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

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 \( \text{Int} * \text{Int} \rightarrow \text{Int}, \text{Real} * \text{Real} \rightarrow \text{Real}, \) but no others

### Polymorphism vs Overloading

- Many useful functions are not parametric.
- Can member work for any type?
  ```haskell```
  ```member :: [w] -> w -> Bool```
  ```No! Only for types \( w \) for that support equality.```
- Can sort work for any type?
  ```sort :: [w] -> [w]```
  ```No! Only for types \( w \) that support ordering.```

- Many useful functions are not parametric.
- Can serialize work for any type?
  ```serialize :: w -> String```
  ```No! Only for types \( w \) that support serialization.```
- Can `sumOfSquares` work for any type?
  ```sumOfSquares :: [w] -> w```
  ```No! Only for types that support numeric operations.```

### Overloading Arithmetic

**First Approach**
- Allow functions containing overloaded symbols to define multiple functions:
  ```haskell```
  ```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!```
- 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 * 3 -- legal```
  ```3.14 * 3.14 -- 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 ‘==’:
  - Approach adopted in first version of SML.

Second Approach
- Make equality fully polymorphic.
- Type of member function:
  - 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.

Third Approach
- Only provides overloading for ‘==’.
- 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:
  - 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, eg. 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.

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.
Type Class Design Overview

- Type class declarations
  - Define a set of operations & give the set a name.
  - The operations `==` and `!=`, each with type `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
```

Works for any type `n` that supports the `Num` operations

```
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 = plusInt a b
    a * b = mulInt a b
    negate a = negInt a
    ... etc...
```

```
An instance declaration for a type `T` says how the Num operations are implemented on `T`.

sqrt :: (RealFloat r) => r -> r
sqrt x = ...
```

```
The class declaration says what the Num operations are.

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`.

```
Class instance declarations generate instances of the Dictionary data type.

```

Qualified Types

- For all types `w` that support the `Eq` operations

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

```
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]
```

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 = plusInt a b
    a * b = mulInt a b
    negate a = negInt a
    ... etc...
```

```
The compiler generates this
```
```
square :: Num n => n -> n
square x = x*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`.
Compiling Type Classes

When you write this...
\[ \text{square :: Num } n \Rightarrow n \times n \]
\[ \text{square } x = x \times x \]
...the compiler generates this
\[ \text{square :: Num } n \Rightarrow n \times n \]
\[ \text{square } d x = (*)(d x) \]

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

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

\[ \text{instance Num } \text{Int where} \]
\[ a + b = \text{plusInt } a \]
\[ a \times b = \text{mulInt } a \]
\[ \text{negate } a = \text{negInt } a \]
...etc...

Pass appropriate dictionary on to each square function.

Compiling Instance Declarations

When you write this...
\[ \text{square :: Num } n \Rightarrow n \times n \]
\[ \text{square } d x = (*)(d x) \]
...the compiler generates this
\[ \text{square :: Num } n \Rightarrow n \times n \]
\[ \text{square } d x = (*)(d x) \]

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

Functions with Multiple Dictionaries

\[ \text{squares :: (Num a, Num b, Num c) } \Rightarrow (a, b, c) \]
\[ \text{squares } (x, y, z) = (\text{square } x \times \text{square } y \times \text{square } z) \]

Note the concise type for the squares function!

Pass appropriate dictionary on to each square function.

Compositionality

- Overloaded functions can be defined from other overloaded functions:

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

\[ \text{sunEq} :: \text{Num } n \Rightarrow n \times n \times n \]
\[ \text{sunEq } x y z = \text{square } x + \text{square } y \]

Extract addition operation from d
Pass on d to square

Build compound instances from simpler ones:

\[ \text{class Eq a where} \]
\[ (==) :: a \times a \Rightarrow \text{Bool} \]
\[ \text{instance Eq } \text{Int where} \]
\[ (==) = \text{eqInt} \]

primitive equality

\[ \text{instance (Eq a, Eq b) } \Rightarrow (a, b) \Rightarrow (a, b) \]
\[ (u,v) == (x,y) = (u == x) \land (v == y) \]

\[ \text{instance Eq } [a] \text{ where} \]
\[ (==) [x], [y] = x == y \]
\[ (==) [x], [y] = \text{True} \]
\[ (==) [x], [y] = \text{False} \]
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

class Num a where
  (+) :: a -> a -> a
  (-) :: a -> a -> a
  fromInteger :: Integer -> a
  ....
  inc :: Num a => a -> a
  inc x = x + 1

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

...
QuickCheck Uses Type Classes

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

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

A completely different example: Quickcheck

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

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

A completely different example: Quickcheck

```haskell
class Arbitrary a where
arby :: RandSupply -> a
instance Arbitrary Int where
arby r = randInt r
instance Arbitrary a => Arbitrary [a] where
arby r | even r1 = []
  | otherwise = arby r2 : arby r3
  where (r1, r2) = split r
    (r2, r3) = split r'

split :: RandSupply -> (RandSupply, RandSupply)
randInt :: RandSupply -> Int
```

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.

QuickCheck: A Lightweight Tool for Unit Testing of Haskell Programs

- 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)
```

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

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

```haskell
data Color = Red | Green | 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

- 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:

```
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 constraint comes from y > z.
- Eq a comes from y == z.
- Eq [a] comes from ys == [z].

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
```

Detecting Errors

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

```
mapList :: (a -> b) -> [a] -> [b]
mapList f [] = []
mapList f (x:xs) = f x : mapList f xs
result = mapList (\x -> x+1) [1,2,4]
```

Constructor Classes

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

Result: mapList (\x -> x+1) [1,2,4] = [2,3,5]
There are many types in Haskell for which it makes sense to have a map function.

```haskell
Data Tree a = Leaf a | Node(Tree a, Tree a)
  deriving Show
mapTree :: (a -> b) -> Tree a -> Tree b
mapTree f (Node(l,r)) = Node (mapTree f l, mapTree f r)
t1 = Node(Node(Leaf 3, Leaf 4), Leaf 5)
result = mapTree (\x -> x+1) t1
```

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

```haskell
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.

They can all be written as:

```haskell
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:

```haskell
class HasMap g where
  map :: (a -> b) -> g a -> g b
instance HasMap [] where
  map :: (a -> b) -> [a] -> [b]
    map f [] = []
    map f (x:xs) = f x : map f xs
instance HasMap Tree where
  map :: (a -> b) -> Tree a -> Tree b
    map f (Leaf x) = Leaf (f x)
    map f (Node(t1,t2)) = Node (map f t1, map f t2)
instance HasMap Opt where
  map :: (a -> b) -> Opt a -> Opt b
    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:

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

- Wadler/Blott type classes (1989)
- Multi-parameter type classes (1991)
- Functional dependencies (2000)
- Associated types (2005)
- Overlapping instances
- Derivable type classes
- “newtype deriving”

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?