is not. If we try to evaluate `exTree` and print it using our pretty default printer, `ghci` will complain that it has no idea how to pretty-print a value of type `Tree` because it has no instance of `Out`:

```
*GenericPretty> exTree
error:
  • No instance for (Out Tree) arising from a use of ‘pp’
  • In a stmt of an interactive GHCi command: pp it
```

This seems inconsistent. True, it doesn’t know how to print a `Tree` prettily, but it does understand how to print a `Tree` unprettily. The code for neatly coercing this unpretty representation to a trivial pretty one follows:

```haskell
instance {-# OVERLAPPABLE #-} Show a => Out a where
doc :: a -> Doc
doc x = text (show x)
docPrec :: Int -> a -> Doc
docPrec _ x = doc x
```

The function `show` is available because we know the type of `x` is a member of the `typeclass` `Show`. `show` returns the single-line string representation of `x`; we then use the basic `Doc` constructor `text` to complete the `Out` instance. (The `docPrec` method is required for the instance declaration, but is a simple redirect to the `doc` method in this case.)

This instance declaration is unusual for Haskell:

1. There are now two sources of `Out` instances: generic derivations and this `Show`-based coercion. How should the compiler choose? The `Overlappable` per-instance pragma resolves this by annotating this instance as the overlapping one. If this is the only instance, the compiler will choose it. If there is another, the compiler will choose the other. This is the behaviour we want; only using the `show`-based option when we have to.

2. Haskell does not ordinarily allow a type variable to be used twice in the instance type, as `a` is used above. This minor restriction is lifted by the `FlexibleInstances` pragma.

3. This instance type introduces potential undecidability, because the constraint `(Show a)` is not smaller than the instance head `(Out a)`. Similar to recursive algorithms, the compiler needs to be careful of cases where a problem is not
simplified. For instance, if we were to define another instance declaration with the following type: \texttt{Out \ a \Rightarrow \ Show \ a}, then the compiler would chase its tail in a spiral until the maximum stack depth is exceeded. The compiler prevents us from defining an instance in this way because the head is a pretty good clue to undecidability. However, we know that this will terminate, because \texttt{Show} never relies on \texttt{Out}. We can weaken the check the compiler performs with the \texttt{UndecidableInstances} pragma.

The per-instance \texttt{Overlappable} pragma was already shown in the instance declaration above. The two full-module pragmas are declared at the top of \texttt{EdPrelude} like so:

\texttt{{-# LANGUAGE UndecidableInstances, FlexibleInstances #-}}

Now, \texttt{EdPrelude} can pretty-print by default without worrying about breaking legacy code that relies on \texttt{Show}:

*\texttt{EdPrelude}> \texttt{exTree} \\
Node Leaf Leaf

In addition, we no longer have to worry that \texttt{GenericPretty} may someday fail to derive an \texttt{Out} instance for a particular type: \texttt{GenericPretty} can now do \texttt{no worse} than \texttt{Show}. As an example, I discovered that \texttt{GenericPretty} does not know how to derive \texttt{Out} instances for tuple types over 8 elements:

*\texttt{GenericPretty}> \texttt{(1,2,3,4,5,6,7)} \\
(1,2,3,4,5,6,7) \\
*\texttt{GenericPretty}> \texttt{(1,2,3,4,5,6,7,8)} \\
error: \\
• No instance for \texttt{(Out (t7, t6, t5, t4, t3, t2, t1, t0))} \\
  arising from a use of \texttt{pp'} \\
• In a stmt of an interactive GHCi command: pp it

But \texttt{Show} knows how to do this!

\texttt{Prelude}> \texttt{print (1,2,3,4,5,6,7,8)} \\
(1,2,3,4,5,6,7,8) \\

So \texttt{EdPrelude}, with its fallback to \texttt{Show}, can now do so as well:

*\texttt{EdPrelude}> \texttt{(1,2,3,4,5,6,7)} \\
(1,2,3,4,5,6,7) \\
*\texttt{EdPrelude}> \texttt{(1,2,3,4,5,6,7,8)} \\
(1,2,3,4,5,6,7,8) \\

The set of pretty-printable datatypes is now necessarily larger than the set of printable datatypes, which is a useful robustness quality. This system also provides elegant access to single-line printing: simply derive \texttt{Show} rather than \texttt{Generic, Out} to use the default unpretty instance.
4.3 Removing Fixed-Size Integers

Haskell presents the programmer with two main integer types: `Int` and `Integer` (Yes, trying to differentiate between them in conversation is as difficult as it sounds). An `Int` is a fixed-size signed integer, usually 32 or 64 bits. An `Integer` is an arbitrary-sized signed integer. There are other options available, such as `Int8`, `Int16`, `Int32`, `Int64` for specific-size signed integers, or `Word8`, `Word16`, `Word32`, `Word64` for specific-size *unsigned* integers, however `Int` is the only integer type `Inf1A` tutorials make use of.

Obviously, `Int` is the frontrunner for efficiency. Though compilers and interpreters can be statically clever, the fact remains that a fixed-size type can generally go through a CPU’s arithmetic circuits whole whereas an arbitrary-size type carries no such promise. A programmer could theoretically request numbers requiring up to the available memory on the system running the program, and these cases need to be accounted for in the design of arbitrary-size integer operations. Various fast algorithms exist for arbitrary-size arithmetic, but they must be implemented in software whereas fixed-size arithmetic algorithms can be implemented in hardware.

Though faster, `Ints` have several downsides. They will happily overflow without complaint in Haskell, but a programmer’s patience will overflow long before an `Integer` does. `Int` overflow can also be inconsistent: as mentioned above, when the size is not made explicit, `Ints` may be 32 or 64 bits, depending on processor architecture, configuration, or compiler. This introduces unpredictable behaviour; code that works on my machine may break on yours.

As a quick overflow example, the following small Haskell function calculates the factorial of a number:

```haskell
Prelude> fac n = if n <= 1 then 1 else n * fac (n-1)
Prelude> fac 5
120
```

The type of the value `fac 5` above is actually not definite - Haskell’s lazy typing means that all it knows is that the value is some sort of number. By providing an explicit type annotation, we can coerce `fac 5` to be any integer type. This lets us perform the following comparison between the coercion to `Integer` and coercion to `Int`:

```haskell
Prelude> fac 21 :: Integer
51090942171709440000
Prelude> fac 21 :: Int
-4249290049419214848
```

21! is the first factorial result that passes the representation limits of 64-bit signed integers, overflowing to a negative number.

To remove access to the `Int` constructor, it serves to simply not export it from `EdPrelude` module. However this is far from enough! The Standard Prelude’s functions are almost exclusively defined on `Ints` when handling integer values. Each function defined in that manner needs a new type declaration and function definition. For instance, `take`, a function that returns the first n items of a list:
4.4 Removing (Or Defaulting) Typeclasses

As described earlier in Section 2.3, typeclasses provide the Haskell type system with polymorphism. Typeclasses have required methods that work in a well-defined and generic way across all instance types, enabling overloaded operators. One example typeclass available in the Standard Prelude is `Foldable`. `Foldable` refers to types which can be ‘folded’, which refers to combining the constituent pieces of the type together using an operator. Fold operations can be accomplished with the higher-order function `foldr`:

\[
\text{foldr} :: (a \to b \to b) \to b \to [a] \to b
\]

\[
\text{foldr} f z \; \text{[]} = z
\]

\[
\text{foldr} f z \; (x:xs) = f \; x \; (\text{foldr} \; f \; z \; xs)
\]

This is a complex abstraction that is difficult to parse at first but has enormous potential. For instance, folding a list of numbers with the addition operator and the identity value of 0 yields the sum of the list:

\[
\text{foldr} \; 0 \; (+) \; [1, \ 2, \ 3] \\
\Rightarrow 1 + (2 + (3 + 0)) \\
\Rightarrow 6
\]

Folding a list of numbers with the multiplication operator and the identity value of 1 yields the product of the list:

\[
\text{foldr} \; 1 \; (*) \; [2, \ 3, \ 4] \\
\Rightarrow 2 \times (3 \times (4 \times 1)) \\
\Rightarrow 24
\]

This `foldr` is defined over lists, but other data types can be folded as well. Consider the `Tree` type introduced previously:

```haskell
data Tree a = Leaf a | Node (Tree a) (Tree a)
```

We can consider folding this Tree in a similar manner to folding a list. Since there is no concept of an empty tree, we do not need to make use of the base case `z` as we did with lists.

\[
\text{foldr} :: (a \to b \to b) \to b \to \text{Tree} a \to b
\]

\[
\text{foldr} f z \; \text{(Leaf} \; x) = x
\]

\[
\text{foldr} f z \; \text{(Node} \; t1 \; t2) = f \; (\text{foldr} \; f \; z \; t1) \; (\text{foldr} \; f \; z \; t2)
\]

Now, we can fold Trees too! For instance:

```haskell
exTree = Node (Node (Leaf 2) (Leaf 3)) (Leaf 4)
```
foldr (+) 0 exTree
   => (2 + 3) + 4
   => 9

An implementation of this foldr method is required to make a type an instance of the typeclass Foldable. Because operations like sums, products, lengths, etc. can be generalized over all types that support fold operations, to extend their genericity in Prelude those functions are defined over the type constraint Foldable. For instance, this is the type of the length function in the Prelude:

length :: Foldable t => t a -> Int

Since lists are a foldable data structure, this behaves completely as expected when applied to list types:

Prelude> length [1,2,3,4]
4

But because it’s defined over Foldable t rather than just [a], the same length function is usable for a foldable tree, where the leaves are counted in the same way list elements are counted:

Prelude> length exTree
3

This is a powerful Haskell feature, and enables some elegant polymorphic code! Unfortunately, when a first-year student goes to inquire what the type of length is, they’re presented with the arcane result above, and confusion results. Students never deal with any Foldable datatype besides lists in their first year - it’s helpful to the expert but off-putting to the beginner.

For EdPrelude, only typeclasses required to implement necessary functions (such as the Num typeclass, which implements addition (+) for numeric types) are re-exported from the module. In addition, function definitions were updated to comply with the new set of available typeclasses. For instance, length was redefined like so:

length :: [a] -> Integer

Now the type of length aligns with its usage by the students.

### 4.5 Shortening Compiler Flags

The ghci command takes as optional arguments many useful flags that underpin the functionalities of EdPrelude. Writing out every flag in its entirety is a repetitive, unmemorable, and error-prone affair - expecting first-year students to begin their Haskell experience with a mandatory copy-pasted behemoth is a recipe for disaster. To simplify the matter, I created a basic UNIX shell-script called ‘edhci’ that would act as a wrapper to ghci, passing all the flags required for the systems to work by default. This ensures the flags will be correct, consistent, and contained within five characters.
Throughout, the priority was to minimize the effort required by a user to write code using EdPrelude vs. Prelude. Prelude is accessible implicitly and universally; to match this as closely as possible I limited the alterations a user would need to make to their workflow to the following:

1. A single import of EdPrelude in the Haskell file they are working on.

2. The use of the edhci script rather than ghci to begin the interpreter.

The following ghci flags were relevant to achieving these aims:

### 4.5.1 -interactive-print

As described previously, this chooses a new default printer for the REPL for rendering the evaluation result of expressions.

### 4.5.2 -ghci-script, -ignore-dot-ghci

.ghci scripts are used to run certain commands automatically at the start of a ghci session. The interpreter looks in various places to find .ghci files, such as the current and home directories. The -ghci-script flag allows you to specify an arbitrary path to find a .ghci file, and the -ignore-dot-ghci flag ignores all .ghci files that are not explicitly specified. The specific .ghci script I passed in made sure that EdPrelude was loaded into the interpreter by default, and other .ghci scripts were ignored to prevent conflicts.

### 4.5.3 -i

Similar to .ghci scripts, the interpreter also has certain places it looks for imported modules. The -i flag allows the addition of other import directories to search. EdPrelude is not a package, so in order to make sure it was available from all other directories, I installed it in /usr/local/share/edprelude and used -i /usr/local/share/edprelude to allow the interpreter to find EdPrelude.hs regardless of where edhci was was called from.

### 4.5.4 -XNoImplicitPrelude

Previously I described the three ways the implicit Prelude import could be disabled: An empty explicit import via import Prelude () in the file, the NoImplicitPrelude language pragma in the file, or the use of the -XNoImplicitPrelude flag for the interpreter. To avoid users needing to write the explicit import or the pragma in their files, I chose the flag option and included it in edhci by default. If necessary the flag can be overridden by an explicit Prelude import in the file via import Prelude, though one should be wary of name conflicts resulting from the use of Prelude and EdPrelude together.
4.5.5 -XDeriveGeneric, -XDeriveAnyClass

Similar to -XNoImplicitPrelude these flags mimic the effect of including the equivalent DeriveGeneric and DeriveAnyClass language pragmas. Precludes the addition of these pragmas to make derivation of Generic and Out instances seamless.

The shorthand script edhci serves as a useful encapsulation of the above flags, and ensures straightforward access to EdPrelude. All other arguments to edhci, such as flags or specific files to load, are passed on to ghci unchanged. The implementation of edhci as a simple shell script allows advanced users to peer inside the wrapper to see the options used.
Chapter 5

Future Work

5.1 Improving GenericPretty

GenericPretty derives pretty-printers that conform to the Hughes-PJ school of pretty-printing, based on Hughes’ algebraic design of a pretty-printer [Hug95]. However, in 2003 Hughes’ pretty-printer was improved upon by Wadler in his paper ‘a prettier printer’ [Wad03]. This gave rise to the Wadler-Leijen school of pretty-printing. Wadler cites improvements of increased efficiency and 60% size in his design; it makes sense to confer those same improvements upon GenericPretty by reimplementing its functionality with compliance to the Wadler-Leijen pretty-printing library [Lei].

Another common pretty-printing function is that of coloring the terminal output. GenericPretty could potentially be extended to support this functionality, say by arbitrarily picking a color to correspond to each constructor. Much like an IDE’s syntax highlighting, this would make the structure of the data easier to visually parse.

5.2 Exploring Non-Generic Pretty-Printing

The goal of this project was to integrate GenericPretty with first-year tutorials, but when I examine the project from the outside, I find myself questioning why GenericPretty should be integrated in these tutorials.

A potential alternative would be to define a custom pretty-printer for the Command datatype. This would lose the immediacy and forwards-compatibility of a derived version, but would gain much finer control over the readability of the output.

1. The turtle graphics support colorations of drawn lines controlled by the GrabPen Command; these color changes could be reflected in coloration of the output.

2. Students create an optimise function that combines consecutive Go or Turn commands together, while eliminating semantically null Sit, Go 0, and Turn 0 values. This function could be applied to Commands before printing, removing nonessential constructors from the visual field.
3. The tree-balancing operation I alluded to in Section 3.4 could be applied to all Commands before they are pretty-printed, creating an optimal minimization of the tree depth.

4. Since the parenthesizing has no ultimate impact on the fractal output, the parentheses could be removed with no loss of information.

Even if the example benefits above are not compelling on their own, it may be worth experimenting to determine what is possible. GenericPretty is straightforward, and it has the admirable quality of “just working”, but it requires students to download an entire 3rd-party package just to pretty-print one datatype. An extension to the LSystem module distributed with Tutorial 7 would rely on no such download and would offer the increased customization described.

As a final argument in favor, Tutorial 6 of Inf1A contains a custom datatype for Well-Formed Formulas of binary propositional logic, and already comes with a custom pretty-printer as part of the tutorial module! I experimented with this pretty-printer in comparison to GenericPretty, and found it to be superior. For the following examples, the Wff datatype and example formula wff are defined as:

```haskell
data Wff a = V a --Logical Atom
  | T --True
  | F --False
  | Not (Wff a) --Negation
  | Wff a :+: Wff a --Or
  | Wff a :&: Wff a --And
  | Wff a :->: Wff a --Implication
  | Wff a :<->: Wff a --Equivalence

wff = (V P :&: (V Q :|: V R)) :&: ((Not (V P) :|: Not (V Q)) :&: (Not (V P) :|: Not (V R)))
```

The example formula `wff` can be pretty-printed using GenericPretty:

```haskell
(V P :&: (V Q :|: V R)) :&:
  ((Not (V P) :|: Not (V Q)) :&:
   (Not (V P) :|: Not (V R)))
```

Or it can be pretty-printed with Tutorial 6’s custom pretty-printer:

```haskell
P & (Q | R) & (~P | ~Q) & (~P | ~R)
```

The latter option is visually neater and available to students without any additional packages installed. In addition, this pretty-printer does not make use of its own typeclass like Out, but instead provides this output as a custom instance of Show. This makes it the default way Wffs are printed without any ghci flag required! It is worth investigating if GenericPretty is truly superior to an equivalent approach to Commands.
5.3 EdPrelude Testing

EdPrelude contains several alterations that logically should improve the accessibility of Haskell. The implementation functions and the performance is not significantly impacted, but given that the ultimate goal is not just working code but helpful code, the true test of EdPrelude’s worth is the response of Inf1A’s students. There are three main areas to explore through user testing:

First, how does EdPrelude compare to Prelude at the introductory stage? What is the impact of requiring a custom Haskell setup on the ability for the teaching support team to troubleshoot installations, a common source of problems for students? And to what extent do the modifications made to standard functions improve students’ experience of Haskell?

Second, how does EdPrelude impact eventual Haskell understanding? At some point, the transition will need to be made from the bubble of EdPrelude to the wider world of ordinary Haskell - what is the best way to handle that process?

Finally, EdPrelude is also a proof of concept, an extensible demonstration that pieces of the standard library can be replaced if there is room for improvement. What other areas of difficulty could EdPrelude be used to address?
Chapter 6

Conclusion

I introduced GenericPretty to a first-year Haskell tutorial that needed a pretty-printer. A student survey confirmed a general readability improvement, and there is space to explore how the representation of complex datatypes could be improved further.

Building on GenericPretty, I created EdPrelude, a functioning alternative standard library. EdPrelude shows that Haskell can do without Ints if necessary. It is a practical improvement for a student’s first look at Haskell. And it’s a meta-demonstration of the process of replacing Prelude.

Though my time as an undergraduate is ending and I do not have the integrated masters’ ability to see EdPrelude through to user testing in September 2021, I feel assured that this avenue of course improvement has been explored. I would be fascinated to hear of the results of any experimental usage, or the results of any further iteration upon the ideas presented.
Appendix A

EdPrelude

A.1 EdPrelude.hs

{-# LANGUAGE NoImplicitPrelude #-}
{-# LANGUAGE FlexibleInstances, UndecidableInstances #-}

-----------------------------------------------------------

-- Module : EdPrelude
-- Copyright : (c) Matthew Marsland
-- License : BSD-style
--
-- Maintainer : marslandm@me.com
--
-- The EdPrelude: A version of Prelude built for students supporting:
-- - Default pretty-printing via GenericPretty
-- - Restricted Numeric Classes
-- - Defaulted Function Signatures (Minimized typeclasses to Num (Real, Fractional, Integral), Eq, Ord, Enum, Show, Generic, Out, Applicative, and Functor)
-----------------------------------------------------------

module EdPrelude ( -- **Export List**
    -- Numeric Types and Functions
    Num((+), (-), (*), negate, abs, signum, fromInteger),
    Real(toRational),
    Fractional((/), recip),
    RealFrac(truncate, round, ceiling, floor),
    Integral(quot, rem, div, mod, quotRem, divMod, toInteger),
    Integer, Double, Rational,
    numerator, denominator,
    (%), (^), (^^),
    even, odd,
--Ordering Types and Functions
Ord(compare, (<), (<=), (>), (>=), max, min),
Ordering,

--Equality and Booleans
Eq((==), (/=)),
Bool(True, False),
otherwise, (||), (&&), not,

--Enum Types and Functions
E.Enum,
EdPrelude.toEnum, EdPrelude.fromEnum,

--List Types and Functions
length, genericLength, take, drop, takeWhile, dropWhile, sum, product, and, or, all, any, (!!), zip, zipWith, unzip, isPrefixOf, map, filter, elem, (++), repeat, replicate, cycle, head, tail, init, last, concat, delete, maximum, minimum, reverse, sort, sortOn, sortBy, nub, nubBy, unlines, lines, words, words, concatMap, null, lookup, transpose, foldr, foldl1, foldl, foldl1, (\\),

--Tuple Functions
fst, snd, curry, uncurry,

--Char Types and Functions
isDigit, isUpper, isLower, isAlpha, isAlphaNum, toUpper, toLower, digitToInteger, integerToDigit, chr, ord,

--Showing as Strings, Characters, and Strings
Show(..),
Char, String,

--Miscellaneous
undefined, error, errorWithoutStackTrace, ($), (.),
seq, sequence_, sequence, id, break,

--Lifting and Monads
Monad, Applicative,
(>>=), (>>), guard, return,
liftM, liftM2, replicateM,
Maybe(Just, Nothing),
Functor, ($$),
Appendix A. EdPrelude

-- IO
IO,
putStr, putStrLn, readFile,

-- Random
randomR, randomRIO, newStdGen,

-- Pretty-Printing
Generic, Out,
pp, print

) where

-- IMPORTED DEFINITIONS (UNCHANGED):
import GHC.Num
import GHC.Real
import Data.Eq
import Data.Ord
import Data.Bool (otherwise, not, (||), (&&), Bool(True, False))
import Data.Char (isDigit, isUpper, isLower, isAlpha, isAlphaNum)
import qualified GHC.Enum as E (Enum,toEnum,fromEnum)
import qualified Data.Char as C (ord,chr)
import Data.List ((++),takeWhile,dropWhile,delete,reverse,map,filter,
                   zip, unzip, zipWith, isPrefixOf, head, tail, init, last, concat,
                   words, unwords, lines, unlines, transpose, repeat, replicate,
                   sort, sortOn, sortBy, nub, nubBy, lookup, genericLength, (\))
import Data.Tuple (fst, snd, curry, uncurry)
import GHC.Show
import GHC.Types (Char,Bool(True,False),Double)
import GHC.Base (String,error,errorWithoutStackTrace,($),(.),undefined,seq,id)
import Data.Maybe (Maybe(Just,Nothing))
--import Control.Functor (Functor, (<<))
import Prelude (Functor, (<<), sequence_, sequence, cycle, break)
import Control.Monad (Monad,liftM,liftM2,(>>=),(>>),guard,return)
import Control.Applicative (Applicative)
import qualified Control.Monad as M (replicateM)
import System.IO (IO,putStr,putStrLn,readFile)
import System.Random (randomR,randomRIO,newStdGen)
import Text.PrettyPrint
import Text.PrettyPrint.GenericPretty
REDEFINED FUNCTIONS:
For comparison, the original type signatures are shown in comments

**Enum Functions**
toEnum :: Enum a => Int -> a
toEnum :: E.Enum a => Integer -> a
toEnum i = E.toEnum (fromIntegral i)

fromEnum :: Enum a => a -> Int
fromEnum :: E.Enum a => a -> Integer
fromEnum x = toInteger (E.fromEnum x)

**Char Functions**
toUpper :: Char -> Char
toUpper c
| isLower c = toEnum (fromEnum c - fromEnum 'a' + fromEnum 'A')
| otherwise = c

toLower :: Char -> Char
toLower c
| isUpper c = toEnum (fromEnum c - fromEnum 'A' + fromEnum 'a')
| otherwise = c

digitToInteger :: Char -> Integer
digitToInteger c
| isDigit c = fromEnum c - fromEnum '0'
| otherwise = errorWithoutStackTrace "Char.digitToInteger: not a digit."

integerToDigit :: Int -> Char
integerToDigit i
| i >= 0 && i <= 9 = toEnum (i + fromEnum '0')
| otherwise = errorWithoutStackTrace "Char.integerToDigit: not a digit."

ord :: Char -> Int
ord :: Char -> Integer
ord c = fromEnum c

chr :: Int -> Char
chr :: Integer -> Char
chr i = toEnum i
-- **List Functions**

-- length :: Foldable t => t a -> Int
length :: [a] -> Integer
length [] = 0
length (x:xs) = 1 + (length xs)

-- take :: Int -> [a] -> [a]
take :: Integer -> [a] -> [a]
take n [] = []
take 0 (x:xs) = []
take n (x:xs) = x : take (n-1) xs

-- drop :: Int -> [a] -> [a]
drop :: Integer -> [a] -> [a]
drop n [] = []
drop 0 xs = xs
drop n (x:xs) = drop (n-1) xs

-- sum :: (Num a, Foldable t) => t a -> a
sum :: Num a => [a] -> a
sum [] = 0
sum (x:xs) = x + sum xs

-- product :: (Num a, Foldable t) => t a -> a
product :: Num a => [a] -> a
product [] = 1
product (x:xs) = x * product xs

-- and :: Foldable t => t Bool -> Bool
and :: [Bool] -> Bool
and [] = True
and (b:bs) = b && (and bs)

-- or :: Foldable t => t Bool -> Bool
or :: [Bool] -> Bool
or [] = False
or (b:bs) = b || (or bs)

-- all :: Foldable t => (a -> Bool) -> t a -> Bool
all :: (a -> Bool) -> [a] -> Bool
all f xs = and (map f xs)

-- any :: Foldable t => (a -> Bool) -> t a -> Bool
any :: (a -> Bool) -> [a] -> Bool
any f xs = or (map f xs)
Appendix A. EdPrelude

--maximum :: (Ord a, Foldable t) => t a -> a
maximum :: (Ord a) => [a] -> a
maximum [] = errorWithoutStackTrace "EdPrelude.maximum: empty list"
maximum [x] = x
maximum (x:xs) = max x (maximum xs)

--minimum :: (Ord a, Foldable t) => t a -> a
minimum :: (Ord a) => [a] -> a
minimum [] = errorWithoutStackTrace "EdPrelude.minimum: empty list"
minimum [x] = x
minimum (x:xs) = min x (minimum xs)

--(!!) :: [a] -> Int -> a
infixl 9 !!
(!!) :: [a] -> Integer -> a
xs !! n | n < 0 = errorWithoutStackTrace "EdPrelude.!!: negative index"
[] !! _ = errorWithoutStackTrace "EdPrelude.!!: index too large"
(x:_!) !! 0 = x
(_:xs) !! n = xs !! (n-1)

--concatMap :: Foldable t => (a -> [b]) -> t a -> [b]
concatMap :: (a -> [b]) -> [a] -> [b]
concatMap f as = concat (map f as)

--elem :: (Eq a, Foldable t) => a -> t a -> Bool
elem :: (Eq a) => a -> [a] -> Bool
elem x [] = False
elem x (y:ys) = x == y || elem x ys

--null :: Foldable t => t a -> Bool
null :: [a] -> Bool
null [] = True
null xs = False

--foldr :: Foldable t => (a -> b -> b) -> b -> t a -> b
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr _ v [] = v
foldr f v xs = f (head xs) (foldr f v (tail xs))

--foldl :: Foldable t => (a -> b -> a) -> a -> t a -> a
foldl :: (a -> b -> a) -> a -> [b] -> a
foldl _ v [] = v
foldl f v xs = f (foldl f v (init xs)) (last xs)
Appendix A. EdPrelude

--foldr1 :: Foldable t => (a -> a -> a) -> t a -> a
foldr1 :: (a -> a -> a) -> [a] -> a
foldr1 _ [] = errorWithoutStackTrace "EdPrelude.foldr1: empty list"
foldr1 _ [x] = x
foldr1 f xs = f (head xs) (foldr1 f (tail xs))

--foldl1 :: Foldable t => (a -> a -> a) -> t a -> a
foldl1 :: (a -> a -> a) -> [a] -> a
foldl1 _ [] = errorWithoutStackTrace "EdPrelude.foldl1: empty list"
foldl1 _ [x] = x
foldl1 f xs = f (foldl1 f (init xs)) (last xs)

-- **Monad Functions**
--replicateM :: Applicative m => Int -> m a -> m [a]
replicateM :: Applicative m => Integer -> m a -> m [a]
replicateM i m = M.replicateM (fromIntegral i) m

-- **Pretty-Printing Functions**
print :: Out a => a -> IO ()
print x = ppStyle (Style {mode = PageMode, lineLength = 80, ribbonsPerLine = 2}) x

-- Automatic Derivation of Out Instances from Show Instances
instance {-# OVERLAPPABLE #-} Show a => Out a where
doc x = text (show x)
docPrec _ x = doc x

A.2 edhci

#!/bin/sh
ghci -XNoImplicitPrelude \
    -XInteractive-print=print \
    -ignore-dot-ghci -ghci-script /usr/local/share/edprelude/.ghci \
    -i/usr/local/share/edprelude \
    -XDeriveGeneric -XDeriveAnyClass \
    "$@"

A.3 .ghci

The .ghci script just ensures EdPrelude is loaded implicitly even if no file is loaded by edhci.

:load EdPrelude
Appendix B

GenericPretty Survey

B.1 Survey Questions

1. How was the installation experience (using cabal)? (Required)
   (a) Unsolvable Errors
   (b) Many Errors
   (c) Few Errors
   (d) No Errors

2. If you can, provide more detail to the above installation rating? (specific problems, time spent, etc.) (Optional)

3. When you used pretty-printing, how would you rate the pretty-printed version compared to a single-line output? (Required)
   (a) Much Better
   (b) More Readable
   (c) Equivalent
   (d) Less Readable
   (e) Much Worse
   (f) Did Not Use - Unsolvable Errors

4. If you can, provide more detail to the above readability rating? (Optional)

5. Were there any notable exceptions to the above general rating? (Optional)

6. If you can, note any values that pretty-printed in an interesting way below! (Optional)


