COMP4075/G54RFP: Lecture 5

Type Classes

Henrik Nilsson

University of Nottingham, UK

Type Classes

- Type classes is one of the distinguishing fetures of Haskell

Type Classes

- Type classes is one of the distinguishing fetures of Haskell
- Introduced to make ad hoc polymorphism, or overloading, less ad hoc

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

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.

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.

Haskell Overloading (1)

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

E.g. the following both work:

$$1 == 2$$

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

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

Maybe $(==) :: a \to a \to Bool$?

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 $(==) :: a \rightarrow a \rightarrow Bool$?

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.

Haskell Overloading (2)

A function like the identity function

$$
\begin{aligned}
id &:: a \to a \\
id\ x &= x
\end{aligned}
$$

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).

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.:

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.

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.

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:

Haskell Overloading (4)

Idea:

- Introduce the notion of a ***type class***: a set of types that support certain related operations.

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.

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

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

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

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

$Eq\ a$ 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:

$$
\begin{array}{l}
\textbf{instance } Eq\ Int \textbf{ where} \\
\quad x == y = primEqInt\ x\ y \\
\textbf{instance } Eq\ Char \textbf{ where} \\
\quad x == y = primEqChar\ x\ y
\end{array}
$$

Instances of Eq (2)

Suppose we have a data type:

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

We can make $Answer$ an instance of Eq as follows:

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

Instances of Eq (3)

Consider:

$$
\begin{aligned}
\textbf{data}\ \mathit{Tree}\ a = &\ \mathit{Leaf}\ a \\
\mid &\ \mathit{Node}\ (\mathit{Tree}\ a)\ (\mathit{Tree}\ a)
\end{aligned}
$$

Can Tree be made an instance of Eq?

Instances of Eq (4)

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

$$
\begin{array}{llll}
\multicolumn{4}{l}{\textbf{instance}\ (Eq\ a) \Rightarrow Eq\ (Tree\ a)\ \textbf{where}} \\
\quad Leaf\ a1 & == Leaf\ a2 & = & a1 == a2 \\
\quad Node\ t1l\ t1r & == Node\ t2l\ t2r & = & t1l == t2l \\
 & & & \quad \&\&\ t1r == t2r \\
\quad \_ & == \_ & = & False
\end{array}
$$

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:

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

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:

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

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:

$$\textbf{newtype}\ \mathit{Time} = \mathit{Time}\ \mathit{Int}\ \textbf{deriving}\ \mathit{Num}$$

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

$$\textbf{deriving instance}\ \mathit{Eq}\ \mathit{Time}$$
$$\textbf{deriving instance}\ \mathit{Eq}\ a \Rightarrow \mathit{Eq}\ (\mathit{Tree}\ a)$$

Class Hierarchy

Type classes form a hierarchy. E.g.:

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

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`:

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

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)

$> \textbf{let } xs = [1,2,3] :: [Int]$

$> \textbf{let } ys = [1,2,3] :: [Double]$

$> xs$

$[1,2,3]$

$> ys$

$[1.0,2.0,3.0]$

$> (read \texttt{"42"} : xs)$

$[42,1,2,3]$

$> (read \texttt{"42"} : ys)$

$[42.0,1.0,2.0,3.0]$

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

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

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.

Some Basic Haskell Classes (1)

$$\textbf{class}\ Eq\ a\ \textbf{where}$$

$$(==), (/=) :: a \to a \to Bool$$

$$\textbf{class}\ (Eq\ a) \Rightarrow Ord\ a\ \textbf{where}$$

$$compare\ :: a \to a \to Ordering$$

$$(<), (<=), (>=), (>) :: a \to a \to Bool$$

$$max, min :: a \to a \to a$$

$$\textbf{class}\ Show\ a\ \textbf{where}$$

$$show :: a \to String$$

Some Basic Haskell Classes (2)

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

Some Basic Haskell Classes (2)

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

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

Some Basic Haskell Classes (3)

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

Some Basic Haskell Classes (3)

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

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

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

Thus:

$$\begin{aligned} 42 &:: \mathit{Num}\ a \Rightarrow a \\ 1.23 &:: \mathit{Fractional}\ a \Rightarrow a \end{aligned}$$

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???

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

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:

$$> \text{:}\mathbf{type}\ [1, [2, 3]]$$
$$[1, [2, 3]] :: (Num\ [t], Num\ t) \Rightarrow [[t]]$$

Why?

A Typing Conundrum (2)

The list is expanded into:

$$[\mathit{fromInteger}\ 1,\\ [\mathit{fromInteger}\ 2, \mathit{fromInteger}\ 3]]$$

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.

A Typing Conundrum (2)

The list is expanded into:

$$[\mathit{fromInteger}\ 1,$$
$$\ [\mathit{fromInteger}\ 2, \mathit{fromInteger}\ 3]]$$

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:

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

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:

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

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

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

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:

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

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:

$$\mathbf{data}\ C = C\ \mathit{Double}\ C$$

$$\mathit{valC}\ (C\ \ a\ \_)\ = a$$
$$\mathit{derC}\ (C\ \_\ x') = x'$$

Functional Automatic Differentiation (2)

Constants and the variable of differentiation:

$$zeroC :: C$$
$$zeroC = C\ 0.0\ zeroC$$

$$constC :: Double \to C$$
$$constC\ \ a = C\ \ a\ zeroC$$

$$dVarC :: Double \to C$$
$$dVarC\ \ a = C\ \ a\ (constC\ 1.0)$$

Functional Automatic Differentiation (3)

Part of numerical instance:

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

Functional Automatic Differentiation (4)

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

$$t = dVarC\ 2$$
$$y = 3 * t * t + 7$$

We can now get whichever derivatives we need:

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

Functional Automatic Differentiation (5)

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

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

Functional Automatic Differentiation (5)

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

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

Or we can define a function:

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

Reading

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