CS 115
Functional Programming

Lecture 7:

Type Classes, part 1

Today

- Type classes
- Motivation
- Examples: `Eq`, `Ord`, `Num`, `Show`
- How type classes are implemented
- Type classes and algebraic datatypes

Motivation: operators

- Many operations referred to by a single name actually behave differently when used on different types
- Example: + (addition)
- The operation of adding two integers is completely different from the operation of adding two floating-point numbers
  - Other kinds of numbers have still other definitions
  - Yet we use the same symbol (+) for all of these!

Motivation: operators

- Use of common symbols for different operations comes from mathematics
  - simplifies notation
  - can use context (type information) to disambiguate actual intended operations
- Most computer languages "overload" such operators based on the types of the operands
- However, such overloading is usually hard-wired (non-extensible to new types)

Motivation: operators

- Some languages (*e.g.* C++) allow user-defined operator overloading
- Still major limitations:
  - *e.g.* cannot define new operators (fixed set)
  - operators have no semantic content, so can lead to hard-to-understand code
- Other languages (*e.g.* Java, OCaml) forbid operator overloading altogether
  - weakens expressive power of language

Motivation: functions

- Operators are not the only language entities that can conceptually be defined for multiple types
- Often have a notion of a function which should be specialized based on a particular type
  - *e.g.* "convert a value of this type to a string"
  - this is a *generic* function for this functionality
- Some languages deal with this through object-oriented features
  - classes, instances, interfaces

Type classes

- Haskell uses type classes to represent generic operations both at the operator and function level
- Type classes provide a very clean solution to the problem of operator overloading
- Also provide a very convenient way to define generic functions
- IMO: One of the uniquely wonderful features of Haskell, responsible for much of its power
  - also: many highly useful extensions!

Type classes

- Type classes referred to sometimes as "ad-hoc polymorphism"
- In contrast to previous kind of polymorphism, which is called "parametric polymorphism" (due to generalizing on type parameters)
- "Ad-hoc" means that it is essentially arbitrary which types instantiate which type classes
- Also *open*: can add new type class instances at any time after definition

Equality

- First example: equality
- Many data values have some well-defined notion of how to compare two such values to see if they are "equal"
- Some data values do not (notably functions)
- We use
  - the `==` operator to test two values for equality
  - the `/=` operator to test two values for inequality

Equality

- Consider two types: `Int` and `Float`
- Both have well-defined notions of equality comparison
- Comparing two `Int`s for equality a completely different operation than comparing two `Float`s
- Worst case: could define `intEq` and `floatEq` functions with these type signatures:
  - `intEq :: Int -> Int -> Bool`
  - `floatEq :: Float -> Float -> Bool`

Equality

- We can extend this to new types:
  - `charEq :: Char -> Char -> Bool`
  - `stringEq :: String -> String -> Bool`
- Also, would want to leave open the possibility of defining new equality operations later for user-defined types
  - *e.g.* `treeEq :: Tree -> Tree -> Bool` for some `Tree` data type

Equality

- Shape of type signature of all these functions
  - `xEq :: x -> x -> Bool`
- It would be nice if there was a way to make the `==` operator work on *all* equality functions of this kind
  - including user-defined ones like `treeEq`

Eq

- The Haskell Prelude defines the **Eq** *type class* for this very purpose
- Definition:

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

Eq

- Interpretation:

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

- `Eq` is a *type class* with one *type parameter* `a`
- It gives the type signatures of two operators: `==` and `/=`
- Each of them takes two arguments of the same type `a` and returns a `Bool`

`class`

- The `class` definition gives the names of the functions (AKA *methods*) and operators of the type class along with their type signatures
- Type signatures in type class definitions always depend on type parameter `a` (or it would be useless)
- No other semantic information included in type class
  - *e.g.* that two values can either be `==` or `/=`, but not both and not neither is not part of the definition
  - (Haskell isn't that powerful!)

`class`

- The `class` terminology is *utterly unrelated* to object-oriented programming (OOP) terminology
- Terms like `class`, `instance`, method are used but mean *completely different things* than in OOP languages!
- Closest match to OOP languages like Java: type classes are like "compile-time interfaces"

`instance`

- Given a `class`, we must be able to create instances of the class
- Assume we have functions `intEq`, `floatEq` for `Int`, `Float` equality comparisons
- We can then define instances of `Eq` for `Int` and `Float`

instance

- Instances are defined as follows:

```
instance Eq Int where
  (==) = intEq
  x /= y = not (x == y)
  -- or: (/=) = (not .) . (==)

instance Eq Float where
  (==) = floatEq
  x /= y = not (x == y)
```

instance

- If you defined a `Tree` data type and `treeEq`:

```
instance Eq Tree where
  (==) = treeEq
  x /= y = not (x == y)
```

Default definitions

- Note redundancy in definition of `/=` operator:

```
instance Eq XXX where
  (==) = xxxEq
  x /= y = not (x == y)
```

- Nearly all types will define `/=` this way
- "Boilerplate" code (code with standard structure, repeated frequently) is anathema to the Haskell programmer
- Therefore, Haskell provides a shortcut

Default definitions

- Can define either `==` or `/=` in terms of the other
- Class definition becomes

```
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
  x /= y = not (x == y)
  x == y = not (x /= y)
```

Default definitions

```
class Eq a where
   (==) :: a -> a -> Bool
   (/=) :: a -> a -> Bool
  x /= y = not (x == y)
  x == y = not (x /= y)
```

- Now only need to define *either* `==` or `/=` for any `Eq` instance
  - The other is supplied automatically from default definitions
  - Can supply both *e.g.* for efficiency reasons

Type classes and functions

- Note the type of `(==)` operator in `ghci`:

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

- This type signature says "for any type `a` *such that* `a` is an instance of `Eq`, the type of `==` is `a -> a -> Bool`"
- The `=>` is a *context arrow*
- LHS of `=>` is the (type) context that the RHS must have
- Can write our own functions with type signatures like this

Type classes and functions

- Example function

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

- Now `allEqual` can be applied to a list of any type `a`, as long as that type is an instance of `Eq`
- `(Eq a) =>` specifies the *context* for the types in the type signature

Ord

- Another very useful type class is **Ord**
- Represents types whose values can be compared with each other
  - (*e.g.* one is "greater than" the other)
- Definition:

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

- … plus various default definitions
- Minimal instance definition: **compare** or **(<=)**

Ord

- `Ordering` is the following data type:

```
data Ordering = LT | EQ | GT
```

- Note context in `class` definition:

```
class (Eq a) => Ord a where ...
```

- This states that for a type to be an instance of `Ord`, it must first be an instance of `Eq` (makes sense)
- Note that we can write multiple method signatures on one line if the type signature is the same

```
(<), (<=), (>), (>=) :: a -> a -> Bool
```

Ord

- Recall `quicksort` definition:

```
quicksort :: [Integer] -> [Integer]
quicksort [] = []
quicksort (x:xs) =
  quicksort lt ++ [x] ++ quicksort ge
  where
    lt = [y | y <- xs, y < x]
    ge = [y | y <- xs, y >= x]
```

- Nothing here is particularly specific to `Integer`s
- How do we generalize this?

Ord

- Use `Ord` constraint:

```
quicksort :: Ord a => [a] -> [a]
quicksort [] = []
quicksort (x:xs) =
  quicksort lt ++ [x] ++ quicksort ge
  where
    lt = [y | y <- xs, y < x]
    ge = [y | y <- xs, y >= x]
```

- Now it will work on *any* orderable type!
  - Because those define the `<` and `>=` operators

Num

- Haskell has a hierarchy of numeric type classes
- Most basic one is called **Num** (for "numeric type")
- Definition:

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

Num

- `Num` instances represent what we expect all numbers to be able to do
- In older versions of GHC, `Num` had these class constraints:

```
class (Eq a, Show a) => Num a where ...
```

- These constraints have been removed!
  - `Show` was always a bogus constraint anyway, `Eq` less so
- `Num` instances also do not need to be instances of `Ord`
  - What would be an example of a `Num` type that isn't orderable?

Num

- Methods:
  - `+ - * negate abs` have the usual meanings
  - `signum` represents "sign" so that
    `abs x * signum x = x`
  - `fromInteger` converts an `Integer` into a value of the numeric type `a`

Numeric literals are overloaded!

- Type classes even evident in the types of numeric literals:

```
ghci> :type 42
42 :: Num a => a
```

- The number `42` has no specific type!
- It is of type `a`, where `a` is any `Num` instance
- `Num` instances include `Int`, `Integer`, `Float`, `Double`
- Therefore `42` is a valid literal for *any* of those types

```
ghci> :t (42 :: Float)
(42 :: Float) :: Float
```

Num example

- Simple function using **Num**:

```
sumOfSquares :: Num a => a -> a -> a
sumOfSquares x y = x * x + y * y

ghci> sumOfSquares 3 4
25
ghci> sumOfSquares 1.2 3.4
12.999999999999998
```

- **sumOfSquares** works generically for any **Num** instance

Implementation of type classes

- Type classes are implemented as a record of methods that is passed as an extra argument to functions using type classes
- The compiler supplies the extra arguments
- Example: **`Num`** instances represented as a record something like this:

```
data NumRecord a =
  NR { addOp :: a -> a -> a, subOp :: a -> a -> a,
       mulOp :: a -> a -> a,
       negateFn :: a -> a, absFn :: a -> a,
       signumFn :: a -> a,
       fromIntegerFn :: Integer -> a }
```

Implementation of type classes

- The **NumRecord** data declaration automatically defines accessors with these types:

```
addOp         :: NumRecord a -> a -> a -> a
subOp         :: NumRecord a -> a -> a -> a
mulOp         :: NumRecord a -> a -> a -> a
negateFn      :: NumRecord a -> a -> a
absFn         :: NumRecord a -> a -> a
signumFn      :: NumRecord a -> a -> a
fromIntegerFn :: NumRecord a -> Integer -> a
```

Implementation of type classes

- For a particular `Num` instance (*e.g.* `Int`), populate record with methods:

```
intNumRecord :: NumRecord Int
intNumRecord = NR intAddOp intSubOp intMulOp intNegateFn
                  intAbsFn intSignumFn intFromIntegerFn
```

- Note that *e.g.* `intAddOp` has the type `Int -> Int -> Int` while generic `addOp` accessor has the type `NumRecord a -> a -> a -> a`

Implementation of type classes

- Given a `NumRecord` value, `addOp` simply picks out the first component of the record, which is the addition operator for that type
- So `addOp intNumRecord` is just `intAddOp`
- Equivalently, `addOp` could be defined explicitly like this:

```
addOp :: NumRecord a -> a -> a -> a
addOp (NR add _ _ _ _ _ _) x y = add x y
```

Implementation of type classes

- Change definitions and function calls using **`Num`** to have extra arguments:

```
sumOfSquares :: NumRecord a -> a -> a -> a
sumOfSquares nr x y =
  addOp nr (mulOp nr x x) (mulOp nr y y)
```

- Note: We use **`addOp`** instead of **`(+)`** *etc.* because operators can only have two arguments

Implementation of type classes

- Compiler does all of these transformations for you
- Type classes are thus nothing more than normal functional programming with some fairly heavy syntactic sugar

Constrained datatypes

- In older versions of GHC, type class constraints could occur in datatype definitions as well
- Consider an ordered binary tree with data in branches
- Left subbranch contains only data "less than" data stored in a node
- Right subbranch contains only data "greater than" data stored in a node
- Let's write the datatype

Constrained datatypes

```
data Ord a => Tree a =    -- not legal anymore!
    Leaf
  | Node a (Tree a) (Tree a)
```

- Let's write a function on this datatype:

```
inTree :: Ord a => a -> Tree a -> Bool
inTree _ Leaf = False
inTree x (Node y left right) =
  case compare x y of
    LT -> inTree x left
    GT -> inTree x right
    EQ -> True
```

Constrained datatypes

- *Problem:* Having a constraint on a datatype doesn't remove the requirement for adding it to functions on that datatype:
- Our previous definition:

```
data Ord a => Tree a = ...
```

- Note the function:

```
inTree :: Ord a => a -> Tree a -> Bool
```

- still needs to have the `Ord` constraint!
- Therefore, it's generally considered a bad idea to add constraints directly to datatypes (useless)
  - Now requires the `DatatypeContexts` compiler option

Constrained datatypes

- Now we just remove the constraint and write:

```
data Tree a =
    Leaf
  | Node a (Tree a) (Tree a)
```

- and put the `Ord a =>` constraints on the functions that manipulate `Tree` values

Show

- Another very useful type class is **`Show`**
- Represents notion of "something that can be converted to a **`String`**"
- Definition:

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

- [A couple of other methods as well, not relevant for now]

Show

- To view a datatype in `ghci`, need to define a `Show` instance
- Example: `Test.hs`

```
data Color = Red | Green | Blue | Yellow
```

- In `ghci`:

```
ghci> :l ./Test.hs
ghci> :t Red
Red :: Color
```

- So far, so good…

Show

```
ghci> Red
<interactive>:1:1:
    No instance for (Show Color)
      arising from a use of `print'
    Possible fix: add an instance declaration for (Show Color)
    In a stmt of an interactive GHCi command: print it
```

- What happened?
- `ghci` is a "read-eval-print" loop (*REPL*)
- It reads an expression, evaluates it, and prints the result
- It can only print the result if the result can be printed!

Show

```
ghci> Red
<interactive>:1:1:
    No instance for (Show Color)
      arising from a use of `print'
    Possible fix: add an instance declaration for (Show Color)
    In a stmt of an interactive GHCi command: print it
```

- If no `Show` instance has been defined for the `Color` datatype, `ghci` can't do the printing → error message
- Error message even suggests what you need to do!
- So let's do it

Show

- In **Test.hs**:

```
data Color = Red | Green | Blue | Yellow
instance Show Color where
   show Red    = "Red"
   show Green  = "Green"
   show Blue   = "Blue"
   show Yellow = "Yellow"
```

Show

- In **ghci**:

```
ghci> :l ./Test.hs
ghci> Red
Red
```

- Woo hoo!
- Problem: This is boring "boilerplate" code
- Haskell programmers hate boilerplate code!
- We'll see a way to get around this next lecture

Next time

- More type classes
- Deriving type classes automatically
- Constructor classes and `Functor`
- Multi-parameter type classes
- A tour of Haskell type classes