Lesson 26

• Type classes and Overloading in Haskell
• Constructor Classes
Polymorphism

- The ability of associating a single interface with entities of different types
- We focus on *polymorphic functions*, applicable to arguments of different types

**Diagram:**
- Polymorphism
  - Universal
    - Parametric
    - Inclusion
  - Ad hoc
    - Overloading
Polymorphism vs Overloading

• Parametric polymorphism
  – Single algorithm may be given many types
  – Type variable may be replaced by any type
  – If \( f::t \to t \) then \( f::\text{Int} \to \text{Int}, f::\text{Bool} \to \text{Bool}, ... \)

• Overloading
  – A single symbol may refer to more than one algorithm.
  – Each algorithm may have different type.
  – Choice of algorithm determined by type context.
  – \( + \) has types \( \text{Int} \to \text{Int} \to \text{Int} \) and \( \text{Float} \to \text{Float} \to \text{Float} \),
    but not \( t \to t \to t \) for arbitrary \( t \).
Why Overloading?

- Many useful functions are not parametric
- Can list membership work for any type?
  
  ```haskell
  member :: [w] -> w -> Bool
  ```
  
  – No! Only for types w for that support equality.
- Can list sorting work for any type?
  
  ```haskell
  sort :: [w] -> [w]
  ```
  
  – No! Only for types w that support ordering.
Overloading Arithmetic, Take 1

• Allow functions containing overloaded symbols to define multiple functions:

  ```
  square x = x * x       -- legal
  -- Defines two versions:
  -- Int -> Int and Float -> Float
  ```

• But consider:

  ```
  squares (x,y,z) =
      (square x, square y, square z)
  -- There are 8 possible versions!
  ```

• Approach not widely used because of exponential growth in number of versions.
Overloading Arithmetic, Take 2

• Basic operations such as + and * can be overloaded, but not functions defined from them

<table>
<thead>
<tr>
<th>Expression</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>3 * 3</td>
<td>-- legal</td>
</tr>
<tr>
<td>3.14 * 3.14</td>
<td>-- legal</td>
</tr>
<tr>
<td>square x = x * x</td>
<td>-- Int -&gt; Int</td>
</tr>
<tr>
<td>square 3</td>
<td>-- legal</td>
</tr>
<tr>
<td>square 3.14</td>
<td>-- illegal</td>
</tr>
</tbody>
</table>

• Standard ML uses this approach.

• Not satisfactory: Programmer cannot define functions that implementation might support
Overloading Equality, Take 1

• Equality defined only for types that admit equality: types not containing function or abstract types.

  3 * 3 == 9          -- legal
  'a' == 'b'          -- legal
  \x->x == \y->y+1    -- illegal

• Overload equality like arithmetic ops + and * in SML.
• But then we can’t define functions using ‘==‘:

  member [] y       = False
  member (x:xs) y   = (x==y) || member xs y

  member [1,2,3] 3     -- ok if default is Int
  member "Haskell" 'k' -- illegal

• Approach adopted in first version of SML.
Overloading Equality, Take 2

• Make type of equality fully polymorphic

\[
(==) :: a \to a \to \text{Bool}
\]

• Type of list membership function

\[
\text{member} :: [a] \to a \to \text{Bool}
\]

• **Miranda** used this approach.
  – Equality applied to a **function** yields a runtime error
  – Equality applied to an **abstract type** compares the underlying representation, which violates abstraction principles
Overloading Equality, Take 3

- Make equality polymorphic **in a limited way**:
  
  \[
  (==) :: a(==) \rightarrow a(==) \rightarrow \text{Bool}
  \]

  where \(a(==)\) is type variable restricted to types with equality

- Now we can type the member function:

  
  ```
  member :: a(==) \rightarrow [a(==)] \rightarrow \text{Bool}
  member 4 [2,3] :: \text{Bool}
  member 'c' ['a', 'b', 'c'] :: \text{Bool}
  member (\y->y \times 2) [\x->x, \x->x + 2] -- type error
  ```

- Approach used in SML today, where the type \(a(==)\) is called an “eqtype variable” and is written "'a."
Type Classes

• Type classes solve these problems
  – Provide concise types to describe overloaded functions, so no exponential blow-up
  – Allow users to define functions using overloaded operations, eg, square, squares, and member
  – Allow users to declare new collections of overloaded functions: equality and arithmetic operators are not privileged built-ins
  – Generalize ML’s eqtypes to arbitrary types
  – Fit within type inference framework
Intuition

• A function to sort lists can be passed a comparison operator as an argument:

\[
\text{qsort} :: (a \rightarrow a \rightarrow \text{Bool}) \rightarrow [a] \rightarrow [a]
\]
\[
\text{qsort } \text{cmp } [] = []
\]
\[
\text{qsort } \text{cmp } (x:xs) = \text{qsort } \text{cmp } (\text{filter } (\text{cmp } x) \; xs)
\]
\[
\quad + [x] +
\]
\[
\quad \text{qsort } \text{cmp } (\text{filter } (\text{not.cmp } x) \; xs)
\]

– This allows the function to be parametric

• We can built on this idea ...
Intuition (continued)

• Consider the “overloaded” parabola function

\[ \text{parabola} \ x = (x \times x) + x \]

• We can rewrite the function to take the operators it contains as an argument

\[ \text{parabola}' \ (\text{plus}, \text{times}) \ x = \text{plus} \ (\text{times} \ x \ x) \ x \]

  – The extra parameter is a “dictionary” that provides implementations for the overloaded ops.

  – We have to rewrite all calls to pass appropriate implementations for plus and times:

\[ y = \text{parabola}' \ (\text{intPlus}, \text{intTimes}) \ 10 \]
\[ z = \text{parabola}' \ (\text{floatPlus}, \text{floatTimes}) \ 3.14 \]
Systematic programming style

-- Dictionary type
data MathDict a = MkMathDict (a->a->a) (a->a->a)

-- Accessor functions
get_plus :: MathDict a -> (a->a->a)
get_plus (MkMathDict p t) = p

get_times :: MathDict a -> (a->a->a)
get_times (MkMathDict p t) = t

-- "Dictionary-passing style"
parabola :: MathDict a -> a -> a
parabola dict x = let plus = get_plus dict
                  times = get_times dict
                  in plus (times x x) x

Type class declarations will generate Dictionary type and selector functions
Systematic programming style

-- Dictionary type
data MathDict a = MkMathDict (a->a->a) (a->a->a)

-- Dictionary construction
intDict = MkMathDict intPlus intTimes
floatDict = MkMathDict floatPlus floatTimes

-- Passing dictionaries
y = parabola intDict 10
z = parabola floatDict 3.14

Type class instance declarations produce instances of the Dictionary

Compiler will add a dictionary parameter and rewrite the body as necessary
Type Class Design Overview

• Type class declarations
  – Define a set of operations, give the set a name
  – Example: \( \text{Eq} \ a \) type class
    • operations \( == \) and \( \mid = \) with \( \text{type} \ a \to a \to \text{Bool} \)

• Type class instance declarations
  – Specify the implementations for a particular type
  – For \( \text{Int} \) instance, \( == \) is defined to be integer equality

• Qualified types (or Type Constraints)
  – Concisely express the operations required on otherwise polymorphic type

\[
\text{member} :: \text{Eq} \ w \to w \to [w] \to \text{Bool}
\]
“for all types \( w \) that support the \( \text{Eq} \) operations”

**Qualified Types**

If a function works for every type with particular properties, the type of the function says just that:

\[
\text{Member} :: \text{Eq } w \Rightarrow w \rightarrow [w] \rightarrow \text{Bool}
\]

Otherwise, it must work for any type whatsoever

\[
\text{sort} :: \text{Ord } a \Rightarrow [a] \rightarrow [a] \\
\text{serialise} :: \text{Show } a \Rightarrow a \rightarrow \text{String} \\
\text{square} :: \text{Num } n \Rightarrow n \rightarrow n \\
\text{squares} :: (\text{Num } t, \text{Num } t1, \text{Num } t2) \Rightarrow (t, t1, t2) \rightarrow (t, t1, t2)
\]

\[
\text{reverse} :: [a] \rightarrow [a] \\
\text{filter} :: (a \rightarrow \text{Bool}) \rightarrow [a] \rightarrow [a]
\]
Type Classes

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

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

instance Num Int where
  a + b = intPlus a b
  a * b = intTimes a b
  negate a = intNeg a
  ...etc...

FORGET all you know about OO classes!

The class declaration says what the Num operations are.

An instance declaration for a type T says how the Num operations are implemented on T's

Works for any type 'n' that supports the Num operations.

intPlus :: Int -> Int -> Int
intTimes :: Int -> Int -> Int
e tc, defined as primitives
Compiling Overloaded Functions

When you write this...

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

...the compiler generates this

\[
\text{square} :: \text{Num } n \rightarrow n \rightarrow n
\]
\[
\text{square } d \times x = (\ast) \times d \times x
\]

The “\text{Num } n \Rightarrow” turns into an extra value argument to the function. It is a value of data type \text{Num } n and it represents a dictionary of the required operations.

A value of type \text{(Num } n) is a dictionary of the Num operations for type \text{n}.
Compiling Type Classes

When you write this...

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

...the compiler generates this

```
square :: Num n -> n -> n
square d x = (* d x x
```

```
class Num n where
  (+)    :: n -> n -> n
  (*)    :: n -> n -> n
  negate :: n -> n
  ...etc...
```

```
data Num n
    = MkNum (n -> n -> n)
      (n -> n -> n)
      (n -> n)
      ...etc...

  (*) :: Num n -> n -> n -> n
  (*) (MkNum _ m _ ...) = m
```

The class decl translates to:
A data type decl for Num
A selector function for each class operation

A value of type (Num n) is a dictionary of the Num operations for type n
Compiling Instance Declarations

When you write this...

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

...the compiler generates this

```
square :: Num n -> n -> n
square d x = (*) d x x
```

```
instance Num Int where
  a + b = intPlus a b
  a * b = intTimes a b
  negate a = intNeg a
  ...etc...
```

```
dNumInt :: Num Int
dNumInt = MkNum intPlus intTimes intNeg ...
```

An instance decl for type T translates to a value declaration for the Num dictionary for T

A value of type (Num n) is a dictionary of the Num operations for type n
Implementation Summary

• The compiler translates each function that uses an overloaded symbol into a function with an extra parameter: the dictionary.
• References to overloaded symbols are rewritten by the compiler to lookup the symbol in the dictionary.
• The compiler converts each type class declaration into a dictionary type declaration and a set of selector functions.
• The compiler converts each instance declaration into a dictionary of the appropriate type.
• The compiler rewrites calls to overloaded functions to pass a dictionary. It uses the static, qualified type of the function to select the dictionary.
Functions with Multiple Dictionaries

squares :: (Num a, Num b, Num c) => (a, b, c) -> (a, b, c)
squares(x,y,z) = (square x, square y, square z)

Pass appropriate dictionary on to each square function.

Note the concise type for the squares function!

squares :: (Num a, Num b, Num c) -> (a, b, c) -> (a, b, c)
squares (da,db,dc) (x, y, z) =
  (square da x, square db y, square dc z)
Compositionality

Overloaded functions can be defined from other overloaded functions:

\[
\text{sumSq} :: \text{Num} \ n \Rightarrow n \rightarrow n \rightarrow n \\
\text{sumSq} \ x \ y = \text{square} \ x + \text{square} \ y
\]

Extract addition operation from d

\[
\text{sumSq} :: \text{Num} \ n \rightarrow n \rightarrow n \rightarrow n \\
\text{sumSq} \ d \ x \ y = (+) \ d \ (\text{square} \ d \ x) \ (\text{square} \ d \ y)
\]

Pass on d to square
Compositionality

Build compound instances from simpler ones:

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

instance Eq Int where
  (==) = intEq -- intEq primitive equality

instance (Eq a, Eq b) => Eq(a,b)
  (u,v) == (x,y) = (u == x) && (v == y)

instance Eq a => Eq [a] where
  (==) []     []     = True
  (==) (x:xs) (y:ys) = x==y && xs == ys
  (==) _      _      = False
Compound Translation

Build compound instances from simpler ones.

class Eq a where
  (==) :: a -> a -> Bool
instance Eq a => Eq [a] where
  (==) [] [] = True
  (==) (x:xs) (y:ys) = x==y && xs == ys
  (==) _ _ = False

data Eq = MkEq (a->a->Bool)  -- Dictionary type
  (==) (MkEq eq) = eq
dEqList :: Eq a -> Eq [a]  -- Selector
dEqList d = MkEq eql
  where
    eql [] [] = True
    eql (x:xs) (y:ys) = (==) d x y && eql xs ys
    eql _ _ = False
Many Type Classes

- **Eq**: equality
- **Ord**: comparison
- **Num**: numerical operations
- **Show**: convert to string
- **Read**: convert from string
- **Testable, Arbitrary**: testing.
- **Enum**: ops on sequentially ordered types
- **Bounded**: upper and lower values of a type
- Generic programming, reflection, monads, ...
- And many more.
Subclasses

• We could treat the Eq and Num type classes separately

```haskell
memsq :: (Eq a, Num a) => a -> [a] -> Bool
memsq x xs = member (square x) xs
```

– But we expect any type supporting Num to also support Eq

• A subclass declaration expresses this relationship:

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

• With that declaration, we can simplify the type of the function

```haskell
memsq :: Num a => a -> [a] -> Bool
memsq x xs = member (square x) xs
```
Default Methods

- Type classes can define “default methods”

```haskell
-- Minimal complete definition:  
--     (==) or (/=)
class Eq a where
    (==) :: a -> a -> Bool
    x == y  =  not (x /= y)
    (/=) :: a -> a -> Bool
    x /= y  =  not (x == y)
```

- Instance declarations can override default by providing a more specific definition.
Deriving

• For Read, Show, Bounded, Enum, Eq, and Ord, the compiler can generate instance declarations automatically

```haskell
data Color = Red | Green | Blue
  deriving (Show, Read, Eq, Ord)
```

```haskell
Main> show Red
"Red"
Main> Red < Green
True
Main> let c :: Color = read "Red"
Main> c
Red
```

— *Ad hoc* : derivations apply only to types where derivation code works
Numeric Literals

class Num a where
  (+) :: a -> a -> a
  (-) :: a -> a -> a
  fromInteger :: Integer -> a
...

inc :: Num a => a -> a
inc x = x + 1

Even literals are overloaded.
1 :: (Num a) => a

“1” means “fromInteger 1”

Advantages:
- Numeric literals can be interpreted as values of any appropriate numeric type
- Example: 1 can be an Integer or a Float or a user-defined numeric type.
Example: Complex Numbers

• We can define a data type of complex numbers and make it an instance of `Num`.

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

```haskell
data Cpx a = Cpx a a
  deriving (Eq, Show)

instance Num a => Num (Cpx a) where
  (Cpx r1 i1) + (Cpx r2 i2) = Cpx (r1+r2) (i1+i2)
  fromInteger n = Cpx (fromInteger n) 0
...
Example: Complex Numbers

• And then we can use values of type \texttt{Cpx} in any context requiring a \texttt{Num}:

```
data Cpx a = Cpx a a

c1 = 1 :: Cpx Int
c2 = 2 :: Cpx Int
c3 = c1 + c2

parabola x = (x * x) + x
c4 = parabola c3
i1 = parabola 3
```
Type Inference

• Type inference infers a qualified type Q => T
  – T is a Hindley Milner type, inferred as usual
  – Q is set of type class predicates, called a constraint

• Consider the example function:

```haskell
example z xs =
  case xs of
    []  -> False
    (y:ys) -> y > z || (y==z && ys == [z])
```

  – Type T is  a -> [a] -> Bool
  – Constraint Q is  { Ord a, Eq a, Eq [a]}

```

Ord a  because    y>z
Eq a   because    y==z
Eq [a] because    ys == [z]
Type Inference

• Constraint sets Q can be simplified:
  – Eliminate duplicates
    • \{\text{Eq } a, \text{Eq } a\} simplifies to \{\text{Eq } a\}
  – Use an **instance declaration**
    • If we have instance \text{Eq } a =\rightarrow \text{Eq } [a],
      • then \{\text{Eq } a, \text{Eq } [a]\} simplifies to \{\text{Eq } a\}
  – Use a **class declaration**
    • If we have class \text{Eq } a =\rightarrow \text{Ord } a \text{ where } ...,
      • then \{\text{Ord } a, \text{Eq } a\} simplifies to \{\text{Ord } a\}

• Applying these rules,
  – \{\text{Ord } a, \text{Eq } a, \text{Eq}[a]\} simplifies to \{\text{Ord } a\}
Type Inference

• Putting it all together:

```haskell
example z xs =
  case xs of
    []    -> False
    (y:ys) -> y > z || (y==z && ys ==[z])
```

- \( T = a \to [a] \to \text{Bool} \)
- \( Q = \{\text{Ord } a, \text{Eq } a, \text{Eq } [a]\} \)
- \( Q \) simplifies to \( \{\text{Ord } a\} \)
- `example :: \{\text{Ord } a\} => a \to [a] \to \text{Bool}`
Detecting Errors

• Errors are detected when predicates are known not to hold:

Prelude> 'a' + 1
   No instance for (Num Char)
   arising from a use of `+' at <interactive>:1:0-6
   Possible fix: add an instance declaration for (Num Char)
   In the expression: 'a' + 1
   In the definition of `it': it = 'a' + 1

Prelude> (\x -> x)
   No instance for (Show (t -> t))
   arising from a use of `print' at <interactive>:1:0-4
   Possible fix: add an instance declaration for (Show (t -> t))
   In the expression: print it
   In a stmt of a 'do' expression: print it
More Type Classes: Constructors

• **Type Classes** are predicates over **types**

• **Constructor Classes** are predicates over **type constructors**

• Example: Map function useful on many Haskell types

• Lists:

```haskell
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = f x : map f xs

result = map (\x->x+1) [1,2,4]
```
Constructor Classes

• More examples of map function

```haskell
data Tree a = Leaf a | Node(Tree a, Tree a)
deriving Show

mapTree :: (a -> b) -> Tree a -> Tree b
mapTree f (Leaf x) = Leaf (f x)
mapTree f (Node(l,r)) = Node (mapTree f l, mapTree f r)

result = mapTree (\x->x+1) t1
```

```haskell
result = mapOpt (\x->x+1) o1
```

Constructor Classes

• All map functions share the same structure

```haskell
map :: (a -> b) -> [a] -> [b]
mapTree :: (a -> b) -> Tree a -> Tree b
mapOpt :: (a -> b) -> Opt a -> Opt b
```

• They can all be written as:

```haskell
fmap :: (a -> b) -> g a -> g b
```

– where \( g \) is:

\([-]\) for lists, \textbf{Tree} for trees, and \textbf{Opt} for options

• Note that \( g \) is a function from types to types, i.e. a \textbf{type constructor}
Constructor Classes

• Capture this pattern in a constructor class,

```
class Functor g where
    fmap :: (a -> b) -> g a -> g b
```

A type class where the predicate is over type constructors
class **Functor** f where  
  fmap :: (a -> b) -> f a -> f b

instance Functor [] where  
  fmap f [] = []  
  fmap f (x:xs) = f x : fmap f xs

instance Functor Tree where  
  fmap f (Leaf x) = Leaf (f x)  
  fmap f (Node(t1,t2)) = Node(fmap f t1, fmap f t2)

instance Functor Opt where  
  fmap f (Some s) = Some (f s)  
  fmap f None = None
Constructor Classes

• Or by reusing the definitions map, mapTree, and mapOpt:

```haskell
class Functor f where
    fmap :: (a -> b) -> f a -> f b

instance Functor [] where
    fmap = map

instance Functor Tree where
    fmap = mapTree

instance Functor Opt where
    fmap = mapOpt
```
Constructor Classes

• We can then use the overloaded symbol `fmap` to map over all three kinds of data structures:

```haskell
*Main> fmap (\x->x+1) [1,2,3]
[2,3,4]
it :: [Integer]

*Main> fmap (\x->x+1) (Node(Leaf 1, Leaf 2))
Node (Leaf 2,Leaf 3)
it :: Tree Integer

*Main> fmap (\x->x+1) (Some 1)
Some 2
it :: Opt Integer
```

• The **Functor** constructor class is part of the standard Prelude for Haskell
Type classes /= OOP

• **Dictionaries** and **method suites** are similar
  – In OOP, a value carries a method suite.
  – With type classes, the dictionary travels separately
• Method resolution is static for type classes, dynamic for objects.
• Dictionary selection can depend on result type
  
  ```haskell
  fromInteger :: Num a => Integer -> a
  ```
• Based on polymorphism, not subtyping.
• Old types can be made instances of new type classes
  but objects can’t retroactively implement interfaces or inherit from super classes.