Type Classes

Haskell
A pure functional language with Class!
Outline

• Introduction to type classes
• Tradeoffs and extensibility
• Relationship to dictionary pattern
• Kinds and type constructor classes
• Multi-parameter type classes
• Laws
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
```

```
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 interface
Constraining types

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

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

not like Java methods!

methods

default implementations
must define either
(==) or (/=)
Anatomy of a type class instance

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

class name

type we're implementing the interface for

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

(==) 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
  \((\text{Eq}, \text{Ord}, \text{Enum}, \text{Show}, \ldots)\)

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

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

```haskell
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 ++ ")"
```
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
```

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

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

Compare via type class

\[
\text{qsort} :: \text{Ord} \; a \Rightarrow [a] \rightarrow [a] \\
\text{qsort} \; [] = [] \\
\text{qsort} \; (x:xs) = \text{qsort} \; [y \mid y \leftarrow xs, y < x] \\
\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad ++ \quad [x] \\
\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad ++ \quad \text{qsort} \; [y \mid y \leftarrow xs, y \geq x]
\]

Compare via higher-order comparison function

\[
\text{qsort} :: (a \rightarrow a \rightarrow \text{Bool}) \rightarrow [a] \rightarrow [a] \\
\text{qsort} \; \text{lt} \; [] = [] \\
\text{qsort} \; \text{lt} \; (x:xs) = \text{qsort} \; \text{lt} \; [y \mid y \leftarrow xs, \text{lt} \; y \; x] \\
\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad ++ \quad [x] \\
\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad ++ \quad \text{qsort} \; \text{lt} \; [y \mid y \leftarrow xs, \text{not} \; (\text{lt} \; y \; x)]
\]

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

```haskell
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)    = ...  -- Using type classes, we can invert this extensibility problem!

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></th>
<th>data-type encoding</th>
<th>type-class encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>concept</td>
<td>data type</td>
<td>type class</td>
</tr>
<tr>
<td>cases</td>
<td>data constructors</td>
<td>data types</td>
</tr>
<tr>
<td>operations</td>
<td>functions</td>
<td>methods</td>
</tr>
<tr>
<td></td>
<td>• easy to add ops</td>
<td>• hard to add ops</td>
</tr>
<tr>
<td></td>
<td>• hard to add cases</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
• Tradeoffs and extensibility
• Relationship to dictionary pattern
• Kinds and type constructor classes
• Multi-parameter type classes
• Laws
Type classes and the 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

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:

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

Lead to multiple dictionaries:

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

Super classes:

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

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

Lead to nested dictionaries:

```haskell
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
• Tradeoffs and extensibility
• Relationship to dictionary pattern
• Kinds and type constructor classes
• Multi-parameter type classes
• Laws
Kinds

Like a super-simple type system for types (only keeps track of arity)

A regular type has kind *

String Int [Bool] Maybe Int
Tree a Map k v Int -> Int a -> a -> Bool

A type constructor takes one or more types as arguments and produces a type

Maybe :: * -> * Tree :: * -> * [] :: * -> *
Map :: * -> * -> * (->) :: * -> * -> *

Haskell functions and values always have regular types!
Functor  class of data structures that can be mapped over

class Functor t where
  fmap :: (a -> b) -> t a -> t b

map :: (a -> b) -> [a] -> [b]
map _ [] = []
map f (x:xs) = f x : map f xs

tmap :: (a -> b) -> Tree a -> Tree b
tmap _ Leaf = Leaf
tmap f (Node x l r) = Node (f x) (tmap f l) (tmap f r)

mmap :: (a -> b) -> Maybe a -> Maybe b
mmap _ Nothing = Nothing
mmap f (Just x) = Just (f x)
Functor  

*class of data structures that can be mapped over*

class Functor t where
  fmap :: (a -> b) -> t a -> t b

data structures that can be mapped over

instance Functor [] where
  fmap _ []     = []
  fmap f (x:xs) = f x : fmap f xs

instance Functor Tree where
  fmap _ Leaf = Leaf
  fmap f (Node x l r) = Node (f x) (fmap f l) (fmap f r)

instance Functor Maybe where
  fmap _ Nothing = Nothing
  fmap f (Just x) = Just (f x)
Foldable class of data structures that can be accumulated over

foldr :: (a -> b -> b) -> b -> [a] -> b
foldr _ b [] = b
foldr f b (x:xs) = f x (foldr f b xs)

tfold :: (a -> b -> b -> b) -> b -> Tree a -> b
tfold _ b Leaf = b
tfold f b (Node x l r) = f x (tfold f b l) (tfold f b r)

Accumulator functions have different types
... can we refactor tfold to make them the same?
Foldable class of data structures that can be accumulated over

class Foldable t where
  foldr :: (a -> b -> b) -> b -> [a] -> b
  foldr _ b [] = b
  foldr f b (x:xs) = f x (foldr f b xs)

tfold :: (a -> b -> b -> b) -> b -> Tree a -> b
  tfold _ b Leaf = b
  tfold f b (Node x l r) = f x (tfold f b l) (tfold f b r)

tfoldr :: (a -> b -> b) -> b -> Tree a -> b
  tfoldr _ b Leaf = b
  tfoldr f b (Node x l r) = tfoldr f (f x (tfoldr f b r)) l
Foldable

class of data structures that can be accumulated over

class Foldable t where
foldr :: (a -> b -> b) -> b -> t a -> b

instance Foldable [] where
foldr _ b [] = b
foldr f b (x:xs) = f x (foldr f b xs)

tfold :: (a -> b -> b -> b) -> b -> Tree a -> b
tfold _ b Leaf = b
tfold f b (Node x l r) = f x (tfold f b l) (tfold f b r)

instance Foldable Tree where
foldr _ b Leaf = b
foldr f b (Node x l r) = foldr f (f x (foldr f b r)) l
Type classes as an abstraction mechanism

**abstraction**: to separate a concept from its specific instances and make it reusable

- **higher-order functions**
  names and makes reusable the implementation of high-level programming patterns for working with a *single data structure*

- **type classes**
  names and makes reusable the interface of high-level programming patterns for working with a *variety of data structures*
Why abstract these shared interfaces?

• *reuse functions over classes of data types* by instantiating Foldable or (especially) Monad you have access to tons of *library functions*

• *write code that is extensible with new data types* describe the *interface* of a data type you expect then program against that interface

• *describe precisely the properties of your data type* type classes induce a *classification scheme* for data types
Outline

• Introduction to type classes
• Tradeoffs and extensibility
• Relationship to dictionary pattern
• Kinds and type constructor classes
• Multi-parameter type classes
• Laws
Multi-parameter type classes

Defines a relation between types

Can convert from \(a\) to \(b\)

\[
\text{class } \text{Cast } a \ b \ \text{where}
\text{cast} :: a \to b
\]

Defines an interface for intersection of types

Implement collection interface for pair of:
- \(c\) – container type
- \(a\) – element type

\[
\text{class } \text{Collection } c \ a \ \text{where}
\text{empty} :: c\ a
\text{insert} :: a \to c\ a \to c\ a
\text{member} :: a \to c\ a \to \text{Bool}
\]
Outline

• Introduction to type classes
• Tradeoffs and extensibility
• Relationship to dictionary pattern
• Kinds and type constructor classes
• Multi-parameter type classes
• Laws
**Functors (review)**

Types that can be mapped over

```haskell
class Functor t where
    fmap :: (a -> b) -> t a -> t b
```

instance Functor [] where
    fmap _ [] = []
    fmap f (x:xs) = f x : fmap f xs

instance Functor Maybe where
    fmap _ Nothing = Nothing
    fmap f (Just x) = Just (f x)
```
Functor laws

Equations that every Functor instance should satisfy:

\[
\begin{align*}
\text{fmap id} & \iff \text{id} \\
\text{fmap } (f \cdot g) & \iff \text{fmap } f \cdot \text{fmap } g
\end{align*}
\]

Means that \text{fmap} preserves the \textit{structure} of values

\ldots code written against the Functor interface can assume this
Applicative functors

functors that support application

class Functor f => Applicative f where
  pure :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b

instance Applicative [] where
  pure x = [x]
  (f:fs) <*> xs = fmap f xs ++ (fs <*> xs)
  [] <*> _ = []

instance Applicative Maybe where
  pure x = Just x
  Just f <*> mx = fmap f mx
  Nothing <*> _ = Nothing
Applicative functor laws

Equations that every Applicative instance should satisfy:

- **identity**
  \[
  \text{pure id <*> v} \iff v
  \]

- **homomorphism**
  \[
  \text{pure f <*> pure x} \iff \text{pure (f x)}
  \]
class Functor f => Applicative f where
pure :: a -> f a
(<*>): f (a -> b) -> f a -> f b

Equations that every Applicative instance should satisfy:

**composition**

pure (.) <*> u <*> v <*> w

===>

u <*> (v <*> w)

**interchange**

u <*> pure y

===>

pure ($ y) <*> u
Relationship to Functor

class Functor t where
  fmap :: (a -> b) -> t a -> t b

class Functor f => Applicative f where
  pure :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b

fmap f x  <=>  pure f <*> x