Type classes

Deian Stefan

(adopted from my & Edward Yang’s CSE242 slides)
A problem w/ parametric polymorphism

- Consider the list member function:

  \[
  \begin{align*}
  \text{member} \ x \ [] & = \text{False} \\
  \text{member} \ x \ (y:ys) & = \text{if} \ x == y \\
  & \quad \text{then True} \\
  & \quad \text{else member} \ x \ ys
  \end{align*}
  \]

- Is the type \( \text{member} :: \ a \to [a] \to \text{Bool} \) 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
  - 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 *

\[ a \times b \]

• Don’t allow for overloading functions defined from them

\[
\begin{align*}
\text{square} & \quad x = x \times x \\
\text{square} & \quad 3 \\
\text{square} & \quad 3.14
\end{align*}
\]

➤ Problem?
Non-solution: local choice

• Overload basic operators such as + and *

  \[ a \times b \]

  \[ a \ `\text{multInt}` \ b \]

• Don’t allow for overloading functions defined from them

  \[ \text{square} \ x = x \times x \]

  \[ \text{Problem?} \]

  \[ \text{square} \ 3 \]

  \[ \text{square} \ 3.14 \]
Non-solution: local choice

- Overload basic operators such as + and *
  
  $a \times b$
  
  a `multInt` b

  a `multFloat` b

- Don’t allow for overloading functions defined from them

Problem?

```haskell
square x = x**x
```

square 3

square 3.14
Non-solution: local choice

• Overload basic operators such as + and *
  
  \[
  a \times b \\
  \]

  \[
  \rightarrow a \ `\text{multInt}\` b \\
  a \ `\text{multFloat}\` b \\
  \]

• Don’t allow for overloading functions defined from them

  
  \[
  \text{square} \ x = x \times x \\
  \]

  \[
  \text{square} 3 \\
  \text{square} 3.14 \\
  \]

  \[
  \text{first usage tells us that} \\
  \text{square} :: \text{Int} \to \text{Int} \\
  \]
Non-solution: local choice

• Overload basic operators such as $+$ and $\times$

  a $\times$ b
    ▶ a `multInt` b
    ▶ a `multFloat` b

• Allow for overloading functions defined from them

  ➤ Problem? square x y = (square x, square y)
square 3 4
square 3.3 4
...

...
Non-solution: fully polymorphic

• Make functions like \(\text{==}\) fully polymorphic
  
  \[ \text{==} :: \ a \rightarrow \ a \rightarrow \text{Bool} \]

• At runtime: compare underlying representation
  
  \[ 3 \times 3 \ \text{==} \ 9 \ \Rightarrow \ ??? \]

  \[ (\times\ x) \ \text{==} \ (\times\ x \ +\ 1) \ \Rightarrow \ ??? \]

  \[ \text{Left}\ 3 \ \text{==} \ \text{Right} \ \text{“}44\text{”} \ \Rightarrow \ ??? \]

• Problem?
Non-solution: “eqtype” polymorphism [SML]

- Make equality polymorphic in a limited way
  - $(==) :: \text{a}== \to \text{a}== \to \text{Bool}$
  - member :: a== \to [a==] \to \text{Bool}

- $\text{a}==$ 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 :: Num a => a -> a
- sort :: Ord a => [a] -> [a]
- serialize :: Show a => a -> ByteString
- member :: Eq a => a -> [a] -> Bool
Type classes

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

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

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

```haskell
instance Num Int where
    (+) a b = plusInt a b
    (*) a b = mulInt 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.,  
  
  \[
  \text{square} :: \text{Num } x \Rightarrow x \rightarrow x \\
  \text{square } x = x \times x
  \]
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 ...`

- 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:

\[
square :: \text{Num} \ x \Rightarrow x \rightarrow x
\]
\[
square \ x = x \times x
\]

• Intuition from C’s qsort:

\[
\text{void qsort}(\text{void } *\text{base}, \text{size_t } \text{nel}, \text{size_t } \text{width}, \text{int } (*\text{compar})(\text{const void } *, \text{const void } *)));
\]

➤ Pass operator as argument!
How do type classes work?

• Basic idea:

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

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

• Intuition from C’s qsort:

```c
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

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

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

```haskell
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

```
class Num a where
    data Num a = MkNumDict
    (+) :: 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

```haskell
class Num a where
data Num a = MkNumDict
  (+) :: a -> a -> a
  (*) :: a -> a -> a
  ...
```

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

```haskell
instance Num Int where
dictNumInt = MkNumDict
  (+) a b = plusInt a b
  (*) a b = mulInt a b
  ...
```
How do type classes work?

- Basic usage: whenever you use operator you must pass it a dictionary value:
  - E.g., \((\ast)\) `dictNumInt 4 5`
  - E.g., \((==)\) `dictEqFloat 3.3 5.5`

- Defining polymorphic functions: always take dictionary values, so type and definition must reflect
  - E.g., `square :: Num x \rightarrow x \rightarrow x`
    ```
square dict x = (\ast) dict x
    ```
  - E.g., `square dictNumFloat 4.4`
type-classes-1.hs
How does this affect type inference?

• Type inference infers a qualified type: $Q \Rightarrow \tau$

• $\tau$ is ordinary Hindley-Miner type, inferred as usual

• $Q$ is a constraint set/set of type class predicates

• Consider:

```haskell
f :: (Eq a, Num a) => a -> 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 \( \tau_0 \) to \( x \)

  ➤ Assign \( \tau_1 \) to \( y \)

  ➤ Constraints:
Generate constraints

- Example: \( f \ x \ y = x == y \)
  - Assign \( \tau_0 \) to \( x \)
  - Assign \( \tau_1 \) to \( y \)
  - Constraints:
    - \( \{ \text{Eq } \tau_0 \} \)
    - \( \tau_0 = \tau_1 \)
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\} \text{ 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\} \text{ if instance Eq } a \implies \text{Eq } [a]

• Use sub-class declaration declaration

  ➤ \{\text{Ord } a, \text{ Eq } a\} = \{\text{Ord } a\} \text{ if class Eq } a \implies \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\} \text{ if instance } \text{Eq } a \Rightarrow \text{Eq } [a]

• Use sub-class declaration declaration

  ➤ \{\text{Ord } a, \text{Eq } a\} = \{\text{Ord } a\} \text{ 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();
}

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