CIS 1904: Haskell

Typeclasses

Logistics

- HW 5 due yesterday
  - Autograder and tests are working
- HW 6 will be released tomorrow
- If you haven't yet, please fill out the Canvas quiz by **March 1** at **11:59pm**

Polymorphism

A symbol is *polymorphic* if it can have more than one type

- Parametric polymorphism
  - e.g. `map :: (a -> b) -> [a] -> [b]` is parametric over `a` and `b`
    - Implicitly, `forall a, forall b, (a -> b) -> [a] -> [b]`
  - Must have same definition for every `a` (`b`, etc.)
- Ad-hoc polymorphism
  - Overloading, like `+`
  - Can behave differently for different types
  - Need not be defined for every type

Parametric Polymorphism

A symbol is *polymorphic* if it can have more than one type

- Parametric polymorphism
  - e.g. `map :: (a -> b) -> [a] -> [b]` is parametric over `a` and `b`
    - Implicitly, `forall a, forall b, (a -> b) -> [a] -> [b]`
  - Must have same definition for every `a` (`b`, etc.)

How many functions have type `a -> a`?

Parametric Polymorphism

A symbol is *polymorphic* if it can have more than one type

- Parametric polymorphism
  - e.g. `map :: (a -> b) -> [a] -> [b]` is parametric over `a` and `b`
    - Implicitly, `forall a, forall b, (a -> b) -> [a] -> [b]`
  - Must have same definition for every `a` (`b`, etc.)

How many functions have type `a -> a`?

One! `id :: a -> a`

Parametric Polymorphism

A symbol is *polymorphic* if it can have more than one type

- Parametric polymorphism
  - e.g. `map :: (a -> b) -> [a] -> [b]` is parametric over `a` and `b`
    - Implicitly, `forall a, forall b, (a -> b) -> [a] -> [b]`
  - Must have same definition for every `a` (`b`, etc.)

How many functions have type `a`?

No “real” functions – only used for error behavior in Haskell.

- E.g., `undefined` has this type

Ad-hoc Polymorphism

- Overloading, like `+`
- Can behave differently for different types
- Need not be defined for every type
- Implemented via typeclasses!

Ad-hoc Polymorphism

- Overloading, like `+`
- Can behave differently for different types
- Need not be defined for every type
- Implemented via typeclasses!

```
x = 1 + 0
```

What is the type of `+`? (Trick question)

Ad-hoc Polymorphism

- Overloading, like +
- Can behave differently for different types
- Need not be defined for every type
- Implemented via typeclasses!

```
x = 1 + 0
```

`Int -> Int -> Int`, `Double -> Double -> Double`, `Float -> Float -> Float`, etc.

Ad-hoc Polymorphism

- Overloading, like `+`
- Can behave differently for different types
- Need not be defined for every type
- Implemented via typeclasses!

```
(+) :: Num a => a -> a -> a
```

Ad-hoc Polymorphism

- Overloading, like `+`
- Can behave differently for different types
- Need not be defined for every type
- Implemented via typeclasses!

```
(+) :: Num a => a -> a -> a
```

`Num` is a typeclass!

Type Classes

- Typeclass: a collection of types that all have a certain set of functions defined for them, though those functions may be implemented differently

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

Type Classes

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

-- 'abs' and 'signum' should satisfy: abs x * signum x == x
```

Type Classes

```
class  Num a  where
    {-# MINIMAL (+), (*), abs, signum, fromInteger, (negate | (-)) #-}
     (+), (-), (*)        :: a -> a -> a
    negate               :: a -> a
    abs                  :: a -> a
    signum               :: a -> a
    fromInteger          :: Integer -> a

    x - y               = x + negate y
    negate x            = 0 - x
```

Type Classes

```
instance Num Integer where
    (+) = integerAdd
    (*) = integerMul
    negate      = integerNegate
    fromInteger i  = i
    abs = integerAbs
    signum = integerSignum

interegerAdd :: Integer -> Integer -> Integer
integerAdd = …
```

Type Classes

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

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

```
[1,2,3] == [4, 8, 1]
“Abc” == “Abc”
```

The compiler figures out which implementation we are calling!

Type Classes

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

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

Pro of default implementations: preserves invariants, e.g. `==` and `/=` are opposites

Con of default implementations: if we don't implement either it recurses infinitely

Type Classes: Show

- Haskell's version of toString
- Already implemented for most primitive types (`Int`, `Bool`, etc.)

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

```
instance Show Bool where
    show True = "True"
    show False = "False"
```

Type Classes: Show

- Haskell’s version of toString
- Already implemented for most primitive types (`Int`, `Bool`, etc.)

```
data Pair a b = Pair a b

instance Show (Pair a b) where
    show (Pair x y) = “(“ ++ show x ++ “, ” ++ show y ++ “)”
```

Type Classes: Show

- Haskell's version of toString
- Already implemented for most primitive types (`Int`, `Bool`, etc.)
-

```
data Pair a b = Pair a b

instance Show (Pair a b) where
    show (Pair x y) = "(" ++ show x ++ ", " ++ show y ++ ")"
```

Type Classes: Show

- Haskell's version of toString
- Already implemented for most primitive types (`Int`, `Bool`, etc.)

```
data Pair a b = Pair a b

instance (Show a, Show b) => Show (Pair a b) where
    show (Pair x y) = "(" ++ show x ++ ", " ++ show y ++ ")"
```

Type Classes: Show

- Haskell's version of toString
- Already implemented for most primitive types (`Int`, `Bool`, etc.)

```
data Pair a b = Pair a b
    deriving (Show)
```

Type Classes: Show

- Haskell's version of toString
- Already implemented for most primitive types (`Int`, `Bool`, etc.)

```
data Pair a b = Pair a b
    deriving (Show, Eq, Ord)
```

Type Classes: Ord

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

    compare x y = if x == y then EQ
                  else if x <= y then LT
                  else GT
    x <= y = case compare x y of { GT -> False; _ -> True }
    x >= y = y <= x
    x > y = not (x <= y)
    x < y = not (y <= x)
    max x y = if x <= y then y else x
    min x y = if x <= y then x else y
```

Type Classes: Ord

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

What is the right definition of `Ord` for a binary tree?

- Prefix-order comparison of elements?
- Infix-order comparison of elements?
- Number of nodes in the tree?
- Depth of the tree?

Type Classes vs. Java-style Interfaces

- Conceptually, typeclasses are a kind of interface
- Typeclasses are *not* Java-style interfaces
    - Java: a class must declare any interfaces it implements when created
    - Haskell: we can make a class an instance of a typeclass at any time

Type Classes vs. Java-style Interfaces

- Java: a class must declare any interfaces it implements when created
- Haskell: we can make a class an instance of a typeclass at any time

Expression Problem

```
data Shape
= Rectangle Int Int
| Circle Int


area :: Shape -> Double
area (Rectangle x y) = …
area (Circle x y) = …
```

```
interface Shape
    double area();
    int perimeter();

class Rectangle implements Shape
    int x, y;
    double area() { … }

class Circle implements Shape
    int r;
    double area() { … }
```

Which is easier to add a new function for?
Which is easier to add a new shape for?

Type Classes vs. Java-style Interfaces

- Typeclasses are more flexible in their method types
  - Implementation can depend on multiple parameter types
  - For multi-argument methods, there is no ambiguity about which argument we are calling the method on

```
class (Foo a, Foo b) => Foo a b where
    foo :: a -> b -> Bool
    foo = foo a || foo b
```