

How was the midterm?

- A: Easy
- B: Okay
- C: Hard

Was the midterm fair?

- A: Yes
- B: No

Logistics

- News: HW and PA due Sunday 23:59
- Type Tetris is now extra credit
 - See week4 notes for how to approach problem

Type classes



Problem w/ parametric polymorphism

- Consider the list member function:

```
member x []      = False
member x (y:ys) = if x == y
                  then True
                  else member x ys
```

- Is the type `member :: a -> [a] -> Bool` correct?
 - A: yes, B: no

Problem w/ parametric polymorphism

- Consider the list member function:

```
member x []      = False
member x (y:ys) = if x == y
                  then True
                  else member x ys
```

- Is the type `member :: a -> [a] -> Bool` correct?

➤ A: yes, B: no

Can these work on any type?

- `sort :: [a] -> [a]`
- `(+) :: a -> a -> a`
- `show :: a -> String`
- `serialize :: a -> ByteString`
- `hash :: a -> Int`

No! But we really want to use those same symbols to work on different types

- E.g., `3.4 + 5.5` and `3+5`
- E.g., `show 4` and `show [1,2,3]`
- E.g., `4 == 5` and `Left "w00t" == Right 44`

Motivation for overloading

- Parametric polymorphism doesn't work...
 - Single algorithm, works on values of any type
 - Type variable may be replaced by any type
- What we want: a form of overloading
 -
 -

Motivation for overloading

- Parametric polymorphism doesn't work...
 - Single algorithm, works on values of any type
 - Type variable may be replaced by any type
- What we want: a form of overloading
 - Single symbol to refer to more than one algorithm
 - Each algorithm may have different type

How should we do overloading?

Non-solution: local choice

- Overload basic operators such as + and *


a * b

- Don't allow for overloading functions defined from them

➤ **Problem?**

Non-solution: local choice

- Overload basic operators such as + and *

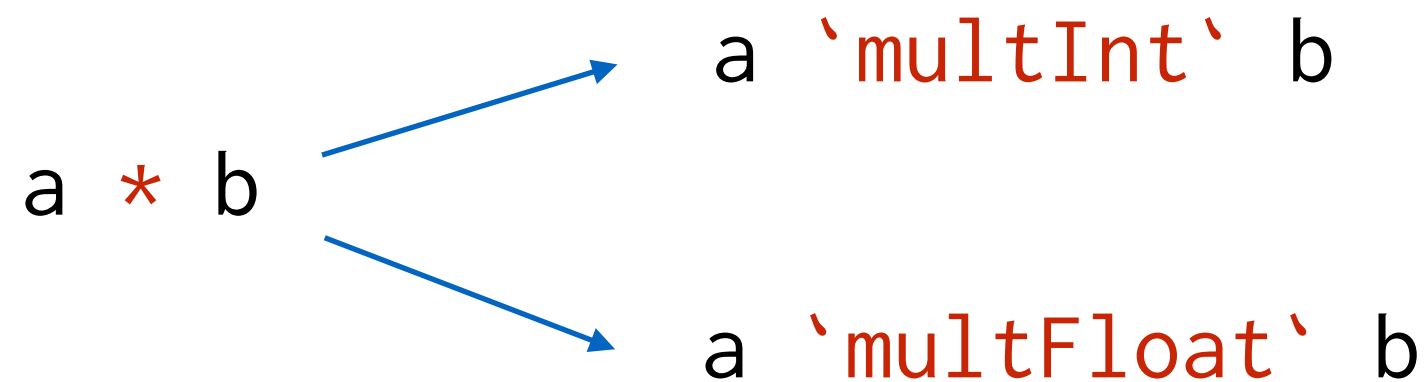
a * b  a 'multInt' b

- Don't allow for overloading functions defined from them

➤ **Problem?**

Non-solution: local choice

- Overload basic operators such as + and *

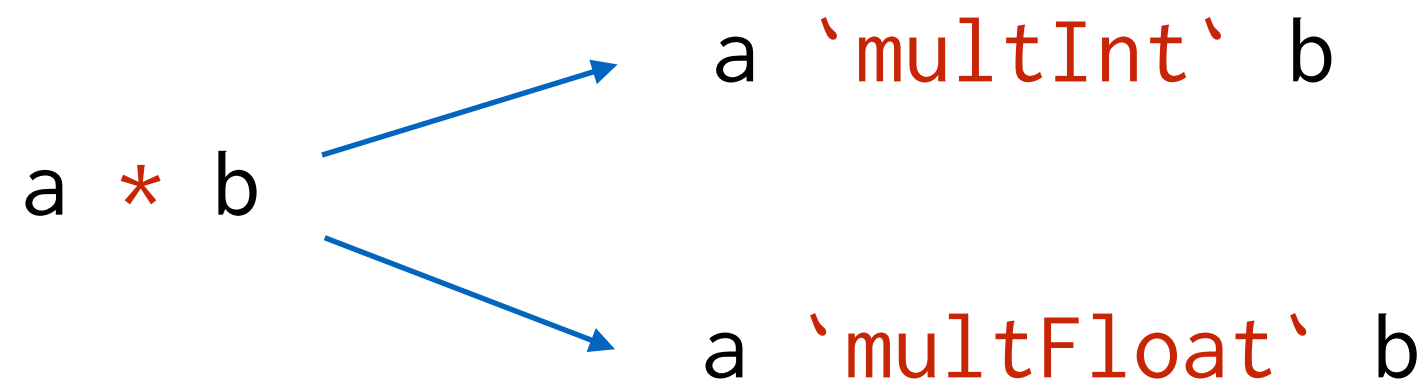


- Don't allow for overloading functions defined from them

➤ **Problem?**

Non-solution: local choice

- Overload basic operators such as + and *



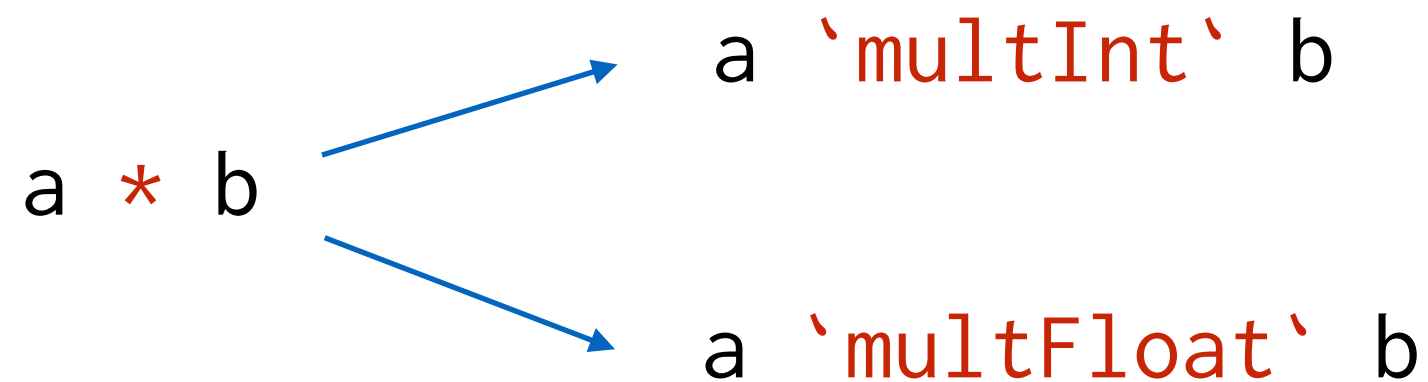
- Don't allow for overloading functions defined from them

➤ Problem?

first usage tells us that
`square :: Int -> Int`

Non-solution: local choice

- Overload basic operators such as + and *



- Don't allow for overloading functions defined from them

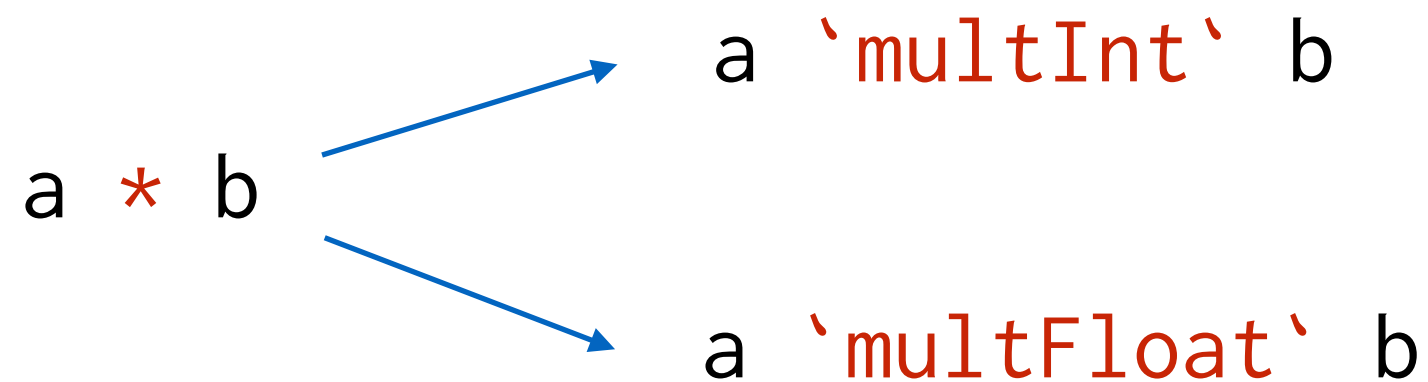
➤ Problem?

```
square x = x*x  
square 3  
square 3.14
```

first usage tells us that
`square :: Int -> Int`

Non-solution: local choice

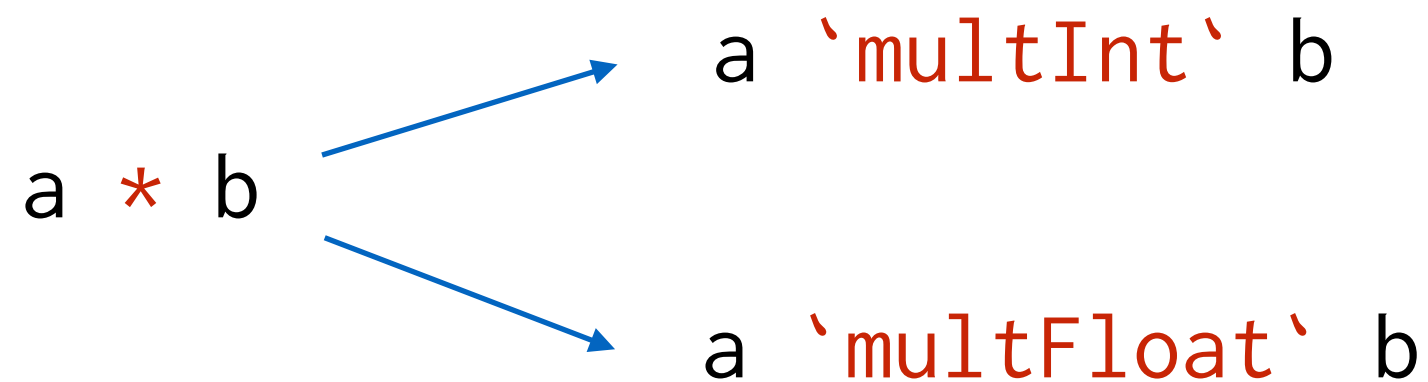
- Overload basic operators such as + and *



- Allow for overloading functions defined from them
 - **Problem?**

Non-solution: local choice

- Overload basic operators such as + and *



- Allow for overloading functions defined from them
 - **Problem?** `ssquare x y = (square x, square y)`
`ssquare 3 4`
`ssquare 3.3 4`
`...`
Code blowup!

Non-solution: fully polymorphic

- Make functions like `==` fully polymorphic
 - `(==) :: a -> a -> Bool`
- At runtime: compare underlying representation
 - `3*3 == 9 => ??`
 - `(\x -> x) == (\x -> x + 1) => ??`
 - `Left 3 == Right "44" => ??`
- **Problems?**

Non-solution: fully polymorphic

- Make functions like `==` fully polymorphic
 - `(==) :: a -> a -> Bool`
- At runtime: compare underlying representation
 - `3*3 == 9 => ??`
 - `(\x -> x) == (\x -> x + 1) => ??`
 - `Left 3 == Right "44" => ??`
- **Problems? Breaks abstraction!**

Non-solution: “eqtype” polymorphism [SML]

- Make equality polymorphic in a limited way
 - $(==) :: a == \rightarrow a == \rightarrow \text{Bool}$
 - $\text{member} :: a == \rightarrow [a ==] \rightarrow \text{Bool}$
- $a ==$ are special type variables restricted to types with equality
- **Problem?**

Solution: type classes



Solution: type classes

- **Idea:** generalize eqtypes to arbitrary types
- Provide concise types to describe overloaded functions
 - Solves:
- Allow users to define functions using overloaded ones
 - Solves:
- Allow users to declare new collections of overloaded functions

Solution: type classes

- **Idea:** generalize eqtypes to arbitrary types
- Provide concise types to describe overloaded functions
 - Solves: exponential blow up
- Allow users to define functions using overloaded ones
 - Solves:
- Allow users to declare new collections of overloaded functions

Solution: type classes

- **Idea:** generalize eqtypes to arbitrary types
- Provide concise types to describe overloaded functions
 - Solves: exponential blow up
- Allow users to define functions using overloaded ones
 - Solves: monomorphism
- Allow users to declare new collections of overloaded functions

Back to our old examples

- `square :: a -> a`
- `sort :: [a] -> [a]`
- `Show :: a -> String`
- `member :: a -> [a] -> Bool`

Back to our old examples

- `square :: Num a => a -> a`
- `sort :: [a] -> [a]`
- `Show :: a -> String`
- `member :: a -> [a] -> Bool`

Back to our old examples

- `square :: Num a => a -> a`
- `sort :: Ord a => [a] -> [a]`
- `Show :: a -> String`
- `member :: a -> [a] -> Bool`

Back to our old examples

- `square :: Num a => a -> a`
- `sort :: Ord a => [a] -> [a]`
- `Show :: Show a => a -> String`
- `member :: a -> [a] -> Bool`

Back to our old examples

- `square :: Num a => a -> a`
- `sort :: Ord a => [a] -> [a]`
- `Show :: Show a => a -> String`
- `member :: Eq a => a -> [a] -> Bool`

Type classes

- **Class declaration:** what are the Num operations?
- **Instance declaration:** how are the ops implemented?

Type classes

- **Class declaration:** what are the Num operations?

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

- **Instance declaration:** how are the ops implemented?

Type classes

- **Class declaration:** what are the Num operations?

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

- **Instance declaration:** how are the ops implemented?

```
instance Num Int where
  (+) a b = plusInt a b
  (*) a b = mulInt a b
  ...

instance Num Float where
  (+) a b = plusFloat a b
  (*) a b = mulFloat a b
  ...
```

Type classes

- **Basic usage:** how do you use the overloaded ops?
 - $3 + 4$
 - $3.3 + 4.4$
 - `"4" + "5"`
- Functions using these ops can be polymorphic too
 - E.g., `square ::`
`square x = x * x`

Type classes

- **Basic usage:** how do you use the overloaded ops?
 - $3 + 4$
 - $3.3 + 4.4$
 - `"4" + "5"`
- Functions using these ops can be polymorphic too
 - E.g., $\text{square} :: \text{Num } x \Rightarrow x \rightarrow x$
 $\text{square } x = x * x$

type-classes-use.hs

Type classes can have subclasses

- Example: consider Eq and Ord classes
 - Eq:
 - Ord:
- Subclass declaration can express relationship:
 - E.g., `class Eq a => Ord a where ...`
- When you declare functions you just need to specify Ord, we know that it must also be Eq

Type classes can have subclasses

- Example: consider Eq and Ord classes
 - Eq: allow for equality checking
 - Ord:
- Subclass declaration can express relationship:
 - E.g., `class Eq a => Ord a where ...`
- When you declare functions you just need to specify Ord, we know that it must also be Eq

Type classes can have subclasses

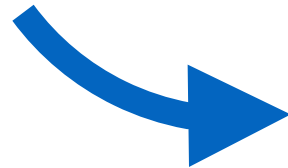
- Example: consider Eq and Ord classes
 - Eq: allow for equality checking
 - Ord: allow for comparing elements of the type
- Subclass declaration can express relationship:
 - E.g., `class Eq a => Ord a where ...`
- When you declare functions you just need to specify Ord, we know that it must also be Eq

How do type classes work?

- Basic idea:

square :: Num x => x -> x

square x = x * x



- Intuition from C's qsort:

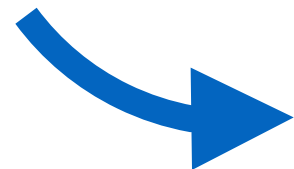
```
void qsort(void *base, size_t nel, size_t width,  
           int (*compar)(const void *, const void *));
```

- Pass operator as argument!

How do type classes work?

- Basic idea:

```
square :: Num x => x -> x  
square x = x * x
```



```
square :: Num x -> x -> x  
square dic x = (*) dic x x
```

- Intuition from C's qsort:

```
void qsort(void *base, size_t nel, size_t width,  
           int (*compar)(const void *, const void *));
```

- Pass operator as argument!

How do type classes work?

- **Class declaration:** desugar to dictionary type decl

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



- **Instance declaration:** desugar to dictionary values

```
instance Num Int where  
  (+) a b = plusInt a b  
  (*) a b = mulInt a b  
  ...
```



How do type classes work?


- **Class declaration:** desugar to dictionary type decl

<code>class</code> Num a <code>where</code>	<code>data</code> Num a = MkNumDict
<code>(+)</code> :: a -> a -> a	(a -> a -> a)
<code>(*)</code> :: a -> a -> a	(a -> a -> a)
...	...



- **Instance declaration:** desugar to dictionary values

```
instance Num Int where  
  (+) a b = plusInt a b  
  (*) a b = mulInt a b  
  ...
```



How do type classes work?


- **Class declaration:** desugar to dictionary type decl

<code>class</code> Num a <code>where</code>	<code>data</code> Num a = MkNumDict
(+) :: a -> a -> a	(a -> a -> a)
(*) :: a -> a -> a	(a -> a -> a)
...	...



- **Instance declaration:** desugar to dictionary values

<code>instance</code> Num Int <code>where</code>	dictNumInt = MkNumDict
(+) a b = plusInt a b	plusInt
(*) a b = mulInt a b	mulInt
...	



How do type classes work?

- Basic usage: whenever you use operator you must pass it a dictionary value:
 - E.g., `(*)`
 - E.g., `(==)`
- Defining polymorphic functions: always take dictionary values, so type and definition must reflect this
 - E.g.,
 - E.g.,

How do type classes work?

- Basic usage: whenever you use operator you must pass it a dictionary value:
 - E.g., `(*) dictNumInt 4 5`
 - E.g., `(==) dictEqFloat 3.3 5.5`
- Defining polymorphic functions: always take dictionary values, so type and definition must reflect this
 - E.g.,
 - E.g.,

How do type classes work?

- Basic usage: whenever you use operator you must pass it a dictionary value:
 - E.g., `(*) dictNumInt 4 5`
 - E.g., `(==) dictEqFloat 3.3 5.5`
- Defining polymorphic functions: always take dictionary values, so type and definition must reflect this
 - E.g., `square :: Num x -> x -> x`
`square dict x = (*) dict x`
 - E.g.,

How do type classes work?

- Basic usage: whenever you use operator you must pass it a dictionary value:
 - E.g., `(*) dictNumInt 4 5`
 - E.g., `(==) dictEqFloat 3.3 5.5`
- Defining polymorphic functions: always take dictionary values, so type and definition must reflect this
 - E.g., `square :: Num x => x -> x`
`square dict x = (*) dict x`
 - E.g., `square dictNumFloat 4.4`

type-classes-desugar.hs

How does this affect type inference?

- Type inference infers a qualified type: $Q \Rightarrow \tau$
- τ is ordinary Hindley-Miner type, inferred as usual
- Q is a constraint set/set of type class predicates
- Consider:

$$\begin{array}{ll} f :: & \Rightarrow a \rightarrow \text{Bool} \\ f\ x = x + 2 == 3 \end{array}$$

How does this affect type inference?

- Type inference infers a qualified type: $Q \Rightarrow \tau$
- τ is ordinary Hindley-Miner type, inferred as usual
- Q is a constraint set/set of type class predicates
- Consider:

$$f :: (Eq\ a, Num\ a) \Rightarrow a \rightarrow Bool$$
$$f\ x = x + 2 == 3$$

Modification to our TI algorithm

- Modify the “Generate constraints” step to include type class constraints
- Simplify constraint set in final step

Generate constraints

- Example: $f\ x\ y = x == y$
 - Assign τ_0 to x
 - Assign τ_1 to y
 - Constraints:
 -
 -

Simplify constraints

- Eliminate duplicates:
 - $\{\text{Num } a, \text{Num } a\} =$
- Use more general instance declaration
 - $\{\text{Eq } [a], \text{Eq } a\} =$
- Use sub-class declaration declaration
 - $\{\text{Ord } a, \text{Eq } a\} =$
- Example: $\{\text{Eq } a, \text{Eq } [a], \text{Ord } a\} =$

Simplify constraints

- Eliminate duplicates:
 - $\{\text{Num } a, \text{Num } a\} = \{\text{Num } a\}$
- Use more general instance declaration
 - $\{\text{Eq } [a], \text{Eq } a\} =$
- Use sub-class declaration declaration
 - $\{\text{Ord } a, \text{Eq } a\} =$
- Example: $\{\text{Eq } a, \text{Eq } [a], \text{Ord } a\} =$

Simplify constraints

- Eliminate duplicates:
 - $\{\text{Num } a, \text{Num } a\} = \{\text{Num } a\}$
- Use more general instance declaration
 - $\{\text{Eq } [a], \text{Eq } a\} = \{\text{Eq } a\}$ if instance $\text{Eq } a \Rightarrow \text{Eq } [a]$
- Use sub-class declaration declaration
 - $\{\text{Ord } a, \text{Eq } a\} =$
- Example: $\{\text{Eq } a, \text{Eq } [a], \text{Ord } a\} =$

Simplify constraints

- Eliminate duplicates:
 - $\{\text{Num } a, \text{Num } a\} = \{\text{Num } a\}$
- Use more general instance declaration
 - $\{\text{Eq } [a], \text{Eq } a\} = \{\text{Eq } a\}$ if instance $\text{Eq } a \Rightarrow \text{Eq } [a]$
- Use sub-class declaration declaration
 - $\{\text{Ord } a, \text{Eq } a\} = \{\text{Ord } a\}$ if class $\text{Eq } a \Rightarrow \text{Ord } a$
- Example: $\{\text{Eq } a, \text{Eq } [a], \text{Ord } a\} =$

Simplify constraints

- Eliminate duplicates:
 - $\{\text{Num } a, \text{Num } a\} = \{\text{Num } a\}$
- Use more general instance declaration
 - $\{\text{Eq } [a], \text{Eq } a\} = \{\text{Eq } a\}$ if instance $\text{Eq } a \Rightarrow \text{Eq } [a]$
- Use sub-class declaration declaration
 - $\{\text{Ord } a, \text{Eq } a\} = \{\text{Ord } a\}$ if class $\text{Eq } a \Rightarrow \text{Ord } a$
- Example: $\{\text{Eq } a, \text{Eq } [a], \text{Ord } a\} = \{\text{Ord } a\}$

Are these the same as in OO?

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

?
=

```
interface Show {  
  String show();  
}
```

Are these the same as in OO?

<pre>class String a where show :: a -> String</pre>	<pre>? =</pre>	<pre>interface Show { String show(); }</pre>
type-based dispatch	vs	value-based dispatch

Summary

- Type classes are a good approach to the overloading
- They provide a form of polymorphism: ad-hoc
- More flexible than designers first realized
 - The type-driven, dictionary approach
- Not the same as OO classes/interfaces