Motivation

- What should the principal type of (+) be?
  - Int -> Int -> Int -- too specific
  - a -> a -> a -- too general
- It seems like we need something "in between" that restricts "a" to be from the set of all number types, say Num = (Int, Integer, Float, Double, etc.).
- The type a -> a -> a is really shorthand for (\forall a) a -> a -> a.
- Qualified types generalize this by qualifying the type variable, as in (\forall a \in \text{Num}) a -> a -> a, which in Haskell we write as Num a => a -> a -> a.

Type Classes

- "Num" in the previous example is called a type class, and should not be confused with a type constructor or a value constructor.
- "Num T" should be read "T is a member of (or an instance of) the type class Num".
- Haskell's type classes are one of its most innovative features.
- This capability is also called "overloading", because one function name is used for potentially very different purposes.
- There are many pre-defined type classes, but you can also define your own.

Example: Equality

- Like addition, equality is not defined on all types (how would we test the equality of two functions, for example?).
- So the equality operator (==) in Haskell has type Eq a => a -> a -> Bool. For example:
  ```haskell
  42 == 42  \rightarrow  True
  'a' == 'a'  \rightarrow  True
  2 == 42  \rightarrow  Error (types don't match)
  (+1) == (+1)  \rightarrow  Error (types don't match)
  ```
  - Note: the type errors occur at compile time!

Equality, cont'd

- Eq is defined by this type class declaration:
  ```haskell
  class Eq a where
  (==) :: a -> a -> Bool
  x == y = not (x /= y)
  ```
- The last two lines are default methods for the operators defined to be in this class.
- A type is made an instance of a class by an instance declaration. For example:
  ```haskell
  instance Eq Int where
  a == y = intEq x y -- primitive equality for Ints
  ```
  ```haskell
  instance Eq Float where
  a == y = floatEq x y -- primitive equality for Floats
  ```

Equality, cont'd

- User-defined data types can also be made instances of Eq. For example:
  ```haskell
  data Tree a = Leaf a | Branch (Tree a) (Tree a)
  ```
  ```haskell
  instance Eq (Tree a) where
  Leaf a1 == Leaf a2 = a1 == a2
  Branch l1 r1 == Branch l2 r2 = l1 == l2 && r1 == r2
  ```
- But something is strange here: is "a1 == a2" on the right-hand side correct? How do we know that equality is defined on the type "a"???
Equality, cont’d

- User-defined data types can also be made instances of `Eq`. For example:

  ```haskell
  data Tree a = Leaf a | Branch (Tree a) (Tree a)
  instance Eq a => Eq (Tree a) where
    Leaf a1      == Leaf a2      = a1 == a2
    Branch l1 r1 == Branch l2 r2 = l1==l2 && r1==r2
    _            == _            = False
  ```

- But something is strange here: Is “a1 == a2” on the right-hand side correct? How do we know that equality is defined on the type “a”??

  - Answer: Add a constraint that requires a to be an equality type.

Constraints / Contexts are Propagated

- Consider this function:

  ```haskell
  x `elem` [] =  False
  x `elem` (y:ys) =  x==y || x `elem` ys
  ```

- Note the use of (==) on the right-hand side of the second equation. So the principal type for `elem` is:

  ```haskell
  elem :: Eq a => a -> [a] -> Bool
  ```

  This is inferred automatically by Haskell, but, as always, it is recommended that you provide your own type signature for all functions.

Classes for Regions

- Useful slogan: “Polymorphism captures similar structure over different values, while type classes capture similar operations over different structures.”
- For a simple example, recall from Chapter 8:

  ```haskell
  class PC t where
    contains :: t -> Point -> Bool
  ```

  These are similar ops over different structures. So:

  ```haskell
  instance PC Shape where
    contains = containsS
  instance PC Region where
    contains = containsR
  ```

- “Polymorphism captures similar structure over different values, while type classes capture similar operations over different structures.”

Numeric Classes

- Haskell’s numeric types are embedded in a very rich, hierarchical set of type classes.
- For example, the `Num` class is defined by:

  ```haskell
  class  (Eq a, Show a) => Num a  where
    (+) , (-) , (*) :: a -> a -> a
    negate :: a -> a
    abs    , signum    :: a -> a
    fromInteger :: Integer -> a
  ```

  ...where `Show` is a type class whose main operator is `show :: Show a => a -> String`

  See the Numeric Class Hierarchy in the Haskell Report on the next slide.

Coercions

- Note this method in the class `Num`:

  ```haskell
  fromIntegral :: Num a => Integer -> a
  ```

- Also, in the class `Integral`:

  ```haskell
  toIntegral :: Integral a => a -> Integer
  ```

  This explains the definition of `intToFloat`:

  ```haskell
  intToFloat :: Int -> Float
  intToFloat n = fromIntegral (toIntegral n)
  ```

  These generic coercion functions avoid a quadratic blowup in the number of coercion functions.

  - Also, every integer literal, say “42”, is really shorthand for “fromIntegral 42”, thus allowing that number to be typed as any member of `Num`.
Derived Instances

- Instances of the following type classes may be automatically derived:
  - `Eq`, `Ord`, `Enum`, `Bounded`, `Ix`, `Read`, and `Show`
- This is done by adding a `deriving` clause to the data declaration. For example:
  
  ```haskell
data Tree a = Leaf a | Branch (Tree a) (Tree a)
deriving (Show, Eq)
```
- This will automatically create an instance for `Show (Tree a)` as well as one for `Eq (Tree a)` that is precisely equivalent to the one we defined earlier.

Derived vs. User-Defined

- Suppose we define an implementation of finite sets in terms of lists, like this:
  ```haskell
data Set a = Set [a]
insert (Set s) x = Set (x:s)
member (Set s) x = elem x s
union (Set s) (Set t) = Set (s++t)
```
- We can automatically derive an equality function just by adding “deriving `Eq`” to the declaration.
  ```haskell
data Set a = Set [a]
deriving Eq
insert (Set s) x = Set (x:s)
member (Set s) x = elem x s
union (Set s) (Set t) = Set (s++t)
```

But is this really what we want?!!

A Better Way

- Data Sets
  ```haskell
data Set a = Set [a]
instance Eq a => Eq (Set a) where
  s == t = subset s t && subset t s
subset (Set ss) t = all (member t) ss
```

Reasoning About Type Classes

- Most type classes implicitly carry a set of laws.
- For example, the `Eq` class is expected to obey:
  ```haskell
  (a == b) == (b == a) 
  (a == b) && (b == c) => (a == c)
  (a == b) && (b == a) => (a == b)
  (a == b) || (b == a) => (a == c)
  ```
- Similarly, for the `Ord` class:
  ```haskell
  (a <= b) || (b <= a) => (a == b)
  (a <= b) && (b <= c) => (a <= c)
  (a <= b) && (b <= a) => (a == b)
  (a <= b) || (b <= a) => (a <= c)
  ```
- These laws capture the properties of an equivalence class and a total order, respectively.
- Unfortunately, there is nothing in Haskell that enforces the laws – it's up to the programmer!