**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
  - Types of symbol may be arbitrarily different
  - + has types $\text{int}*\text{int} \to \text{int}$, $\text{real}*\text{real} \to \text{real}$, but no others

---

**Why 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?
  
  ```haskell
  sort :: [w] -> [w]
  ```

  No! Only for types w that support ordering.

---

**Why Overloading?**

Many useful functions are not parametric.

- Can sumOfSquares work for any type?
  
  ```haskell
  sumOfSquares :: [w] -> w
  ```

  No! Only for types that support numeric operations.
Overloading Arithmetic
First Approach
- Allow functions containing overloaded symbols to define multiple functions:
  \[
  \text{square } x = x * x \quad \text{-- legal} \\
  \text{-- Defines two versions:} \\
  \text{-- Int } \to \text{ Int and Float } \to \text{ Float}
  \]
- But consider:
  \[
  \text{squares (x,y,z) =} \\
  (\text{square } x, \text{ square } y, \text{ square } z) \\
  \text{-- There are 8 possible versions!}
  \]
- This approach has not been widely used because of exponential growth in number of versions.

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

\[
3 * 3 \quad \text{-- legal} \\
3.14 * 3.14 \quad \text{-- legal} \\
\text{square } x = x * x \quad \text{-- int } \to \text{ int} \\
\text{square } 3 \quad \text{-- legal} \\
\text{square } 3.14 \quad \text{-- illegal}
\]
- 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.

\[
3 * 3 == 9 \quad \text{-- legal} \\
\text{`a'} == \text{`b'} \quad \text{-- legal} \\
\text{\textbackslash x->x == \textbackslash y->y+1} \quad \text{-- illegal}
\]
- Overload equality like arithmetic ops + and * in SML.
- But then we can’t define functions using ‘==’:

\[
\text{member } [] \ y = \text{False} \\
\text{member } (x:xs) \ y = (x==y) || \text{member } xs \ y \\
\text{member } [1,2,3] \ 3 \quad \text{-- legal if int is the default} \\
\text{member } \text{“Haskell” } \text{‘k’} \quad \text{-- illegal}
\]
- Approach adopted in first version of SML.

Overloading Equality
Second Approach
- Make equality fully polymorphic.

\[
(==) :: a \to a \to \text{Bool}
\]
- Type of member 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

Third Approach

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

  where \(a_{(==)}\) is a type variable ranging only over types that admit equality.

- Now we can type the member function:

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

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

- Approach used in SML today, where the type \(a_{(==)}\) is called an “eqtype variable” and is written \(\text{a}_{(==)}\).

Type Classes

- Type classes solve these problems. They
  - Allow users to define functions using overloaded operations, eg, \texttt{square}, \texttt{squares}, and \texttt{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:

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

  which allows the function to be parametric.

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

Intuition, continued.

- Consider the “overloaded” function \texttt{parabola}:

  \[
  \text{poly x} = (x * x) + x
  \]

  \[
  \text{poly1 (plus, times)} x = \text{plus} (\text{times} x x) x
  \]

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

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

  \[
  y = \text{poly1(int_plus,int_times)} 10
  z = \text{poly1(float_plus, float_times)} 3.14
  \]
-- 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”
poly2 :: MathDict a -> a -> a
poly2 dict x = let plus  = get_plus  dict
                times = get_times dict
                in plus (times x x) x

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

-- Passing dictionaries
y = poly2 intDict   10
z = poly2 floatDict 3.14

Type class
declarations
will generate Dictionary
type and accessor
functions.

If a function has a qualified type, the 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.
  - 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] -> [w]
```

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

  The class declaration says what the Num operations are.

- `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...

- FORGET all you know about OO classes!

**Compiling Overloaded Functions**

When you write 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.

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

  A value of type `(Num n)` is a dictionary of the Num operations for type `n`.

**Compiling Type Classes**

When you write this...

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

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

- `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...

  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`

  A value of type `(Num n)` is a dictionary of the Num operations for type `n`.

- `instance Num Int where
  a + b = plusInt a b
  a * b = mulInt a b
  negate a = negInt a
    ...etc...

  dNumInt :: Num Int
  dNumInt = MkNum plusInt
            mulInt
            negInt
            ...

  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 accessor 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

- The concise type for the `squares` function!

Compositionality

- Overloaded functions can be defined from other overloaded functions:

```haskell
sumSq :: Num n => n -> n -> n
sumSq x y = square x + square y
```

```haskell
sumSq :: Num n -> n -> n -> n
sumSq d x y = (+) d (square d x) (square d y)
```

- Build compound instances from simpler ones:

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

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

instance (Eq a, Eq b) => Eq(a,b) where
  (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.

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

```haskell
data Eq = MkEq (a->a->Bool)    -- Dictionary type
        (==) (MkEq eq) = eq            -- Selector
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
  ```

- With that declaration, we can simplify the type:
  ```haskell
  memsq :: Num a => [a] -> a -> Bool
  memsq xs x = member xs (square x)
  ```

**Numeric Literals**

- Even literals are overloaded. `1` means “fromInteger 1”

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

Haskell defines numeric literals in this indirect way so that they can be interpreted as values of any appropriate numeric type. Hence 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
  ....

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 Cpx in any context requiring a Num:

```haskell
data Cpx a = Cpx a a

c1 = 1 :: Cpx Int

-- Parabola function
parabola x = (x * x) + x

-- Complex numbers
parabola c3

-- Complex numbers
i1 = parabola 3
```

Recall:

- Quickcheck is a Haskell library for randomly testing boolean properties of code.

```haskell
-- Reverse function
reverse [] = []

reverse (x:xs) = (reverse xs) ++ [x]

-- Write properties in Haskell

-- Parabola function

-- Quickcheck

Prelude Test.QuickCheck> quickCheck prop_RevRev
+++ OK, passed 100 tests

Prelude Test.QuickCheck> :t quickCheck
quickCheck :: Testable a => a -> IO ()
```

Quickcheck Uses Type Classes

```haskell
-- Quickcheck uses type classes

quickCheck :: Testable a => a -> IO ()

-- Class and instance definitions

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
-- Different example

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

-- Testing

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

= prop_RevRev (arby r1)
```

Using instance for (->)

Using instance for Bool
A completely different example: Quickcheck

```
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, r') = split r
    (r2, r3) = split r'

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

QuickCheck uses type classes to auto-generate
random values
testing functions

based on the type of the function under test

Nothing is built into Haskell;
QuickCheck is just a library!

Plenty of wrinkles, especially
- test data should satisfy preconditions
- generating test data in sparse domains

QuickCheck: A Lightweight tool for random testing of Haskell Programs

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.”

```
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 \( Q \Rightarrow T \)
  - \( T \) is a Hindley Milner type, inferred as usual.
  - \( Q \) is set of type class predicates, called a constraint.
- Consider the example function:

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

- \( T = a \rightarrow [a] \rightarrow \text{Bool} \)
- Constraint \( Q \) is \( \{\text{Ord } a, \text{Eq } a, \text{Eq } [a]\} \)

- 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:

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

- \( T = a \rightarrow [a] \rightarrow \text{Bool} \)
- Constraint \( Q \) is \( \{\text{Ord } a, \text{Eq } a, \text{Eq } [a]\} \)

- 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:

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

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

Constructor Classes

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

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

```
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)
t1 = Node(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.

They can all be written as:

\[ \text{map} : (a \rightarrow b) \rightarrow (f a \rightarrow f b) \]

where \( f \) is \([-]\) for lists, \( \text{Tree} \) for trees, and \( \text{Opt} \) for options.

Note that \( f \) is a function from types to types. It is a \textit{type constructor}.

We can capture this pattern in a \textit{constructor class}, which is a type class where the predicate ranges over type constructors:

\[
\text{class HasMap f where}
\begin{align*}
\text{map} & : (a \rightarrow b) \rightarrow (f a \rightarrow f b) \\
\end{align*}
\]

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

```haskell
class HasMap f where
    map :: (a -> b) -> (f a -> f 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 s) = Some (f s)
    map f None = None
```

We can then use the overloaded symbol \texttt{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+1) (Node(Leaf 1, Leaf 2))
Node (Leaf 2,Leaf 3)
*Main> map (\x->x+1) (Some 1)
Some 2
```

The \texttt{HasMap} constructor class is part of the standard Prelude for Haskell, in which it is called \textit{"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.

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
- Variants adopted in Isabel, Clean, Mercury, Hal, Escher, ...