Polymorphism vs Overloading

- **Parametric polymorphism**
  - Single algorithm may be given many types
  - Type variable may be replaced by any type
  - \( f : t \rightarrow t \rightarrow \text{int} \rightarrow \text{int}, f : \text{bool} \rightarrow \text{bool}, \ldots \)

- **Overloading**
  - A single symbol may refer to more than one algorithm
  - Each algorithm may have different type
  - Choice of algorithm determined by type context
  - Types of symbol may be arbitrarily different
  - + has types \( \text{int} \times \text{int} \rightarrow \text{int}, \text{real} \times \text{real} \rightarrow \text{real}, \ldots \)

---

**TYPE CLASSES**

Kathleen Fisher

Reading "A history of Haskell: Being lazy with class"
Section 3 (skip 5), Section 6 (skip 6.4 and 6.7)
"How to Make All the Polymorphism Fit at Last"
Sections 1 – 7
"Real World Haskell", Chapter 6: Using Typeclasses

Thanks to Simon Peyton Jones for some of these slides.

---

**Why Overloading?**

Many useful functions are not parametric.

- Can member work for any type?
  
  \[
  \text{member} :: [w] \rightarrow w \rightarrow \text{Bool}
  \]

  No! Only for types \( w \) for that support equality.

- Can sort work for any type?
  
  \[
  \text{sort} :: [w] \rightarrow [w]
  \]

  No! Only for types \( w \) that support ordering.

---

**Overloading Arithmetic**

**First Approach**

- Allow functions containing overloaded symbols to define multiple functions:

  \[
  \text{square} \ x = x \times x \quad -- \text{legal} \\
  \quad \text{-- Defines two versions:} \\
  \quad \text{-- Int} \rightarrow \text{Int} \text{ and Float} \rightarrow \text{Float}
  \]

- But consider:

  \[
  \text{squares} \ (x,y,z) = \text{(square} \ x, \text{square} \ y, \text{square} \ z) \\
  \quad \text{-- There are 8 possible versions!}
  \]

- This approach has not been widely used because of exponential growth in number of versions.

**Second Approach**

- Basic operations such as + and * can be overloaded, but not functions defined in terms of them.

  \[
  3 \times 3 \quad -- \text{legal} \\
  3.14 \times 3.14 \quad -- \text{legal} \\
  \text{square} \ x = x \times x \quad -- \text{int} \rightarrow \text{int} \\
  \text{square} \ 3 \quad -- \text{legal} \\
  \text{square} \ 3.14 \quad -- \text{illegal}
  \]

- Standard ML uses this approach.
- Not satisfactory: Why should the language be able to define overloaded operations, but not the programmer?
Overloading Equality

First Approach
- Equality defined only for types that admit equality:
  types not containing function or abstract types.
- Overload equality like arithmetic ops + and * in SML.
- But then we can’t define functions using ‘==’:

```
3 * 3 == 9            -- legal
'a' == 'b'            -- legal

x->x == \y->y+1       -- illegal
```

- Approach adopted in first version of SML.

Overloading Equality

Second Approach
- Make equality fully polymorphic:
  
  ```
  (==) :: a -> a -> Bool
  ```
- Type of member function:
  
  ```
  member :: [a] -> a -> 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

Third Approach
- Make equality polymorphic in a limited way:
  
  ```
  (==) :: a -> a -> Bool
  ```
where `a` is a type variable ranging only over types that admit equality.
- Now we can type the member function:

```
member :: [a] -> a -> Bool
member [2,3] 4 :: Bool
member ['a', 'b', 'c'] 'c' :: Bool
member [(x->x, x->x+2) (y->y+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. They
  
  - Allow users to define functions using overloaded operations, e.g. square, squares, and member.
  - Generalize ML’s eqtypes to arbitrary types.
  - Provide concise types to describe overloaded functions, so no exponential blow-up.
  - Allow users to declare new collections of overloaded functions: equality and arithmetic operators are not privileged.
  - Fit within type inference framework.
  - Implemented as a source-to-source translation.

Intuition

- Sorting functions often take a comparison operator as an argument:

```
qsort :: (a -> a -> Bool) -> [a] -> [a]
qsort cmp [] = []
qsort cmp (x:xs) = qsort cmp (filter (cmp x) xs)
  ++ [x] ++
  qsort cmp (filter (not. cmp x) xs)
```

- which allows the function to be parametric.

- We can use the same idea with other overloaded operations.

```
qsort cmp [2,3] 4
```

Intuition, continued.

- Consider the "overloaded" function `parabola`:

```
parabola x = (x * x) + x
```

- We can rewrite the function to take the overloaded operators as arguments:

```
parabola' (plus, times) x = plus (times x x) x
```

- The extra parameter is a "dictionary" that provides implementations for the overloaded ops.

- We have to rewrite our call sites to pass appropriate implementations for plus and times:

```
y = parabola' (int_plus, int_times) 10
z = parabola' (float_plus, float_times) 3.14
```
Type Class Design Overview

- **Type class declarations**
  - Define a set of operations & give the set a name.
  - The operations are +, -, /, each with type a -> a -> a, Bool, form the Eq a type class.
- **Type class instance declarations**
  - Specify the implementations for a particular type.
  - For Int, == is defined to be integer equality.
- **Qualified types**
  - Concisely express the operations required on otherwise polymorphic type.

```
member :: Eq w => w -> [w] -> Bool
```

Qualified Types

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

```
sort :: Ord a => [a] -> [a]
serialise :: Show a => a -> String
squares :: (Num t, Num t1, Num t2) => (t, t1, t2) -> (t, t1, t2)
```

- Otherwise, it must work for any type whatsoever

```
reverse :: [a] -> [a]
filter :: (a -> Bool) -> [a] -> [a]
```

Compiled Overloaded Functions

When you write this...

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

...the compiler generates this

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

The "Num n :=" turns into an extra value argument to the function. It is a value of data type Num n. This extra argument is a dictionary providing implementations of the required operations.

A value of type (Num n) is a dictionary of the Num operations for type n
**Compound Translation**

- Build compound instances from simpler ones.

```haskell
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**

```haskell
data Eq = MkEq (Eq a -> a -> Bool) -- Dictionary type
dEqList :: Eq a -> Eq [a] -- List Dictionary
dEqList d = MkEq eql
           where eql []     []     = True
                eql (x:xs) (y:ys) = (==) d x y &&
                                   eql xs ys
                eql _      _      = False
```

**Subclasses**

- We could treat the Eq and Num type classes separately, listing each if we need operations from each.
  ```haskell
  memsq :: (Eq a, Num a) => [a] -> a -> Bool
  memsq xs x = member xs (square x)
  ```

- But we would expect any type providing the ops in Num to also provide the ops in Eq.

- **A subclass declaration** expresses this relationship:
  ```haskell
  class Eq a => Num a where
  (+) :: a -> a -> a
  (-) :: a -> a -> a
  fromInteger :: Integer -> a
  ...  
  inc :: Num a => a -> a
  inc x = x + 1
  ```

- **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
    (-) :: a -> a -> a
    fromInteger :: Integer -> a
    ...
    
    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
    ...
    ```

  - And then we can use values of type Cpx in any context requiring a Num:
    ```haskell
    data Cpx a = Cpx a a
    c1 = Cpx 1 0
    c2 = Cpx 2 0
    c3 = Cpx 3 0
    c4 = Cpx 1 1
    c5 = Cpx 1 1
    f = Cpx 1 1
    ```

  - **Recall:** Quickcheck is a Haskell library for randomly testing boolean properties of code.
    ```haskell
    Prelude Test.QuickCheck> quickCheck prop_BevRev
    +++ OK, passed 100 tests
    ```

- **Example: Complex Numbers**

  - Completely Different Example
    ```haskell
    Prelude Test.QuickCheck> quickCheck prop_BevRev
    ```
QuickCheck Uses Type Classes

```haskell
class Testable a where
test :: a -> RandSupply -> Bool
instance TestableBool where
test b r = b
class Arbitrary a where
  arby :: RandSupply -> a
instance (Arbitrary a, Testable b) where
test f r = test (f (arby r1)) r2
  where (r1, r2) = split r
```

```haskell
split :: RandSupply -> (RandSupply, RandSupply)
```

A completely different example: Quickcheck

```haskell
class Testable a where
test :: a -> RandSupply -> Bool
instance Testable Bool where
test b r = b
instance (Arbitrary a, Testable b) where
test f r = test (f (arby r1)) r2
  where (r1, r2) = split r
```

```haskell
test prop_RevRev r = test (prop_RevRev (arby r1)) r2
  where (r1, r2) = split r
```

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.

Default Methods

- Type classes can define "default methods."

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

```
```
For Read, Show, Bounded, Enum, Eq, and Ord type classes, the compiler can generate instance declarations automatically.

\[
data \text{Color} = \text{Red} | \text{Green} | \text{Blue}
\]
deriving (Read, Show, Eq, Ord)

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

Type inference infers a qualified type \( Q \Rightarrow T \)

- \( T \) is a Hindley Milner type, inferred as usual.
- \( Q \) is set of type class predicates, called a constraint.

Consider the example function:

\[
\text{example } \text{z xs} =
\begin{cases}
\text{case xs of} \\
[[]] & \rightarrow \text{False} \\
(y:ys) & \rightarrow y > z \land (y==z \land ys==[z])
\end{cases}
\]

\( T = a \rightarrow [a] \rightarrow \text{Bool} \)
- Constraint \( Q \) is \( \{\text{Ord } a, \text{Eq } a, \text{Eq } [a]\} \)
- \( Q \) simplifies to \( \{\text{Ord } a\} \)
- So, the resulting type is \( \{\text{Ord } a\} \Rightarrow a \rightarrow [a] \rightarrow \text{Bool} \)

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 (\x->x))
arising from a use of `print' at <interactive>:1:0-4
Possible fix: add an instance declaration for (Show (\x->x))
In the expression: print it
In a stmt of a 'do' expression: print it
```

There are many types in Haskell for which it makes sense to have a map function.

```
\text{mapList} :: (a \rightarrow b) \rightarrow [a] \rightarrow [b]
\text{mapList } f [1] = [f 1]
\text{result} = \text{mapList } (x \rightarrow x+1) [1,2,4]
```
There are many types in Haskell for which it makes sense to have a map function.

```
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(t1,t2)) = Node(mapTree f t1, mapTree f t2)
t1 = Node(Leaf 3, Leaf 4), Leaf 5)
result = mapTree (\x -> x+1) t1
```

```
Data Opt a = Some a | None
  deriving Show
mapOpt :: (a -> b) -> Opt a -> Opt b
mapOpt f None = None
mapOpt f (Some x) = Some (f x)
o1 = Some 10
result = mapOpt (\x -> x+1) o1
```

All of these map functions share the same structure.

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

They can all be written as:

```
map :: (a -> b) -> g a -> g b
```

where \( g \) is [-] for lists, Tree for trees, and Opt for options.

Note that \( g \) is a function from types to types. It is a type constructor.

We can make Lists, Trees, and Opt instances of this class:

```
class HasMap g where
  map :: (a -> b) -> g a -> g b
instance HasMap [] where
  map f [] = []
  map f (x:xs) = f x : map f xs
instance HasMap Tree where
  map f (Leaf x) = Leaf (f x)
  map f (Node(t1,t2)) = Node(map f t1, map f t2)
instance HasMap Opt where
  map f (Some x) = Some (f x)
  map f None = None
```

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

```
*Main> map (\x -> x+1) [1,2,3]
[2,3,4]
*Main> map (\x -> x+1) (Node(Leaf 1, Leaf 2))
Node (Leaf 2,Leaf 3)
*Main> map (\x -> x+1) (Some 1)
Some 2
```

The HasMap constructor class is part of the standard Prelude for Haskell, in which it is called "Functor."
Type classes = OOP?

- In OOP, a value carries a method suite.
- With type classes, the method suite travels separately from the value.
- Old types can be made instances of new type classes (e.g., introduce new Serialise class, make existing types an instance of it).
- Method suite can depend on result type (e.g., fromInteger :: Num a => Integer -> a).
- Polymorphism, not subtyping.
- Method is resolved statically with type classes, dynamically with objects.

Peyton Jones’ take on type classes over time

- Type classes are the most unusual feature of Haskell’s type system.

Type-class fertility

- Constructor Classes (1995)
- Implicit parameters (2000)
- Functional dependencies (2000)
- Generic programming
- Testing
- Applications
- Newtype deriving
- Multi-parameter type classes (1991)
- Extensible records (1996)
- Computation at the type level
- Associated types (2005)
- Overlapping instances
- "newtype deriving"
- Derivable type classes
- Variations

Type classes summary

- A much more far-reaching idea than the Haskell designers first realised: the automatic, type-driven generation of executable "evidence," i.e., dictionaries.
- Many interesting generalisations: still being explored heavily in research community.
- Variants have been adopted in Isabel, Clean, Mercury, Hal, Escher,...
- Who knows where they might appear in the future?