Type Classes

HASKELL

A pure functional language with Class!
Outline

• Introduction to type classes
• Associated laws
• Tradeoffs and extensibility
• Relationship to dictionary pattern
• Multi-parameter type classes
What is a type class?

An *interface* that is supported by many different types

A set of *types* that have a common behavior

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

class Show a where
    show :: a -> String

class Num a where
    (+) :: a -> a -> a
    (*) :: a -> a -> a
    negate :: a -> a
    ...
```

*types whose values can be compared for equality*

*types whose values can be shown as strings*

*types whose values can be manipulated like numbers*

... similar to a Java/C# interface
Constraining types

List elements can be of any type

```haskell
class Eq a where
  (==) :: a -> a -> Bool
length :: [a] -> Int
length []     = 0
length (_:xs) = 1 + length xs
```

List elements must be of a type that supports equality!

```haskell
elem :: Eq a => a -> [a] -> Bool
elem _     []       = False
elem y (x:xs) = x == y || elem y xs
```

use method ⇒ add constraint
Anatomy of a type class definition

class Eq a where

    (==) :: a -> a -> Bool
    (/=) :: a -> a -> Bool

    x == y = not (x /= y)
    x /= y = not (x == y)
Anatomy of a type class instance

instance Eq Bool where
  True  == True  = True
  False == False = True
  _     == _     = False
Constraints on instances

if we can check equality of \( a \) then we can check equality of \([a]\)

\[
\text{instance } \text{Eq } a \Rightarrow \text{Eq } [a] \text{ where }
\]
\[
[] \quad == \quad [] \quad = \quad \text{True}
\]
\[
(x:xs) \quad == \quad (y:ys) \quad = \quad x \quad == \quad y \quad && \quad xs \quad == \quad ys
\]

\( (==) \) for element type \( a \)

recursively apply \( (==) \) for type \([a]\)

\[
\text{instance } (\text{Eq } a, \text{Eq } b) \Rightarrow \text{Eq } (a,b) \text{ where }
\]
\[
(a1,b1) \quad == \quad (a2,b2) \quad = \quad a1 \quad == \quad a2 \quad && \quad b1 \quad == \quad b2
\]

\( (==) \) for type \( a \)

\( (==) \) for type \( b \)
Deriving type class instances

Generate a “standard” instance for your own data type

- derived from the structure of your type
- possible only for some built-in type classes
  
  \((\text{Eq, Ord, Enum, Show, ...})\)

```
data Set a = Empty
    | Elem a (Set a)
deriving (Eq, Show)
```

```
instance Eq a => Eq (Set a) where
    Empty      == Empty      = True
    Elem a1 s1 == Elem a2 s2 = a1 == a2 && s1 == s2
    _          == _          = False

instance Show a => Show (Set a) where
    show Empty     = "Empty"
    show (Elem a s) = "(Elem " ++ show a ++ " " ++ show s ++ ")"
```

if this isn't what you want, write a custom instance!
Class extension

any instance of Ord must also be an instance of Eq

```
class Eq a => Ord a where
  compare              :: a -> a -> Ordering
  (<), (<=), (>=), (>) :: a -> a -> Bool
  max, min             :: a -> a -> a

data Ordering = LT | EQ | GT

find :: Ord a => a -> Tree a -> Bool
find _ Leaf             = False
find x (Node y l r) | x == y     = True
                     | x < y     = find x l
                     | otherwise = find y r
```

```
“superclass”

| Any instance of Ord must also be an instance of Eq |

| “subclass” |

type class we’re defining a.k.a. “subclass” |

why don't we need a constraint for Eq?
```
Outline

• Introduction to type classes

• **Associated laws**

• Tradeoffs and extensibility

• Relationship to dictionary pattern

• Multi-parameter type classes
Associated laws

Most type classes come with **laws**
= equations or properties that every instance must satisfy

class Monoid a where
  mempty :: a
  mappend :: a -> a -> a

*left identity*  \[\text{mappend } x \text{ mempty } \leftrightarrow x\]
*right identity*  \[\text{mappend } \text{ mempty } x \leftrightarrow x\]
*associativity*  \[\text{mappend } x (\text{mappend } y z) \leftrightarrow \text{mappend } (\text{mappend } x y) z\]

library authors will assume your instances follow the laws!
Outline

• Introduction to type classes
• Associated laws

• Tradeoffs and extensibility
• Relationship to dictionary pattern
• Multi-parameter type classes
Type classes vs. explicit parameters

**Compare via type class**

```haskell
qsort :: Ord a => [a] -> [a]
qsort [] = []
qsort (x:xs) = qsort [y | y <- xs, y < x] ++ [x] ++ qsort [y | y <- xs, y >= x]
```

**Compare via higher-order comparison function**

```haskell
qsort :: (a -> a -> Bool) -> [a] -> [a]
qsort _lt [] = []
qsort _lt (x:xs) = qsort _lt [y | y <- xs, _lt y x] ++ [x] ++ qsort _lt [y | y <- xs, not (_lt y x)]
```

What are the tradeoffs of these approaches?
Type classes vs. explicit parameters

Rely on type class:
- do the same thing for each type
- don’t need to pass around function parameter

Pass explicit parameter:
- can do different things for the same type
- must thread parameters through functions

*In Data.List see *By functions for passing equivalence predicate rather than relying on Eq*
Type classes and extensibility

Consider a shape library:

- easy to add new operations
- hard to add new shapes

“hard” = not modular

```hs
type Radius = Float
type Length = Float
type Width = Float

data Shape = Circle Radius |
| Rectangle Length Width |
| Triangle Length

area :: Shape -> Float
area (Circle r) = pi * r * r
area (Rectangle l w) = l * w
area (Triangle l) = ...  

perim :: Shape -> Float
perim (Circle r) = 2 * pi * r
perim (Rectangle l w) = 2*l + 2*w
perim (Triangle l) = l + l + l
```

(ShapeData.hs, ShapeClass.hs)
# Type classes and extensibility

<table>
<thead>
<tr>
<th>Concept</th>
<th>Data-type encoding</th>
<th>Type-class encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cases</td>
<td>Data type</td>
<td>Type class</td>
</tr>
<tr>
<td>Operations</td>
<td>Functions</td>
<td>Data types</td>
</tr>
<tr>
<td></td>
<td>• easy to add ops</td>
<td>• methods</td>
</tr>
<tr>
<td></td>
<td>• hard to add cases</td>
<td>• hard to add ops</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• easy to add cases</td>
</tr>
</tbody>
</table>

**What are some other tradeoffs of these approaches?**

Later we’ll see encodings that support extension in both dimensions!
Outline

• Introduction to type classes
• Associated laws
• Tradeoffs and extensibility
• Relationship to dictionary pattern
• Multi-parameter type classes
Type classes vs. dictionary pattern

class Num a where
  (+) :: a -> a -> a

instance Num Int where
  (+) = primIntAdd

instance Num Float where
  (+) = primFloatAdd

double :: Num a => a -> a
  double x = x + x

data NumD a = ND (a -> a -> a)

add :: NumD a -> a -> a -> a
  add (ND f) = f

intD :: NumD Int
  intD = ND primIntAdd

floatD :: NumD Float
  floatD = ND primFloatAdd

double :: NumD a -> a -> a
  double d x = add d x x

explicitly pass dictionary

Phil Wadler, How to make ad-hoc polymorphism less ad hoc
POPL 1989
Multiple constraints and super classes

Multiple class constraints:

\[ \texttt{doubles :: (Num } a, \texttt{ Num } b) \Rightarrow a \to b \to (a, b) \]
\[ \texttt{doubles } x \; y = (x + x, y + y) \]

Lead to multiple dictionaries:

\[ \texttt{doubles :: (NumD } a, \texttt{ NumD } b) \Rightarrow a \to b \to (a, b) \]
\[ \texttt{doubles } (da, db) \; x \; y = (\text{add } da \; x \; x, \text{add } db \; y \; y) \]

Super classes:

\[ \texttt{class Eq } a \; \texttt{ where} \]
\[ \quad (==) :: a \to a \to \texttt{Bool} \]
\[ \texttt{class Eq } a \Rightarrow \texttt{Ord } a \; \texttt{ where} \]
\[ \quad (<) :: a \to a \to \texttt{Bool} \]
\[ \ldots \]

Lead to nested dictionaries:

\[ \texttt{data EqD } a = \]
\[ \quad \texttt{ED } (a \to a \to \texttt{Bool}) \]
\[ \texttt{data OrdD } a = \]
\[ \quad \texttt{OD } (\texttt{EqD } a) (a \to a \to \texttt{Bool}) \]
\[ \ldots \]
Translating to the dictionary pattern

Type classes are *implemented* in Haskell by dictionaries:

- translate type classes to dictionary data types
- translate instances to dictionary values
- translate constraints to function arguments
- *use type system to automatically insert dictionary values*

Phil Wadler, *How to make ad-hoc polymorphism less ad hoc*
POPL 1989
Outline

• Introduction to type classes
• Associated laws
• Tradeoffs and extensibility
• Relationship to dictionary pattern
• Multi-parameter type classes
Multi-parameter type classes

Defines a relation between types

Can convert from $a$ to $b$

```haskell
class Cast a b where
    cast :: a -> b
```

Defines an interface for intersection of types

Implement collection interface for pair of:
- $c$ – container type
- $a$ – element type

```haskell
class Collection c a where
    empty :: c a
    insert :: a -> c a -> c a
    member :: a -> c a -> Bool
```

(Cast.hs, Collection.hs)

Turn on your GHC extensions!