COMP4075: Lecture 6

Type Classes

Henrik Nilsson

University of Nottingham, UK

Type Classes

- Type classes is one of the distinguishing fetures of Haskell
- Introduced to make ad hoc polymorphism, or overloading, less ad hoc
- Promotes reuse, making code more readable
- Central to elimination of all kinds of "boiler-plate" code and sophisticated datatype-generic programming.

Key reason why many practitioners like Haskell: lots of "programming" can happen automatically!

Haskell Overloading (1)

What is the type of $(==)$?

E.g. the following both work:

$$1 == 2$$

$$\texttt{'a'} == \texttt{'b'}$$

I.e., $(==)$ can be used to compare both numbers and characters.

Maybe $(==) :: a \to a \to \mathit{Bool}$?

No!!! Cannot work uniformly for arbitrary types!

Haskell Overloading (2)

A function like the identity function

$$id :: a \to a$$
$$id\ x = x$$

is ***polymorphic*** precisely because it works uniformly for all types: there is no need to "inspect" the argument.

In contrast, to compare two "things" for equality, they very much have to be inspected, and an ***appropriate method of comparison*** needs to be used.

Haskell Overloading (3)

Moreover, some types do not in general admit a decidable equality. E.g. functions (when their domain is infinite).

Similar remarks apply to many other types. E.g.:

- We may want to be able to add numbers of any kind.
- But to add properly, we must understand what we are adding.
- Not every type admits addition.

Haskell Overloading (4)

Idea:

- Introduce the notion of a ***type class***: a set of types that support certain related operations.
- ***Constrain*** those operations to ***only*** work for types belonging to the corresponding class.
- Allow a type to be ***made an instance of*** (added to) a type class by providing ***type-specific implementations*** of the operations of the class.

The Type Class Eq

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

$(==)$ is not a function, but a ***method*** of the ***type class*** Eq. It's type signature is:

$$(==) :: Eq\ a \Rightarrow a \to a \to \mathit{Bool}$$

$Eq\ a$ is a ***class constraint***. It says that that the equality method works for any type belonging to the type class Eq.

Instances of Eq (1)

Various types can be made instances of a type class like Eq by providing implementations of the class methods for the type in question:

$$\textbf{instance}\ Eq\ \mathit{Int}\ \textbf{where}$$
$$\quad x == y = \mathit{primEqInt}\ x\ y$$
$$\textbf{instance}\ Eq\ \mathit{Char}\ \textbf{where}$$
$$\quad x == y = \mathit{primEqChar}\ x\ y$$

Instances of Eq (2)

Suppose we have a data type:

$$\textbf{data}\ \mathit{Answer} = \mathit{Yes} \mid \mathit{No} \mid \mathit{Unknown}$$

We can make Answer an instance of Eq as follows:

$$\textbf{instance}\ Eq\ \mathit{Answer}\ \textbf{where}$$
$$\begin{array}{llll} \quad \mathit{Yes} & == \mathit{Yes} & = \mathit{True} \\ \quad \mathit{No} & == \mathit{No} & = \mathit{True} \\ \quad \mathit{Unknown} & == \mathit{Unknown} & = \mathit{True} \\ \quad \_ & == \_ & = \mathit{False} \end{array}$$

Instances of Eq (3)

Consider:

$$\begin{aligned}&\mathbf{data}\ Tree\ a = Leaf\ a\\ &\qquad\qquad\quad \mid Node\ (Tree\ a)\ (Tree\ a)\end{aligned}$$

Can $Tree$ be made an instance of Eq?

Instances of Eq (4)

Yes, for any type a that is already an instance of Eq:

$$\begin{aligned}&\mathbf{instance}\ (Eq\ a) \Rightarrow Eq\ (Tree\ a)\ \mathbf{where}\\ &\quad Leaf\ a1 \quad\ == Leaf\ a2 \quad\ = a1 == a2\\ &\quad Node\ t1l\ t1r == Node\ t2l\ t2r = t1l == t2l\\ &\qquad\qquad\qquad\qquad\qquad\qquad\quad \&\&\ t1r == t2r\\ &\quad \_ \qquad\qquad == \_ \qquad\qquad = False\end{aligned}$$

Note that $(==)$ is used at type a (whatever that is) when comparing $a1$ and $a2$, while the use of $(==)$ for comparing subtrees is a recursive call.

Derived Instances (1)

Instance declarations are often obvious and mechanical. Thus, for certain ***built-in*** classes (notably Eq, Ord, $Show$), Haskell provides a way to ***automatically derive*** instances, as long as

- the data type is sufficiently simple
- we are happy with the standard definitions

Thus, we can do:

$$\begin{aligned}&\mathbf{data}\ Tree\ a = Leaf\ a\\ &\qquad\qquad\quad \mid Node\ (Tree\ a)\ (Tree\ a)\\ &\qquad\quad \mathbf{deriving}\ Eq\end{aligned}$$

Derived Instances (2)

GHC provides ***many*** additional possibilities. With the extension `-XGeneralizedNewtypeDeriving`, a new type defined using **newtype** can "inherit" any of the instances of the representation type:

$$\mathbf{newtype}\ Time = Time\ Int\ \mathbf{deriving}\ Num$$

With the extension `-XStandaloneDeriving`, instances can be derived separately from a type definition (even in a separate module):

$$\begin{aligned}&\mathbf{deriving\ instance}\ Eq\ Time\\ &\mathbf{deriving\ instance}\ Eq\ a \Rightarrow Eq\ (Tree\ a)\end{aligned}$$

Class Hierarchy

Type classes form a hierarchy. E.g.:

$$\begin{aligned}&\mathbf{class}\ Eq\ a \Rightarrow Ord\ a\ \mathbf{where}\\ &\quad (<=) :: a \to a \to Bool\\ &\quad \ldots\end{aligned}$$

Eq is a superclass of Ord; i.e., any type in Ord must also be in Eq.

Haskell vs. OO Overloading (1)

A method, or overloaded function, may thus be understood as a family of functions where the right one is chosen depending on the types.

A bit like OO languages like Java. But the underlying mechanism is quite different and much more general. Consider `read`:

$$read :: (Read\ a) \Rightarrow String \to a$$

Note: overloaded on the ***result*** type! A method that converts from a string to ***any*** other type in class `Read`!

Haskell vs. OO Overloading (2)

$$\begin{aligned}&> \mathbf{let}\ xs = [1,2,3] :: [Int]\\ &> \mathbf{let}\ ys = [1,2,3] :: [Double]\\ &> xs\\ &[1,2,3]\\ &> ys\\ &[1.0,2.0,3.0]\\ &> (read\ \texttt{"42"} : xs)\\ &[42,1,2,3]\\ &> (read\ \texttt{"42"} : ys)\\ &[42.0,1.0,2.0,3.0]\end{aligned}$$

Haskell vs. OO Overloading (3)

Taking Java as a typical OO example:

- ***Classes*** and ***interfaces*** define sets of methods that elements of a type must support.
- Through ***generics***, classes can be parametrised on types that can be bounded by classes and interfaces, a little like constraints in Haskell's class/instance declarations.
- However, the overloading is always on the ***object***; i.e. effectively the ***first argument*** to a method:

```
object.method(arg1, arg2, ...)
```


Implementation (1)

The class constraints represent extra implicit arguments that are filled in by the compiler. These arguments are (roughly) the functions to use.

Thus, internally $(==)$ is a ***higher order function*** with ***three*** arguments:

$$(==)\ eqF\ x\ y = eqF\ x\ y$$

Implementation (2)

An expression like

$$1 == 2$$

is essentially translated into

$$(==)\ primEqInt\ 1\ 2$$

Implementation (3)

So one way of understanding a type like

$$(==) :: Eq\ a \Rightarrow a \to a \to Bool$$

is that $Eq\ a$ corresponds to an extra implicit argument.

The implicit argument corresponds to a so called directory, or tuple/record of functions, one for each method of the type class in question.

Implementation (4)

A rough illustration of the idea:

$$\begin{array}{l} \textbf{class}\ Foo\ a\ \textbf{where} \\ \quad fie :: a \to Bool \\ \quad fum :: a \to Int \end{array}$$

The types of methods fie and fum:

$$\begin{array}{l} fie :: Foo\ a \Rightarrow a \to Bool \\ fum :: Foo\ a \Rightarrow a \to Int \end{array}$$

Implementation (5)

As Foo have two methods, the dictionary needs to carry two functions. If a pair were to be used for this purpose, the actual implementations would be something along the lines:

$$\begin{array}{l} fie :: (a \to Bool, a \to Int) \to a \to Bool \\ fie\ dict\ x = (fst\ dict)\ x \end{array}$$

$$\begin{array}{l} fie :: (a \to Bool, a \to Int) \to a \to Bool \\ fie\ dict\ x = (snd\ dict)\ x \end{array}$$

Some Basic Haskell Classes (1)

$$\begin{array}{l} \textbf{class}\ Eq\ a\ \textbf{where} \\ \quad (==), (/=) :: a \to a \to Bool \\ \textbf{class}\ (Eq\ a) \Rightarrow Ord\ a\ \textbf{where} \\ \quad compare\ \ :: a \to a \to Ordering \\ \quad (<), (<=), (>=), (>) :: a \to a \to Bool \\ \quad max, min :: a \to a \to a \\ \textbf{class}\ Show\ a\ \textbf{where} \\ \quad show :: a \to String \end{array}$$

Some Basic Haskell Classes (2)

$$\begin{array}{l} \textbf{class}\ Num\ a\ \textbf{where} \\ \quad (+), (-), (*) :: a \to a \to a \\ \quad negate\ \ \ \ \ :: a \to a \\ \quad abs, signum :: a \to a \\ \quad fromInteger :: Integer \to a \end{array}$$

$$\begin{array}{l} \textbf{class}\ Num\ a \Rightarrow Fractional\ a\ \textbf{where} \\ \quad (/)\ \ \ :: a \to a \to a \\ \quad recip :: a \to a \\ \quad fromRational :: Rational \to a \end{array}$$

Some Basic Haskell Classes (3)

Quiz: What is the type of a numeric literal like 42? What about 1.23? Why?

Haskell's numeric literals are overloaded:

- 42 means $fromInteger\ 42$
- 1.23 means $fromRational\ (133\ \%\ 100)$

Thus:

$$\begin{array}{l} 42\ \ \ :: Num\ a \Rightarrow a \\ 1.23 :: Fractional\ a \Rightarrow a \end{array}$$

A Typing Conundrum (1)

Overloaded (numeric) literals can lead to some surprises.

What is the type of the following list? Is it even well-typed???

$$[1, [2, 3]]$$

Surprisingly, it is well-typed:

$$\begin{array}{l} > \textbf{:type}\ [1, [2, 3]] \\ [1, [2, 3]] :: (Num\ [t], Num\ t) \Rightarrow [[t]] \end{array}$$

Why?

A Typing Conundrum (2)

The list is expanded into:

$$\begin{array}{l} [fromInteger\ 1, \\ \ [fromInteger\ 2, fromInteger\ 3]] \end{array}$$

Thus, if there were some type t for which $[t]$ were an instance of Num, the 1 would be an overloaded literal of that type, matching the type of the second element of the list.

Normally there are no such instances, so what almost certainly is a mistake will be caught. But the error message is rather confusing.

Multi-parameter Type Classes

GHC supports an extension to allow a class to have more than one parameter; i.e., defining a ***relation*** on types rather than just a predicate:

$$\mathbf{class}\ C\ a\ b\ \mathbf{where}\ldots$$

This often lead to type inference ambiguities. Can be addressed through ***functional dependencies***:

$$\mathbf{class}\ \mathit{StateMonad}\ s\ m \mid m \to s\ \mathbf{where}\ldots$$

This enforces that all instances will be such that m uniquely determines s.

Application: Automatic Differentiation

- ***Automatic Differentiation***: method for augmenting code so that derivative(s) computed along with main result.
- Purely algebraic method: arbitrary code can be handled
- Exact results
- But no separate, self-contained representation of the derivative.

Automatic Differentiation: Key Idea

Consider a code fragment:

$$\begin{aligned} z1 &= x + y \\ z2 &= x * z1 \end{aligned}$$

Suppose x' and y' are the derivatives of x and y w.r.t. a common variable. Then the code can be augmented to compute the derivatives of $z1$ and $z2$:

$$\begin{aligned} z1 &= x + y \\ z1' &= x' + y' \\ z2 &= x * z1 \\ z2' &= x' * z1 + x * z1' \end{aligned}$$

Approaches

- Source-to-source translation
- Overloading of arithmetic operators and mathematical functions

The following variation is due to Jerzy Karczmarczuk. Infinite list of derivatives allows derivatives of ***arbitrary*** order to be computed.

Functional Automatic Differentiation (1)

Introduce a new numeric type C: value of a continuously differentiable function at a point along with ***all*** derivatives at that point:

$$\begin{aligned} &\mathbf{data}\ C = C\ \mathit{Double}\ C \\ &\mathit{valC}\ (C\ a\ \_) \ = a \\ &\mathit{derC}\ (C\ \_\ x') = x' \end{aligned}$$

Functional Automatic Differentiation (2)

Constants and the variable of differentiation:

$$\begin{aligned} &\mathit{zeroC} :: C \\ &\mathit{zeroC} = C\ 0.0\ \mathit{zeroC} \\ &\mathit{constC} :: \mathit{Double} \to C \\ &\mathit{constC}\ a = C\ a\ \mathit{zeroC} \\ &\mathit{dVarC} :: \mathit{Double} \to C \\ &\mathit{dVarC}\ a = C\ a\ (\mathit{constC}\ 1.0) \end{aligned}$$

Functional Automatic Differentiation (3)

Part of numerical instance:

$$\begin{aligned} &\mathbf{instance}\ \mathit{Num}\ C\ \mathbf{where} \\ &\quad (C\ a\ x') + (C\ b\ y') = C\ (a + b)\ (x' + y') \\ &\quad (C\ a\ x') - (C\ b\ y') = C\ (a - b)\ (x' - y') \\ &\quad x@(C\ a\ x') * y@(C\ b\ y') = \\ &\quad\quad C\ (a * b)\ (x' * y + x * y') \\ &\quad \mathit{fromInteger}\ n = \mathit{constC}\ (\mathit{fromInteger}\ n) \end{aligned}$$

Functional Automatic Differentiation (4)

Computation of $y = 3t^2 + 7$ at $t = 2$:

$$\begin{aligned} t &= \mathit{dVarC}\ 2 \\ y &= 3 * t * t + 7 \end{aligned}$$

We can now get whichever derivatives we need:

$$\begin{aligned} &\mathit{valC}\ y &&\Rightarrow\ 19.0 \\ &\mathit{valC}\ (\mathit{derC}\ y) &&\Rightarrow\ 12.0 \\ &\mathit{valC}\ (\mathit{derC}\ (\mathit{derC}\ y)) &&\Rightarrow\ 6.0 \\ &\mathit{valC}\ (\mathit{derC}\ (\mathit{derC}\ (\mathit{derC}\ y))) &&\Rightarrow\ 0.0 \end{aligned}$$

Functional Automatic Differentiation (5)

Of course, we're not limited to picking just one point. Let tvals be a list of points of interest:

$$[3 * t * t + 7 \mid \mathit{tval} \leftarrow \mathit{tvals}, \mathbf{let}\ t = \mathit{dVarC}\ \mathit{tval}]$$

Or we can define a function:

$$\begin{aligned} &y :: \mathit{Double} \to C \\ &y\ \mathit{tval} = 3 * t * t + 7 \\ &\quad \mathbf{where} \\ &\quad\quad t = \mathit{dVarC}\ \mathit{tval} \end{aligned}$$

Reading

- Jerzy Karczmarczuk. Functional differentiation of computer programs. *Higher-Order and Symbolic Computation*, 14(1):35–57, March 2001.