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.

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

A type-class instance

```
instance Eq Expr where
 (Const _) == (Add _ _)    = False
 (Add _ _) == (Const _)    = False
 (Const i) == (Const i')   = i == i'
 (Add x y) == (Add x' y') =
                        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 defintion.

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

```
> :t id
a -> a
```

No constraint =
parametric polymorphism

```
> :t (==)
(Eq a) => a -> a -> Bool
```

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

C#/Java concept	**Haskell concept**
Class	—
Interface	***Type class***
Interface member	Type-class member
Interface implementation	Type-class instance

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
```

instance

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

static

```
> :t (==)
(Eq a) => a -> a -> Bool
```

binary

Let's solve the expression problem with open datatypes and open functions.

Point of reference: the **closed datatype**

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

Note that there are two constructors;
one of them 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

```
data Const   = Const Int
data Add l r = Add l r
```

```
class Expr x
instance Expr Const
instance (Expr l, Expr r) =>
           Expr (Add l r)
```

Type-parameter constraints

A type class for the original datatype

The **open function** (type-class declaration)

A super-class constraint

```
class Expr x => Evaluate x

 where

  evaluate :: x -> Int
```

The **open function** (type-class instances)

```
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

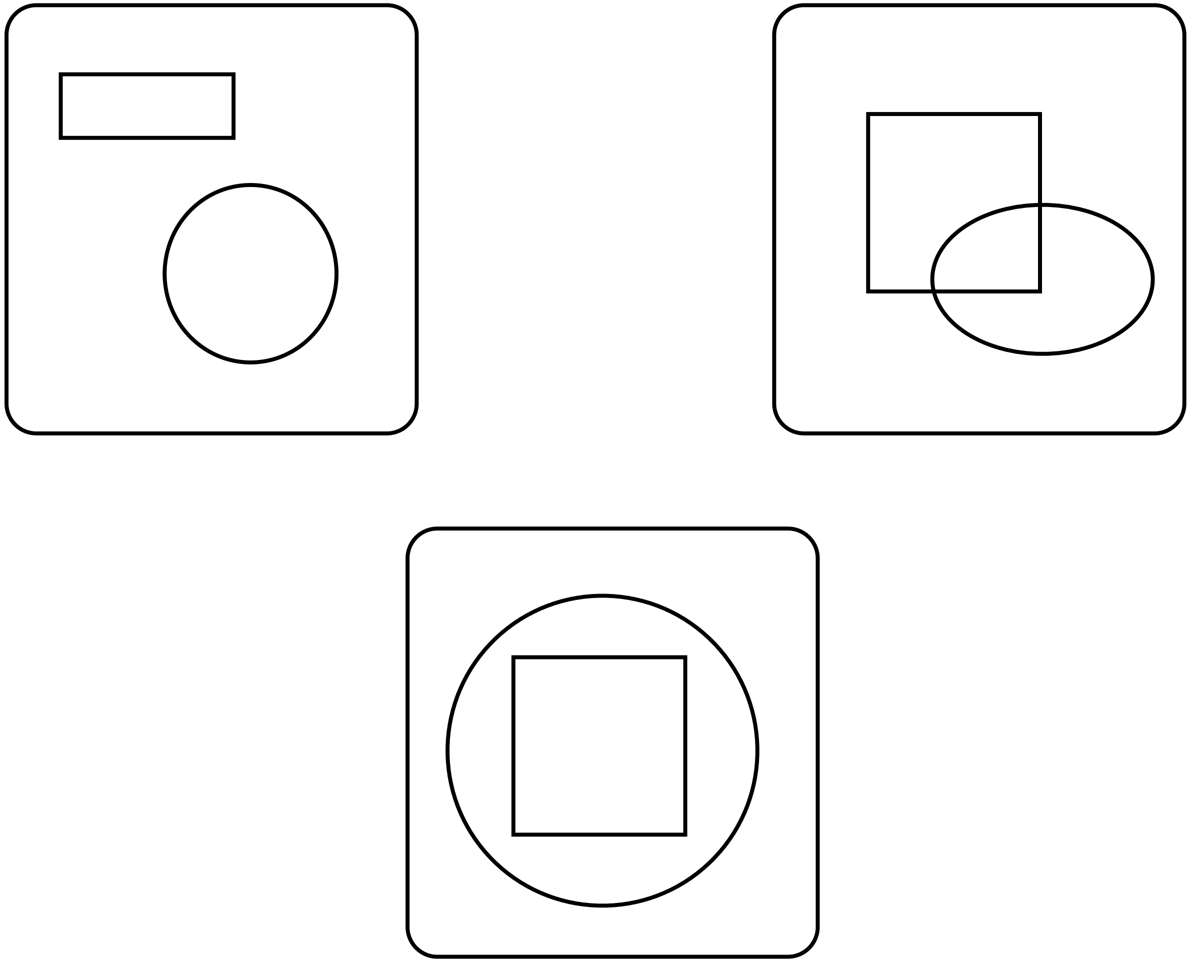
A data extension

```
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**

```
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**

```
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.

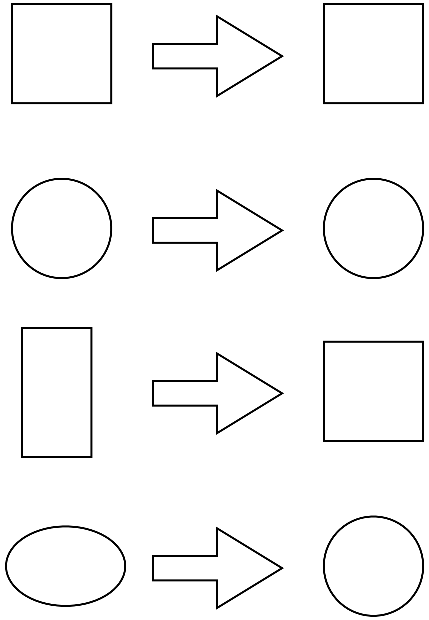
```
class (Shape x, Shape y) => Intersect x y
 where
  intersect :: x -> y -> Bool

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 *normalize*.



Preserve area and origin!

Point of reference: the **closed function**

```
normalize :: Shape -> Shape
normalize s@(Square _ _ _)    = s
normalize (Rectangle x y h w) = Square ...
normalize c@(Circle _ _ _)    = c
normalize (Ellipse x y a i)   = Circle ...
```

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

```
instance Normalize Square      Square where ...
instance Normalize Circle      Circle where ...
instance Normalize Rectangle   Square where ...
instance Normalize Ellipse     Circle where ...
```

An instance in a module that is compiled later

```
instance Normalize Square      Circle where ...
```

Type classes with functional dependencies

```
class (Shape s1, NormalShape s2)
   => Normalize s1 s2
   | s1 -> s2
 where
  normalize :: s1 -> s2
```

There cannot be two instances with the same s1 (but different 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.

```
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.