Haskell’s type classes

Ralf Lämmel
Software Language Engineer
University of Koblenz-Landau
Germany

Thanks to Oleg Kiselyov, Klaus Ostermann, Peter Thiemann and Stefan Wehr for joint work on this subject.
A type-class primer
A standard type class

Formal type parameter for instantiating type

class Eq a where
  (==) :: a -> a -> Bool
Let's define equality for expressions.

```haskell
data Expr = Const Int
           | Add Expr Expr
```
A type-class instance

instance Eq Expr where

(\text{Const } _) == (\text{Add } _) _ = \text{False}

(\text{Add } _) _ == (\text{Const } _) _ = \text{False}

(\text{Const } i) == (\text{Const } i') = i == i'

(\text{Add } x y) == (\text{Add } x' y') =

\quad x == x' \&\& y == y'

The (==) function is defined by pattern matching.
The full Eq class

class Eq a
  where
    (==), (=/=) :: a -> a -> Bool
    x/=y = not (x==y)
    x==y = not (x/=y)

Either of (==) or (=/=) is sufficient for a complete definition.
Another type class

class Show a
  where
    show :: a -> String
    ...

instance Show Expr
  where
    show (Const i) = "Const " ++ show i
    show (Add x y) = "Add" ++ f x ++ f y
    where
      f x = " (" ++ show x ++ ")"
Types with constraints

No constraint = parametric polymorphism

Constraint on actual type parameter = type-class polymorphism
Types with constraints

> :t filter  
(a -> Bool) -> [a] -> [a]

> :t \a -> filter (a/=)  
(Eq a) => a -> [a] -> [a]
Type classes vs. interfaces
<table>
<thead>
<tr>
<th>C#/Java concept</th>
<th>Haskell concept</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>—</td>
</tr>
<tr>
<td>Interface</td>
<td>Type class</td>
</tr>
<tr>
<td>Interface member</td>
<td>Type-class member</td>
</tr>
<tr>
<td>Interface implementation</td>
<td>Type-class instance</td>
</tr>
</tbody>
</table>
Specifics of type classes when compared to C#/Java-like interfaces

- **Retroactive implementation**
- Explicit reference to implementing type
  - Multiple references (“binary methods”)
  - Reference in result position (“static methods”)
- Default implementations of members
- Multiple type parameters
- ...
Different kinds of “methods”

> :t show
(Show a) => a -> String

> :t read
(Read a) => String -> a

> :t (==)
(Eq a) => a -> a -> Bool
Let’s solve the expression problem with open datatypes and open functions.
Point of reference: the **closed datatype**

```haskell
data Expr = Const Int
          | Add Expr Expr
```

Note that there are two constructors; one of them involves recursive references.
Point of reference: the **closed function**

evaluate :: Expr -> Int
evaluate (Const i) = i
evaluate (Add l r) =
    evaluate l + evaluate r

Note that there is one equation per datatype constructor, and there are recursive function applications.
The **open datatype**

One datatype per original constructor

```haskell
data Const = Const Int

data Add l r = Add l r
```

```haskell
class Expr x

instance Expr Const

instance (Expr l, Expr r) => Expr (Add l r)
```

A type class for the original datatype
The **open function** (type-class declaration)

```
class Expr x => Evaluate x
where
  evaluate :: x -> Int
```

A super-class constraint
The **open function**
(type-class instances)

```haskell
instance Evaluate Const
    where
        evaluate (Const i) = i

instance (Evaluate l, Evaluate r) => Evaluate (Add l r)
    where
        evaluate (Add l r) = evaluate l + evaluate r
```

**Constraints for recursive calls**
A data extension

data Expr x => Neg x = Neg x

instance Expr x => Expr (Neg x)

instance Evaluate x => Evaluate (Neg x)
where
  evaluate (Neg x) = 0 - evaluate x

3 steps:
- Declare a designated datatype for the data variant.
- Instantiate the type class for the open datatype.
- Instantiate all type classes for existing operations.
Multi-parameter type classes: from sets of types (with common operations) to relations on types
A programming scenario: shapes and intersection
Point of reference:
the **closed datatype**

```haskell
data Shape =
    Square    { x,y :: Int, length :: Int }
| Rectangle { x,y :: Int, height,width :: Int }
| Circle    { x,y :: Int, radius :: Float }
| Ellipse   { x,y :: Int, major,minor :: Float }
```

Suppose we want to be extensible with regard to shapes.
Point of reference: 
the closed function

intersect :: Shape -> Shape -> Bool

intersect (Square x y l)      (Square x' y' l')        = ...
intersect (Rectangle x y h w) (Rectangle  x' y' h' w') = ...
intersect (Circle x y r)      (Circle x' y' r')        = ...
intersect (Ellipse x y a i)   (Ellipse x' y' a' i')    = ...
intersect (Square x y l)      (Rectangle x' y' h w)    = ...

There are as many equations as there are combinations of forms of shape.
The **open datatype**

```haskell
data Square    = Square    Int Int Int
data Rectangle = Rectangle Int Int Int Int
data Circle    = Circle    Int Int Float
data Ellipse   = Ellipse   Int Int Float Float

class Shape x
instance Shape Square
instance Shape Rectangle
instance Shape Circle
instance Shape Ellipse
```
The open function

Wow!
Type classes may have multiple type parameters.

class (Shape x, Shape y) => Intersect x y
  where
  intersect :: x -> y -> Bool

instance Intersect Square Square
  where
  intersect s s' = ...

instance Intersect Rectangle Rectangle
  where
  intersect r r' = ...

Exercise: Fill in the “…”!
Functional dependencies: from *relations* on types (with common operations) to *functions* on types
What if the result type depends on the argument type(s)?

Consider an operation \textit{normalize}.

\begin{center}
\begin{tikzpicture}
\begin{scope}[scale=0.5, every node/.style={scale=0.5}]
\node (a) at (0.5,0.5) [circle, fill=white] {0};
\node (b) at (1,0.5) [circle, fill=white] {0};
\node (c) at (1,0.75) [circle, fill=white] {0};
\node (d) at (1,1) [rectangle, fill=white] {0};
\node (e) at (1.5,0.5) [circle, fill=white] {0};
\node (f) at (2,0.5) [circle, fill=white] {0};
\node (g) at (2,0.75) [circle, fill=white] {0};
\node (h) at (2,1) [rectangle, fill=white] {0};
\node (i) at (2.5,0.5) [circle, fill=white] {0};
\node (j) at (3,0.5) [circle, fill=white] {0};
\node (k) at (3,0.75) [circle, fill=white] {0};
\node (l) at (3,1) [rectangle, fill=white] {0};
\node (m) at (3.5,0.5) [circle, fill=white] {0};
\node (n) at (4,0.5) [circle, fill=white] {0};
\node (o) at (4,0.75) [circle, fill=white] {0};
\node (p) at (4,1) [rectangle, fill=white] {0};
\node (q) at (4.5,0.5) [circle, fill=white] {0};
\node (r) at (5,0.5) [circle, fill=white] {0};
\node (s) at (5,0.75) [circle, fill=white] {0};
\node (t) at (5,1) [rectangle, fill=white] {0};
\node (u) at (5.5,0.5) [circle, fill=white] {0};
\node (v) at (6,0.5) [circle, fill=white] {0};
\node (w) at (6,0.75) [circle, fill=white] {0};
\node (x) at (6,1) [rectangle, fill=white] {0};
\node (y) at (6.5,0.5) [circle, fill=white] {0};
\node (z) at (7,0.5) [circle, fill=white] {0};
\node (aa) at (7,0.75) [circle, fill=white] {0};
\node (bb) at (7,1) [rectangle, fill=white] {0};
\node (cc) at (7.5,0.5) [circle, fill=white] {0};
\node (dd) at (8,0.5) [circle, fill=white] {0};
\node (ee) at (8,0.75) [circle, fill=white] {0};
\node (ff) at (8,1) [rectangle, fill=white] {0};
\node (gg) at (8.5,0.5) [circle, fill=white] {0};
\node (hh) at (9,0.5) [circle, fill=white] {0};
\node (ii) at (9,0.75) [circle, fill=white] {0};
\node (jj) at (9,1) [rectangle, fill=white] {0};\end{scope}
\node at (4.5,3) {Preserve area and origin!};
\draw[->, line width=1.2mm] (a) -- (b);
\draw[->, line width=1.2mm] (c) -- (d);
\draw[->, line width=1.2mm] (e) -- (f);
\draw[->, line width=1.2mm] (g) -- (h);
\draw[->, line width=1.2mm] (i) -- (j);
\draw[->, line width=1.2mm] (k) -- (l);
\draw[->, line width=1.2mm] (m) -- (n);
\draw[->, line width=1.2mm] (o) -- (p);
\draw[->, line width=1.2mm] (q) -- (r);
\draw[->, line width=1.2mm] (s) -- (t);
\draw[->, line width=1.2mm] (u) -- (v);
\draw[->, line width=1.2mm] (w) -- (x);
\draw[->, line width=1.2mm] (y) -- (z);
\draw[->, line width=1.2mm] (aa) -- (bb);
\draw[->, line width=1.2mm] (cc) -- (dd);
\draw[->, line width=1.2mm] (ee) -- (ff);
\end{tikzpicture}
\end{center}
Point of reference: the **closed function**

\[
\text{normalize} :: \text{Shape} \rightarrow \text{Shape} \\
\text{normalize } s@(\text{Square } _ _ _ _) = s \\
\text{normalize } (\text{Rectangle } x \ y \ h \ w) = \text{Square } \ldots \\
\text{normalize } c@(\text{Circle } _ _ _ _) = c \\
\text{normalize } (\text{Ellipse } x \ y \ a \ i) = \text{Circle } \ldots
\]
The **open datatype** for normal shapes

class Shape s => NormalShape s

instance NormalShape Square

instance NormalShape Circle

“A normal shape is a shape.”
The open function for normalization

class (Shape s1, NormalShape s2)
    => Normalize s1 s2

    where
    normalize :: s1 -> s2

instance Normalize Square Square
    where
    normalize = id

instance Normalize Circle Circle
    where
    normalize = id

instance Normalize Rectangle Square where ...
instance Normalize Ellipse Circle where ...
A weird type error

> normalize (Square 1 2 3)
Type error!

> normalize (Square 1 2 3) :: Square
Square 1 2 3

Why do we need to specify the result type? There is only one instance with argument type Square!
A hypothetical program

Instances at compile time of the expression

\begin{verbatim}
instance Normalize Square    Square where ... 
instance Normalize Circle    Circle where ... 
instance Normalize Rectangle Square where ... 
instance Normalize Ellipse   Circle where ... 
\end{verbatim}

An instance in a module that is compiled later

\begin{verbatim}
instance Normalize Square    Circle where ... 
\end{verbatim}
Type classes with functional dependencies

class (Shape s1, NormalShape s2) => Normalize s1 s2
  | s1 -> s2

where

normalize :: s1 -> s2

> normalize (Square 1 2 3)
Square 1 2 3
Further reading

- JavaGI (Wehr et al., ECOOP 2007; see also Wehr’s PhD thesis)
- Haskell’s type classes (Lämmel, Ostermann, GPCE 2006)
- Open data types and functions (Löh and Hinze, PPDP 2006)
- Fun with Type Functions (Kiselyov et al., May 2010)
- Language support for generic programming (Garcia et al., JFP 2007)
- Multiple dispatch in Multijava (Clifton et al., ACM TOPLAS 2006)
- Multimethods à la Clojure
- ...
Nifty issues

• Scrap your boilerplate code
• Equality on open data
• Construct open data
• Over-precise open types
• Heterogenous lists
• ...

Let’s use riddles for explanation.
A riddle on instance derivation (“scrap your boilerplate” code)

Derive such instances automatically.

```haskell
data Expr = Const Int
           | Add Expr Expr

deriving (Eq, Show, Read)
```

How to implement other **generic** operations once and for all?
A riddle on open equality

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

This type class cannot work for open datatypes since, in general, values of an open datatype can be of different Haskell types. How do we recover?
A riddle on open data construction

read :: (Read a) => String -> a

Now suppose you instantiate the Read type class for the different data variants of the open datatype Expr. How would you read an arbitrary expression?
A riddle on type overprecision

> let n1 = Const 1
> let n40 = Const 40
> let n42 = Add (Add n1 n1) n40
> :t n42

n42 :: Add (Add Const Const) Const

Isn’t the type a bit too precise? The type resembles the structure of the value!
A riddle on heterogenous lists: intersection for a list of shapes

intersectMany :: [Shape] -> Bool
intersectMany [] = False
intersectMany (x:[]) = False
intersectMany (x:y:z) =
    intersect x y
    || intersectMany (x:z)
    || intersectMany (y:z)

How to do such an operation with an open datatype? More specifically, what’s the type of intersect?
Thanks!
Questions and comments welcome.