Programming Languages
4. Introduction to Haskell: A Quick Note on Type Classes

Shin-Cheng Mu

Autumn 2025

1 Parametric Polymorphism in Haskell

The Type of $take$

- Recall the definition:

$$\begin{array}{lll} take\ 0\ xs & = & [\,] \\ take\ (1+n)\ [\,] & = & [\,] \\ take\ (1+n)\ (x : xs) & = & x : take\ n\ xs\ \ . \end{array}$$

- The first argument has to be of a numeric type (e.g. Int), since we pattern matched it against 0 and $1+$.
- The second argument must be a list, since we patten matched it against [ ] and (:).
- But the element of the list is not examined at all. It is merely copied to the output.

The Type of $take$

- The type of $take$ can be
  - $Int \to List\ Int \to List\ Int$;
  - $Int \to List\ Char \to List\ Char$, etc.
- There is a *most general* type: $Int \to List\ a \to List\ a$.
  - The small letter means that a is a type variable. One can imagine that there is an implicit $\forall$ that quantifies all type varaibles: $\forall a.Int \to List\ a \to List\ a$.

The Identity Function

- For a more obvious example, consider the (simple but important) identity function

$$id\ x = x\ \ .$$

- The argument is not touched at all.
- It may have type $Int \to Int$, $Char \to Char$, or even $(Int \to Int) \to (Int \to Int)$.
- The most general type is $a \to a$.

Filter

- Recall *filter*:

$$\begin{array}{lll} filter & :: & (a \to Bool) \to List\ a \to List\ a \\ filter\ p\ [\,] & = & [\,] \\ filter\ p\ (x : xs) & \mid & p\ x\ =\ x : filter\ p\ xs \\ & \mid & \mathbf{otherwise}\ =\ filter\ p\ xs\ \ . \end{array}$$

- Still, in $filter\ p\ (x : xs)$ we merely passes x to p, without looking into x.
- Therefore *filter* works for any type a for which there exists functions of type $a \to Bool$ — which is true for all type a.

Counting Lowercase Characters

- For a counterexample, consider the following function:

$$\begin{array}{lll} lowers & :: & List\ Char \to Int \\ lowers\ [\,] & = & 0 \\ lowers\ (x : xs) & = & \mathbf{if}\ isLower\ x \\ & & \quad \mathbf{then}\ 1 + lowers\ xs \\ & & \quad \mathbf{else}\ lowers\ xs\ \ . \end{array}$$

- The function counts the number of lowercase characters in a string.
- It is equivalent to $length \cdot filter\ isLower$.
- x is passed to $isLower$, which forces x to be a $Char$.

Parametric Polymorphism

- *Polymorphism*: allowing a piece of code to have many types, such that it can be used in many occasions.
  - Indeed, $take$ can be applied to all types of lists. We do not need to define a separate version for $List\ Int$, $List\ (Int \to Int)$.
- *Parametric* polymorphic, as we have seen just now, is common in many functional programming languages.
- When $take\ n :: List\ a \to List\ a$ is applied to an argument, say $[1, 2, 3]$, the type variable a is instantiated to the type of the argument (Int in this case).
  - The type variable a behaves like a parameter, thus the name.
  - Observe: the same piece of code (e.g. $take$, $filter$) works for all instantiations of a .
- Object-oriented languages often adopt another kind of polymorphism for operator overloading, called *ad-hoc* polymorphism, to be introduced later.

2 Type Classes

Membership Test

- Given the definition below, $elem\ x\ xs$ yields $True$ iff. x occurs in xs.

$$\begin{array}{lll} elem\ x\ [\,] & = & False \\ elem\ x\ (y : xs) & \mid & x == y\ =\ True \\ & \mid & \textbf{otherwise}\ =\ elem\ x\ xs\ . \end{array}$$

- It could have type $Int \to List\ Int \to Bool$, $Char \to List\ Char \to Bool$, etc.
- We do not want to define $elem$ once for each type, thus we wish that it has a polymorphic type, say $a \to List\ a \to Bool$.
- However, not all values can be tested for equality! The operator $(==)$ is defined for some types, but not all types. For example, we cannot in general decide whether two functions are equal.
- Thus $elem$ cannot have type, for example, $(Int \to Int) \to List\ (Int \to Int) \to Bool$.

2.1 Class and Instance Declarations

The Eq Class

- There is such a definition in the Standard Prelude:

$$\begin{array}{l} \textbf{class}\ Eq\ a\ \textbf{where} \\ \quad (==) :: a \to a \to Bool\ . \end{array}$$

- which says that a type a is in the *type class* Eq if there is an operator $(==)$, of type $a \to a \to Bool$, defined.
- Int is in Eq since we can define $(==)$ for numbers. So is $Char$, although $(==)$ for $Char$ implements a different algorithm from that of Int.

Type of $elem$

- The most general type of $elem$ is $Eq\ a \Rightarrow a \to List\ a \to Bool$,
  - which means that $elem$ takes a value of type a and a list of type $List\ a$ and returns a $Bool$, provided that a is in Eq.
- The additional constraint arises from the fact that $elem$ calls $(==)$.

Instance Declaration

- To use $elem$ on concrete types, we have to teach Haskell how to check equality for each type. The following are defined somewhere in the Haskell Prelude:

$$\begin{array}{l} \textbf{instance}\ Eq\ Int\ \textbf{where} \\ \quad m == n = \ \text{\{- how to check equality for } Int \text{ -\}} \\ \\ \textbf{instance}\ Eq\ Char\ \textbf{where} \\ \quad m == n = \ \text{\{- how to check equality for } Char \text{ -\}} \end{array}$$

- It is not possible to give a definition for, for example $Eq\ (a \to a)$. Thus $elem$ cannot be applied to such types.

Instance Declaration

- When we define a new type, we might want to teach Haskell how to check equality:

$$\textbf{data}\ Color\ =\ Red \mid Green \mid Blue \ldots$$

-

$$\begin{array}{l} \textbf{instance}\ Eq\ Color\ \textbf{where} \\ \quad Red == Red\ \ \ = True \\ \quad Red == Green\ = False \\ \ldots \end{array}$$

Summary So Far...

- Class declaration:

$$\textbf{class } Eq\ a \textbf{ where}$$
$$(==) :: a \to a \to Bool\ .$$

  - The *method* $(==)$ then has type $Eq\ a \Rightarrow a \to a \to Bool$.

- Instance declaration:

$$\textbf{instance } Eq\ MyType \textbf{ where}$$
$$x == y = \ldots$$

  - $(==)$ above should have type $MyType \to MyType \to Bool$, but the type is not written.

- A function that calls a function with class constraint $Eq\ a$ (e.g. $(==)$) also has the constraint in its type:

$$elem :: Eq\ a \Rightarrow a \to List\ a \to Bool$$
$$elem = \ldots == \ldots$$

- $elem\ \ 2\ \ [1,2,3]$ is allowed because there is an instance declaration for $Eq\ \ Int$, while $elem\ \ id\ \ [id,(1+),(2+)]$ is not (unless you define and instance $Eq\ \ (Int \to Int)$, which is usually not a good idea).

Ad-hoc Polymorphism

- Note that $(==)$ for Int is a different program from that for $Char$.
- Type classes is thus a way to describe *operator loading* — using one name to refer to different piece of code.
- Such mechanisms are often called *ad-hoc* polymorphism.
- Compare with parametric polymorphism, where the same code, say, the same definition of $take$, works for all types.

Other Important Type Classes

- $Show$: things that can be printed (converted to string).
- $Read$: things that can be parsed from strings.
- Num: things that behave like numbers (with addition, multiplication, etc).
- $Integral$: things that behave like integers.
- $Monad$, $Functor$...hope we will be able to talk about them later!
- Use `:i` in `GHCi` to find out what methods and instances each class has!

Derived Instances

- The Haskell compiler may automatically construct some routine instance declarations, to save you some typing. E.g.

$$\textbf{data } Colors = Red \mid Green \mid Blue$$
$$\textbf{deriving } (Eq, Show, Read)\ .$$

2.2 Instance Inheritance

Instance Inheritance

- How do we check whether two lists are equal? We can do so if we know how to check whether their elements are equal.

$$\begin{aligned}&\textbf{instance } Eq\ a \Rightarrow Eq\ (List\ a) \textbf{ where}\\ &[\,] == [\,] &&= True\\ &[\,] == (x : xs) &&= False\\ &(x : xs) == [\,] &&= False\\ &(x : xs) == (y : ys) &&= x == y\ \wedge\ xs == ys\ .\end{aligned}$$

- Note that in $x == y$, the $(==)$ refers to the method for type a, while the $(==)$ in $xs == ys$ is a recursive call.

Instance Inheritance

- Another example:

$$\textbf{instance } (Eq\ a, Eq\ b) \Rightarrow Eq\ (a, b) \textbf{ where}$$
$$(x_1, y_2) == (x_2, y_2)\ =\ (x_1 == x_2) \wedge (y_1 == y_2)\ .$$

- All the three $(==)$ in the expression above refer to different methods!

2.3 Class Hierarchy

The Type Class Ord

- Another type class Ord includes things are can be "ordered":

$$\textbf{class } Eq\ a \Rightarrow Ord\ a \textbf{ where}$$
$$(<) :: a \to a \to Bool$$
$$(\geq) :: a \to a \to Bool$$
$$(>) :: a \to a \to Bool$$
$$(\leq) :: a \to a \to Bool\ \ \ldots$$

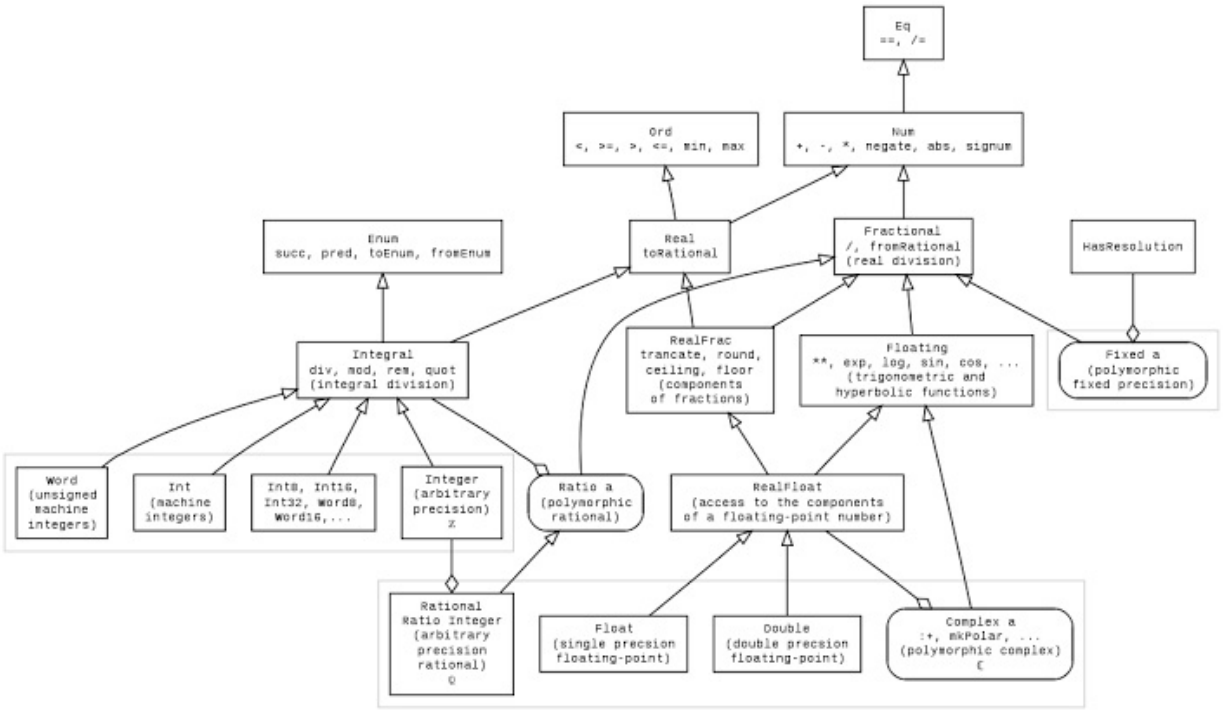

Figure 1: Standard Haskell Numerical Classes

- The declaration $Eq\ a \Rightarrow Ord\ a$ intends to mean that for a type a to be in class Ord it has to be in class Eq.
  - The methods $(<)$, $(\geq)$, etc, is allowed to use $(==)$.
  - Logically, it makes more sense to write $Eq\ a \Leftarrow Ord\ a$. But it's a historical mistake that has been made.
- The function $sort$ that sorts a list might have type $Ord\ a \Rightarrow List\ a \rightarrow List\ a$.

Class Hierarchy

- Inheritance between *type classes* are not to be confused with inheritance between *types*.
- Through inheritance, type classes form a hierarchy.
- Types in the standard Haskell Prelude form a complex hierarchy.
- Other libraries may extend the existing hierarchy or build their own hierarchy.

Notes

- The name "type class" is merely a mechanism for operator loading and shall not be confused with classes in object oriented languages.
- Type classes are an important feature of Haskell. Use of type classes has extended far beyond the inventors had imagined.
- We may use type classes in this course, but might not talk too much about their theoretical aspects, as they are orthogonal to the purpose of this course.