Type Classes

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

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

types whose values can be compared for equality

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

types whose values can be shown as strings

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

types whose values can be manipulated like numbers

. . . similar to a Java/C# interface

Constraining types

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

List elements can be of any type

```
length :: [a] -> Int
length []     = 0
length (_:xs) = 1 + length xs
```

List elements must be of a type that supports equality!

```
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 name

variable representing instance type

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

  x == y = not (x /= y)
  x /= y = not (x == y)
```

methods

default implementations

must define either `(==)` *or* `(/=)`

Anatomy of a type class instance

class name — *type we're implementing the interface for*

```
instance Eq Bool where
  True  == True  = True
  False == False = True
  _     == _     = False
```

regular function definition

don't need to define (/=)

Constraints on instances

if we can check equality of `a` *then we can check equality of* `[a]`

```
instance Eq a => Eq [a] where
  []     == []     = True
  (x:xs) == (y:ys) = x == y && xs == ys
```

`(==)` *for element type* `a`

recursively apply `(==)` *for type* `[a]`

```
instance (Eq a, Eq b) => Eq (a,b) where
  (a1,b1) == (a2,b2) = a1 == a2 && b1 == 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
  (`Eq`, `Ord`, `Enum`, `Show`, . . .)

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

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

```
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 ++ ")"
```

(Time.hs)

Class extension

any instance of **`Ord`** *must also be an instance of* **`Eq`**

"superclass"

type class we're defining a.k.a. "subclass"

```
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
```

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
```

library authors will assume your instances follow the laws!

left identity `mappend x mempty  <=>  x`

right identity `mappend mempty x  <=>  x`

associativity `mappend x (mappend y z)`
`<=>  mappend (mappend x y) z`

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

```
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

```
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?

(QSort.hs)

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

```
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
```

Using type classes, we can invert this extensibility problem!

(ShapeData.hs, ShapeClass.hs)

Type classes and extensibility

	data-type encoding	type-class encoding
concept	data type	type class
cases	data constructors	data types
operations	functions	methods
	• *easy to add ops* • *hard to add cases*	• *hard to add ops* • *easy to add cases*

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

```haskell
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
```

```haskell
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

(MonoidClass.hs)

Multiple constraints and super classes

Multiple class constraints:

```
doubles :: (Num a, Num b) => a -> b -> (a,b)
doubles x y = (x + x, y + y)
```

Lead to multiple dictionaries:

```
doubles :: (NumD a, NumD b) -> a -> b -> (a,b)
doubles (da,db) x y = (add da x x, add db y y)
```

Super classes:

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

class Eq a => Ord a where
  (<)  :: a -> a -> Bool
  ...
```

Lead to nested dictionaries:

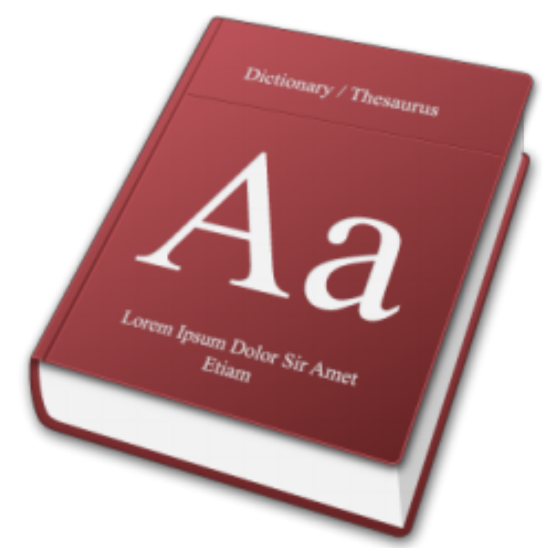
```
data EqD a =
  ED (a -> a -> Bool)

data OrdD a =
  OD (EqD a) (a -> a -> Bool)
...
```

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

Turn on your GHC extensions!

Defines a *relation* between types

Can convert from a to b

```
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*

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

(Cast.hs, Collection.hs)