The number systems in Haskell 98

1.0.1 Foreword

Of all more or less popular programming languages, Haskell has the most complicated number system by far. And it is complicated from every perspective, there is no simple angle to start from. It is not very elegant in itself, but powerful and flexible. In this respect, purity and beauty has been sacrificed for the sake of usefulness. It is difficult to understand and difficult to explain, because it is the complex result of many different design paradigms. But however complicated, it is at least compact and we can summarize everything on one or two pages: figure 1 is the complete listing of all number–related Haskell 98 declarations. As far as the core mathematical aspect of the number system is concerned, figure 2 is a comprehensive summary and should suffice as a reference, once the picture is explained and understood. And for all string conversions of numeral representations, there is a separate part, summarized in figure 3.

So understanding Haskells number system is no more than understanding the pictures 2 and 3. And we introduce into this world by stepwise building up these hierarchies.

1.0.2 Remark

There are two established ways to look at numbers:

\(\mathbb{N} \subseteq \mathbb{Z} \subseteq \mathbb{Q} \subseteq \mathbb{R} \subseteq \mathbb{C}\)

In the mathematical tradition, there is a hierarchy, an evolution from natural numbers, integers, rational numbers, real numbers, to complex numbers. In this sequence, each number system emerges by overcoming certain operational limitations of the predecessor system. The whole is a beautiful and elegant achievement, shaped in the 19th century and a standard part of scientific culture ever since.

More recent is the computer science tradition. In the first place, this is making computers do what the mathematical tradition has taught us. But that did not come without certain sacrifices in accuracy and number size. In the computer language C for example, we have types like int for integers, and float and double for real numbers. But different to the mathematical number systems, these types are defined by machine words: int numbers are stored in 2 or 4 bytes, depending on the actual implementation, each float is made of 4 byte and double comprises 8 byte words (hence the title: “double” is “double size float”). And when the actual numbers become too big or too small for these limitations, things are rounded. Strictly speaking, that violates destroys the whole mathematical design. Of course, these inaccuracies can be precisely determined, there are established standardizations by now, the result is just another kind of mathematical theory. But the point is, that this is a different kind of thinking, nevertheless.

For a real understanding of Haskells number concept, we need to be aware of these two traditions, because they are both explicitly present. There is \(\mathbb{Z}\) and \(\mathbb{Q}\) in their full potentials (called Integer and Rational in Haskell), but int, float and double from C are reborn in Haskell as well (only with capital initials: Int, Float and Double).

1.0.3 Introduction

Anyway, let us start all over again. Our goal is the stepwise (re)construction of figure 2. And we take off in the middle.
1.1 The four sorts of numbers

1.1.1 Definition

There are four “sorts” of numbers in Haskell:

<table>
<thead>
<tr>
<th>Integral</th>
<th>RealFloat</th>
<th>Ratio</th>
<th>Complex</th>
</tr>
</thead>
</table>

1. Integral numbers are the Haskell version of the integers \( \mathbb{Z} \) in mathematics. As usual, the default representation is by decimal numerals with an optional negation symbol, as in

\[
123456789 \quad 0 \quad -77
\]

But it is also possible to use octal numerals (with a \( 0o \) prefix; e.g. \( 0o123 \) denotes the integer 83) and hexadecimal numerals (with a \( 0x \) prefix; e.g. \( 0x123 \) stands for 291).

2. RealFloat numbers are the Haskell name for what is commonly called floating-point numbers. As usual, there is the dot notation with optional \( e \) or \( E \) exponent, e.g.

\[
12.34 \quad -12.34e56 \quad 1234e-56 \quad 0.0 \quad 10E1234
\]

Floating point numbers approximate the real numbers \( \mathbb{R} \), but can only cope with a certain accuracy.

3. Ratio numbers are the Haskell version of the rational numbers \( \mathbb{Q} \). Recall, that the standard mathematical notation of a \( \mathbb{Q} \) element is

\[
\frac{n}{d} \quad \text{with} \quad n, d \in \mathbb{Z} \quad \text{and} \quad d \neq 0
\]

Due to the layout restrictions of a programming language, this has become

\[
n \div d \quad \text{with} \quad n \quad \text{and} \quad d \quad \text{being} \quad \text{Integral numbers}
\]

in Haskell.

4. Complex numbers are the Haskell version of the standard complex number system \( \mathbb{C} \) in mathematics. The default representation of such a number is a pair

\[
(x, y) \quad \text{or} \quad x + yi \quad \text{with} \quad x, y \in \mathbb{R}
\]

(\( i \) being the imaginary unit with \( i^2 = -1 \)). In Haskell notation, this is written

\[
x \div i + y \quad \text{with} \quad x \quad \text{and} \quad y \quad \text{being} \quad \text{RealFloat numbers}
\]

*** picture 4 shows the syntax of Integer and Float literals, as in the Haskell Report; but that is probably too much information ***

1.1.2 Remark

(1) The four names \texttt{Integral}, \texttt{RealFloat}, \texttt{Ratio} and \texttt{Complex} are Haskell keywords, but they are no proper types as such. For example, we cannot say “\texttt{123456::Integral}” or “\texttt{12.3456::RealFloat}”, that is no legal Haskell code. Of course, Haskell has types and type classes, but no sorts. Nevertheless, let us continue with our four “sorts” for now.

(2) The constructors \( \div \) for \texttt{Ratio} and \( :+ \) for \texttt{Complex} numbers may have optional spaces around them.

(3) In general, the number notion may refer to both, a kind of platonic value or a syntactic sequence of symbols. But if one specifically refers to the latter, i.e. the syntactical representa-

- There doesn’t seem to be a real standard in this respect. For example, “\texttt{123456}” (without spaces) and “\texttt{12 :+ 34}” (with spaces) is the default layout in GHC. But Hugs outputs “\texttt{12 \% 34}” instead.
- In the Haskell Report, a numeral is called a numeric literal.
- ML has a similar type system, and there, \texttt{int} and \texttt{real} numbers are really distinct. The numeral 0 is of type \texttt{int} and one has to write something like \texttt{0.0} to refer to zero in \texttt{real}. To migrate from one type to the other, one has to use explicit type converter functions.

(4) Are these four number sorts distinct? Well, “yes” and “no”, the full answer is complicated and has to wait after the introduction of the numeric type classes. But the short answer is a “yes”, we can use an \texttt{Integral} numeral like 1234 for any of the other three sorts as well.
1.1.3 Example input of numerals

Let us input some numbers in a GHCi session. We call ghci from the shell and its prompt invites for input. By default, this prompt is Prelude>

(1) Integral numbers in default decimal notation are basic values in the sense that they are not evaluated any further

Prelude> 1234
1234

Note, that it is possible in Haskell to deal with Integral numbers of arbitrary size

Prelude> 123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890
123456789012345678901234567890123456789012345678901234567890

Hexadecimal numerals (with a 0x, "0" is zero) and octal numerals are converted into the default decimal representation

Prelude> 0x1234
4660
Prelude> 0o1234
668

Recall, that these conversions are computed by

\[ 0x1234 = 4 \cdot 16^0 + 3 \cdot 16^1 + 2 \cdot 16^2 + 1 \cdot 16^3 = 4660 \]
\[ 0o1234 = 4 \cdot 8^0 + 3 \cdot 8^1 + 2 \cdot 8^2 + 1 \cdot 8^3 = 668 \]

(2) RealFloat numbers. One common representation is the decimal dot notation. But note, that the accuracy is limited and all too long numbers are shortened.

Prelude> 12.34
12.34
Prelude> 0.0
0.0
Prelude> 12.3456789012345678901234567890123456789012345678901234567890123456789012345678901234567890
12.3456789012345678901234567890123456789012345678901234567890

Recall, that these representations are computed by

\[ 0.1234 = 4 \cdot 10^{-1} + 2 \cdot 10^{-2} + 3 \cdot 10^{-3} = 0.1234 \]
\[ 0.01234 = 4 \cdot 10^{-4} + 3 \cdot 10^{-5} + 2 \cdot 10^{-6} + 1 \cdot 10^{-7} = 0.01234 \]

Also, there is the notation with the “e” or “E”. Recall, that “nEm” or “nem” stands for \( n \cdot 10^m \) (with \( 10^{-m} = \frac{1}{10^m} \))

Prelude> 1.234e0
1.234
Prelude> 1.234e1
12.34
Prelude> 1.234e-1
0.1234

And as usual, the default representation is with one digit preceding the dot:

Prelude> -12.34e6
-1.234567
Prelude> 1234e-56
1.23456e-53

But again, the accuracy is limited:

Prelude> 10E-1234
0.0

and so is the size of RealFloat numbers:

Prelude> 10E1234
Infinity

(3) Ratio numbers are implemented in the standard Ratio module. So we need to make its entities available first. Depending on the interpreter, there are several ways to load Ratio. In the GHC interpreter we use the :module or :m command.

Prelude> :module Ratio

The changed prompt indicates a successful loading. Example input is always changed to the unique reduced form (with positive denominator and no common divisor in nominator and denominator).

Prelude Ratio> -7 % 5
(-7)%5
Prelude Ratio> 7 % (-5) {: mind the parentheses! -}
(-7)%5

Made of two Integral numbers, Ratio numbers don’t suffer from any limits in size

Prelude Ratio> 7 % 12345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789
71123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789

Zero denominators are refused, as in other programming languages

Prelude Ratio> 0 % 2
*** Exception: Ratio.%: zero denominator

(4) Complex numbers are implemented in the standard Complex module. Again, we have its entities available after calling the :module or :m command.

Prelude> :module Complex

Any pair \( x, y \) of RealFloat numbers makes a complex number \( x + y \), where \( x \) is the real and \( y \) is the imaginary part.

Prelude Complex> 0.123 :::+ 123.0
0.123 :::+ 123.0
Prelude Complex> (-1234.56e-3) :::+ (-222.22) {
(parentheses! -}
(-1.23456) :::+ (-222.22)

In this context, every Integral \( x \) or \( y \) is accepted as a RealFloat

Prelude Complex> 1.0 :::+ 1.0
1.0 :::+ 1.0

Being pairs of RealFloat numbers, Complex numbers suffer from the same limitations in size and accuracy.

Prelude Complex> 1e1000 :::+ 1e1000
Infinity :::+ Infinity
Prelude Complex> 1e-1000 :::+ 1e-1000
0.0 :::+ 0.0

\footnote{GHC is the Glasgow Haskell Compiler suite and GHCi is its interactive/interpreter program.}

\footnote{For a repetition of the complex number system, see ???? below.}
1.2 The eight standard number types

1.2.1 Let us take the next step towards the hierarchy of figure 2. In definition 1.1.1, we started with our four “sorts” of numbers:

| Integral | RealFloat | Ratio | Complex |

Let us now get down to the proper number types in Haskell. It turns out, that the two primitive sorts Integral and RealFloat each split into two different types. And since each Ratio number is a composition of two Integral numbers, we have two types for Ratio as well. Similarly for Complex numbers, which are pairs of RealFloat numbers. So altogether, our four sorts split into eight proper Haskell types and these are the standard number types. In the end, we have a new picture:

However, if we need the real integers \( \mathbb{Z} \) without any limits, we may rather use

But note, that once the type is fixed, all values and results are bound to that type and type mixes lead to error messages, even if all types are Integral. For example, both the following inputs are fine:

```
> \text{let } \text{int } n = \text{5 : Integral} \text{; float } m = \text{5 : RealFloat} \text{; } \text{it } x = \text{5 : Integral} \text{; y = 6 : Integral} \text{; } \text{in } x + y
```

but this won’t work and produces an error message:

```
> \text{let } \text{int } n = \text{5 : Integer} \text{; float } m = \text{5 : RealFloat} \text{; } \text{in } x + y
```

```
 error ....
```

1.2.2 Definition the standard Integral number types

There are two Integral data types:

(a) data Int = minBound \ldots \text{-1 | 0 | 1 \ldots maxBound}

Fixed sized integers Int, ranging from minBound to maxBound, depending on the implementation. Int is very similar to the int type from C.

(b) data Integer = \ldots \text{-1 | 0 | 1 \ldots}

Integers of arbitrary size.

Integral itself is a type class

```haskell
class Integral a where ...
```

and thus has two instances

```haskell
instance Integral Int where ...
instance Integral Integer where ...
```

1.2.3 Remark

(i) From a purely functional point of view, this duality of types is absurd. Integer comprises all the members of Int and is safer, because it doesn’t interrupt or misbehaves due to unexpected overflows. But of course, Int is introduced into the language because it enables the use of built-in processor arithmetic, which is way faster. All “syntactical” operations in Haskell that involve numbers also use Int instead of Integer. For example,

```
length :: [a] -> Int or (\text{return}) :: [a] -> Int -> a
```

So if speed is not totally irrelevant and the values are certain to stay in a reasonable range, then Int should be the first choice.

(ii) The actual bounds of Int are depending on the implementation. But the Haskell Report demands at least

```haskell
\text{minBound \leq -2^{29} = -536870912}
\text{maxBound \geq 2^{31} - 1 = 536870911}
```

For example, on my own system (Debian Linux on an Intel Pentium Dual CPU) and with the GHC interpreter (version 6.8.2) I obtain:

```
> \text{minBound :: Int}
-2147483648
> \text{maxBound :: Int}
2147483647
```

1.2.4 Remark

If you need to write a program that involves Integral numbers, you may know in advance which type suits you more: either Int for fast functions and compatibility with the list function arguments or Integer for real large numbers. And in that case, you can fix the type everywhere by adding a type declaration to every definition, which is good programming style anyway.

For example, suppose we need a simple triple function, where say triple 5 is 15. If we know in advance, that we only operate on small Integral numbers, we should use this version

```
\text{triple :: Int \rightarrow Int}
\text{triple } n \text{ = } 3 \times n
```

but this won’t work and produces an error message:

```
> \text{let } \text{int } \text{triple } n = \text{3 : Integer} \text{; float } m = \text{5 : RealFloat} \text{; } \text{in } x + y
```

```
 error ....
```

1.2.5 Definition the standard RealFloat number types

There are two RealFloat data types:

(a) data Float = \ldots

Single precision floating point numbers, with a range depending on the implementation, very similar to the float type in C.

(b) data Double = \ldots

Double precision floating point numbers, with a range depending on the implementation, very similar to the double type in C.

RealFloat itself is a type class

```haskell
class RealFloat a where ...
```

and thus has two instances

```haskell
instance RealFloat Float where ...
instance RealFloat Double where ...
```

However, see also exercise 1.2.12, showing real Haskell systems may show some unexpected behavior in this respect.

---

\(^{1}\) However, we also exercise 1.2.12, showing real Haskell systems may show some unexpected behavior in this respect.

\(^{2}\) The actual bounds of Int are depending on the implementation. But the Haskell Report demands at least

\text{minBound \leq -2^{29} = -536870912}
\text{maxBound \geq 2^{31} - 1 = 536870911}

For example, on my own system (Debian Linux on an Intel Pentium Dual CPU) and with the GHC interpreter (version 6.8.2) I obtain:

```
> \text{minBound :: Int}
-2147483648
> \text{maxBound :: Int}
2147483647
```

---

\(^{3}\) from C.
1.2.6 Example

The following session gives an impression of the difference between Float and Double.

```
> 1.23456789012345678901234567890 :: Float
1.234568 :: Float
> 1.23456789012345678901234567890 :: Double
1.23456789012346 :: Double
```

1.2.7 Definition the standard Ratio number types

```
Ratio is made of Integral number pairs, its definition is a parameterized data type

data (Integral a) => Ratio a = a%a
```

And with Integral comprising two standard types, Ratio has two types as well:
(a) Ratio Int
(b) Ratio Integer

type Rational = Ratio Integer

(Note: that the Ratio module has to be imported/loaded in order to make full use of Ratio numbers.)

1.2.8 Example

Both, the numerator n and denumerator d in n%d have to be of the same type. In Hugs (and similar for the GHC interpreter) we have

```
Hugs> :load Ratio
{; or: module Ratio to import the module -}
```

```
Ratio> (123 :: Int) % (456 :: Int)
41 % 152 :: Ratio Int
```

Of course, instead of typing each component with say

```
Ratio> (123 :: Integer) % (456 :: Integer)
41 % 152 :: Ratio Integer
```

we may as well type it like this

```
Ratio> (123 % 456 :: Integer)
41 % 152 :: Ratio Integer
```

which is of course just type synonym for

```
Ratio> (123 % 456 :: Rational)
41 % 152 :: Rational
```

1.2.9 Definition the standard Complex number types

```
Complex is made of RealFloat number pairs, its definition is a parameterized data type

data (RealFloat a) => Complex a = a:+ a
```

And with RealFloat comprising two standard types, Complex has two types as well:
(a) Complex Float
(b) Complex Double

(Note, that the Complex module has to be imported/loaded.)

1.2.10 Example

To demonstrate the difference between the two standard Complex

```
1.23456789012345678901234567890 2 + 0.98765432109876543211 :: Complex Float
1.234568 2 + 0.98765433109876543211 :: Complex Double
```

1.2.11 Remark

Note the conservative choice of the names for the standard types:

(c) it preserves the legacy of C and its successor language:

<table>
<thead>
<tr>
<th>type in C</th>
<th>same type in Haskell</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>Int</td>
</tr>
<tr>
<td>float</td>
<td>Float</td>
</tr>
<tr>
<td>double</td>
<td>Double</td>
</tr>
</tbody>
</table>

(d) At least two types have the full potential of their counterparts in mathematics

<table>
<thead>
<tr>
<th>number system in mathematics</th>
<th>same type in Haskell</th>
</tr>
</thead>
<tbody>
<tr>
<td>the integers ( \mathbb{Z} )</td>
<td>Integer</td>
</tr>
<tr>
<td>the rational numbers ( \mathbb{Q} )</td>
<td>Rational</td>
</tr>
</tbody>
</table>

Also note the difference between the four sorts again:
(c) The composed sorts (Ratio a) and (Complex a) are data types, although with a parameter type a. These data types are well-defined by now.
(c) The primitive sorts (Integral a) and (RealFloat a) are actually more complicated type classes and their proper definition is still to come.

So by now we really need to turn from the lower type part of figure 2 to upper type class part.
1.2.12 Exercise

Suppose, we have two versions of a simple addition of positive integral numbers defined in identical fashion, but for the two integral types:

```haskell
addInt :: Int -> Int -> Int
addInt n m = if n < 0 then m else addInt (n-1) (m+1)

addInteger :: Integer -> Integer -> Integer
addInteger n m = if n < 0 then m else addInteger (n-1) (m+1)
```

In the Hugs and GHC interpreter, we can ask for additional information about the speed and memory for each call with the "set +s" command. Hugs answers with “reductions and cells” and GHC returns information about “seconds and bytes”. We can also enforce a type expression with each value output with the "set +t". An example dialog with Hugs is then given by

```haskell
> addInt 12345 12345
24691 :: Int
(209915 reductions, 281951 cells)
> addInteger 12345 12345
24691 :: Integer
(209915 reductions, 358082 cells)
```

Thus a (small) space advantage for the Int version, as expected. But the GHC interpreter shows the following behavior:

```haskell
> addInteger 123456 123456
246913 :: Integer
{ actually, this output line is two lines }
(0.36 secs, 22699272 bytes)
> addInt 123456 123456
246913 :: Int
{ again, the actual output comes in two lines }
(0.38 secs, 22696400 bytes)
```

Strangely and against all earlier reasoning, the Integer version is faster! *

*I don’t have an explanation for this behavior. If this is an example of a general pattern, it would contradict the whole justification for the distinction between the two Integral types so far.*
1.3 Type classes and type class instances

1.3.1 Introduction

Modern mathematics emerged with the understanding, that values such as numbers cannot be described appropriately as such, but only as elements of a structure or algebra\(^\text{a}\). It also turned out, that even a structure can hardly be defined as such. Instead, it is often described as a model or instance of a theory. An algebraic theory only specifies the signature of the algebra and a couple of rules, called axioms, that have to hold.

For example, the theory of rings defines a ring as a set \(R\) together with two constants 0 and 1 and functions \(+\), \(-\), \(\cdot\) \(R \times R \rightarrow R\), such that a couple of axioms have to hold (such as the associativity of \(+\) and \(-\), etc.). Many different, but important algebras are rings, i.e. they are models of this theory, in particular the integers together with their usual arithmetic operations.

Haskell has adopted this design, but here, theories are called type classes and their models are called instances. For example, there is a type class called Num and all eight standard numeric classes like Int, Rational or Complex Float are instances of Num.

1.3.2 Definition

A **type class** comprises

(a) A **class name**, which is a Haskell identifier with an initial capital letter (e.g. Eq, Ord, Show)

(b) A **signature**, comprising

   (i) a distinguished, but arbitrary type variable, say \(a\)

   (ii) some type declarations for values (constants, functions), based on already well-defined types and \(a\)

(c) Some **axioms or rules**, i.e. statements about the values that have to be satisfied.

The actual syntax for a type class definition of name \(C\) on type variable \(a\) is

```haskell
class C a where
  (- declarations -)
  (- axioms -)
```

This may be read as “type \(a\) is of class \(C\), if the following declarations are defined and axioms are satisfied”.

Suppose, a type class named \(C\) is defined as just described. An **instance** of \(C\) is an actual type \(T\) that takes the place of the variable \(a\). In general, this instantiation is done with an instance declaration and an explicit definition for all value declarations in the definition of \(C\). The general form to do that is

```haskell
instance C T where
  (- value definitions -)
```

For some type classes however, there is a shortcut. When \(T\) is defined as a data or newtype, Haskell can **derive** all the value definitions for \(T\) by putting a deriving statement immediately after the definition of \(T\):

```haskell
data T = ...

deriving C
```

But this only works for some standard Haskell type classes (namely Eq, Ord, Enum, Bounded, Read, and Show). In all other situations, all the values have to be defined explicitly in the instance declaration.

---

\(^a\)Many text books in mathematical logic and abstract algebra define a structure as a couple of carrier sets, together with a couple of functions and relations, defined on these sets. An algebraic structure or algebra is then a structure that has exactly one carrier set and no relations, but only functions (including constants, which are subsumed as nullary functions). In Haskell, there are no relations anyway. There, a relation say \(R : X \rightarrow Y\) takes the form \(R : X \times Y \rightarrow \text{Bool}\), or more often the curried version \(R : X \rightarrow Y \rightarrow \text{Bool}\).

---

\(^{10}\)see The Haskell Report, 6.3.1
1.3.4 Example an example Eq instance

Suppose, we define a type Binum (for binary numerals) a the data type

```haskell
data Binum = BIN [Bool]
```

Note, that Haskell has no default operation to test the equality of two values of the same data type. An input like

```haskell
> BIN [True, False] == BIN [True, False]
```

doesn’t answer with True or False, but with an error message. We really have to define the equality. The easy way to do this is the deriving statement immediately after the data type declaration:

```haskell
data Binum = BIN [Bool] deriving Eq
```

If we do that, then the two functions (==) and (/=) are available on Binum values as well, and we e.g. obtain as expected

```haskell
> BIN [True, False] == BIN [True, False]
True
```

At this point and for this example data type, the explicit declaration of a “trivial” function like the equality (==) may seem an unnecessary burden on the programmer. But a default equality definition would rather restrict our freedom and needs in many other cases. For example, consider two Rational numbers. We will certainly expect Haskell to do the following

```haskell
> 6%8 == 3%4
True
```

because 6/8 = 3/4. But a default equality would find the two values to be different. The same goes for two Float numbers, say 12.34e2 and 1.234E3, which ought to be equal.

Anyway, we can also turn Binum into an Eq instance by means of an explicit instance declaration. For example, by

```haskell
instance Eq NatBinum where
  BIN [] == BIN [] = True
  BIN [] == BIN _ = False
  BIN _ == BIN [] = False
  BIN (False:xL) == BIN (True:yL) = False
  BIN (True:xL) == BIN (False:yL) = False
  BIN (False:xL) == BIN (False:yL) = BIN xL == BIN yL
  BIN (True:xL) == BIN (True:yL) = BIN xL == BIN yL
```

Note, that we only need to define (==), because the second function (/=) of Eq is fully specified by the axioms, i.e. the class definition of Eq itself.

1.3.5 Example standard Eq instances

*** CONTINUE HERE ***

1.3.6 Remark

(1) type classes are algebraic theories, instances are models or algebras

(2) multiple type classes

(3) instances can only be defined by data or

---

**newtype**: alternative to the instance declaration

---

1.3.7 the Num type class

*** CONTINUE HERE ***
1.3.8 Introduction

Maybe the best way to understand type classes is a repetition of some basic concept of modern algebra. What has become a type class in Haskell is very much what is traditionally called an algebraic theory. The instances of type classes are very much the models in mathematical jargon.

1.3.9 Definition

An algebraic structure or algebra \( \mathfrak{A} \) is given by

\[
\mathfrak{A} = (A, c_1, \ldots, c_n, f_1, \ldots, f_m)
\]

where

1. \( A \) is the carrier set
2. the constants \( c_1, \ldots, c_n \) are distinguished elements of \( A \)
3. each of the functions \( f_1, \ldots, f_m \) is an ordinary function on \( A \), i.e. \( f_i : A \times \ldots \times A \rightarrow A \)

1.3.10 Example

(1) An algebra is given by \( \langle \mathbb{Z}, 0, 1, +, \cdot \rangle \), the integers \( \mathbb{Z} \) are the carrier set, there are two constants 0 and 1 and two binary functions \( + : \mathbb{Z} \times \mathbb{Z} \rightarrow \mathbb{Z} \) and \( \cdot : \mathbb{Z} \times \mathbb{Z} \rightarrow \mathbb{Z} \).

(2) \( \langle \mathbb{Z}, 0, +, \text{ negate} \rangle \) is also an algebra, where \( \mathbb{Z}, 0 \) and \( + \) as in (1), and \( \text{ negate} : \mathbb{Z} \rightarrow \mathbb{Z} \) is the unary function that turns each \( n \in \mathbb{Z} \) into \(-n\).

(3) \( \langle \mathbb{Z}, \leq \rangle \) is not an algebra, because in mathematics \( \leq \) is not a function, but a relation\(^{12}\)

1.3.11 Definition

An algebraic theory \( \mathcal{T} \) comprises

1. an (algebraic) signature, made of symbols \( A, c_1, \ldots, c_n, f_1, \ldots, f_m \), where
2. \( A \) is a carrier set symbol
3. the constants \( c_1, \ldots, c_n \) are symbols for constants
4. the \( f_1, \ldots, f_m \) are symbols for functions

and where each function symbol has an own type expression of the form

\[ f_i : A \times \ldots \times A \rightarrow A \]

5. A set of axioms, which are formulas on the given signature, making certain statements that are supposed to hold.

1.3.12 Example

Standard examples of algebraic theories are the following.

\( \star \) The theory of monoids.

The signature: A monoid has the form \( \langle M, e, \circ \rangle \), where \( M \) stands for a carrier set, \( e \) is a constant of \( M \), called the neutral element, and \( \circ : M \times M \rightarrow M \) is a binary function, usually written in infix notation.

The axioms:

1. The associativity axiom: \( \forall x, y, z \in M . (x \circ y) \circ z = x \circ (y \circ z) \)
2. The neutral element axiom: \( \forall x \in M . x \circ e = e \circ x = x \).

\( \star \) The theory of groups.

A group has the form *** CONTINUE HERE ***

\(^{12}\) A structure in general is made of carriers and constants, functions and relations on these carrier classes. An algebraic structure in particular is a structure with exactly one carrier class and no relations other than equality.
1.3.13 the standard numeric type classes and types

The whole hierarchy of type classes and types make a complex picture.

The two type classes Show and Read are excluded for now (see ??? below). They don’t belong to the mathematical aspect of the number system, but deal with the string conversion of numeral representation.

*** CONTINUE HERE ***
Figure 1: The number system of Haskell 98, as given in the Prelude, Ratio, Complex and Numeric modules
class Eq a where
  (==), (/=) :: a -> a -> Bool

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

class Num a where
  (+), (-), (*) :: a -> a -> a
  negate :: a -> a
  abs, signum :: a -> a
  fromIntegral :: Integer -> a
  subtract :: (Num a) => a -> a -> a

module Ratio

  data (Integral a) => Ratio a = a%a
  (%) :: (Integral a) => a -> a -> Ratio a
  numerator :: (Integral a) => Ratio a -> a
  denominator :: (Integral a) => Ratio a -> a
  approxRational :: (RealFrac a) => Ratio a -> a
  toRational :: a -> Rational

class RealFrac a where
  properFraction :: (Integral b) => a -> (b,a)
  truncate, round :: (Integral b) => a -> b
  ceiling, floor :: (Integral b) => a -> b
  realToFrac :: (Real a, Fractional b) => a -> b

class (Real a, Enum a) => Integral a where
  quot, rem, div, mod :: a -> a -> a
  quotRem, divMod :: a -> a -> (a,a)
  toInteger :: a -> Integer
  even :: (Integral a) => a -> Bool
  odd :: (Integral a) => a -> Bool
  gcd :: (Integral a) => a -> a -> a
  lcm :: (Integral a) => a -> a -> a
  (^) :: (Num a, Integral b) => a -> b -> a
  fromIntegral :: (Integral a, Num b) => a -> b

class Bounded a where
  minBound :: a
  maxBound :: a

class (Real a, Fractional a) => RealFrac a where
  floatRadix :: a -> Integer
  floatDigits :: a -> Int
  floatRange :: a -> (Int,Int)
  decodeFloat :: a -> (Integer,Int)
  encodeFloat :: Integer -> Int -> a
  exponent :: a -> Int
  significand :: a -> a
  scaleFloat :: Int -> a -> a
  isNaN, isInfinite, isDenormalized,
  isNegativeZero, IEEE :: a -> Bool
  atan2 :: a -> a -> a

module Complex

  data (RealFloat a) => Complex a = !a :+ !a
  realPart :: (RealFloat a) => Complex a -> a
  imagPart :: (RealFloat a) => Complex a -> a
  conjugate :: (RealFloat a) => Complex a -> Complex a
  mkPolar :: (RealFloat a) => a -> a -> Complex a
  cis :: (RealFloat a) => a -> Complex a
  polar :: (RealFloat a) => Complex a -> (a,a)
  magnitude :: (RealFloat a) => Complex a -> a
  phase :: (RealFloat a) => Complex a -> a

  data Float = Float Double

  data Double

module Numeric

  fromRat :: (RealFloat a) => Rational -> a
  floatToDigits :: (RealFloat a) => Integer -> a -> ([Int],Int)

  (^) :: (Fractional a, Integral b) => a -> b -> a
  numericEnumFrom :: (Fractional a) => a -> [a]
  numericEnumFromThen :: (Fractional a, Ord a) => a -> a -> [a]
  numericEnumFromThenTo :: (Fractional a, Ord a) => a -> a -> a -> [a]

Legend

⇒ denotes a type class dependency
→ is an instantiation
type ShowS = String -> String

class Show a where
  showsPrec :: Int -> a -> ShowS
  show :: a -> String
  showList :: [a] -> ShowS
  show :: (Show a) => a -> ShowS
  showChar :: Char -> ShowS
  showString :: String -> ShowS
  showParen :: Bool -> ShowS -> ShowS

instance Show Int where ...
instance Read Int where ...
instance Show Integer where ...
instance Read Integer where ...
instance Show Float where ...
instance Read Float where ...
instance Show Double where ...
instance Read Double where ...
instance Show () where ...
instance Read () where ...
instance Show Char where ...
instance Read Char where ...
instance (Read a) => Read [a] where ...
instance (Show a, Show b) => Show (a,b) where ...
instance (Read a, Read b) => Read (a,b) where ...

-- module Numeric
showSigned :: (Real a) => (a -> ShowS) -> Int -> a -> ShowS
showIntBase :: Integral a => a -> (Int -> Char) -> a -> ShowS

showSigned :: (Real a) => (a -> ShowS) -> Int -> a -> ShowS

showInt :: Integral a => a -> ShowS
showOct :: Integral a => a -> ShowS
showHex :: Integral a => a -> ShowS

readSigned :: (Real a) => ReadS a -> ReadS a

readInt :: (Integral a) => ReadS a
readDec :: (Integral a) => ReadS a
readOct :: (Integral a) => ReadS a
readHex :: (Integral a) => ReadS a

showSigned :: (Real Float a) => Maybe Int -> a -> ShowS

showFloat :: (RealFloat a) => (Real Float a) => Maybe Int -> a -> ShowS

showSigned :: (RealFloat a) => Maybe Int -> a -> ShowS

readFloat :: (RealFrac a) => ReadS a

lexDigits :: ReadS String
Figure 4: The syntax for integers and floating point numbers

An **integer literal** has the form `integer`, which is defined by the following grammar:

```plaintext
  digit    →  ascDigit | uniDigit
  ascDigit →  0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
  uniDigit →  any Unicode decimal digit
  octit    →  0 | 1 | 2 | 3 | 4 | 5 | 6 | 7
  hexit    →  0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F | a | b | c | d | e | f
  decimal  →  digit { digit }
  octal    →  octit { octid }
  hexadecimal  →  hexit { hexit }
  integer  →  decimal
             | 0o octal | 0O octal
             | 0x hexadecimal | 0X hexadecimal
```

An **float literal** has the form `float`, which is defined by the following grammar:

```plaintext
  digit    →  ascDigit | uniDigit
  ascDigit →  0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
  uniDigit →  any Unicode decimal digit
  decimal  →  digit { digit }
  float    →  decimal . decimal | exponent
              | decimal exponent
  exponent →  (0 | E) [ + | - ] decimal
```