301AA - Advanced Programming

Lecturer: Andrea Corradini
andrea@di.unipi.it
http://pages.di.unipi.it/corradini/

Course pages:
http://pages.di.unipi.it/corradini/Didattica/AP-18/

AP-2018-20: Type Classes in Haskell
Core Haskell

• Basic Types
  – Unit
  – Booleans
  – Integers
  – Strings
  – Reals
  – Tuples
  – Lists
  – Records

• Patterns
• Declarations
• Functions
• Polymorphism
• Type declarations
  • Type Classes
  • Monads
  • Exceptions
Polymorphism in Haskell

Polymorphism

Universal

Parametric

Coercion

Implicit

Explicit

Bounded

Covariant

Invariant

Contravariant

Overroding

Overloading

Type classes

Type Inference
Ad hoc polymorphism: overloading

- Present in all languages, at least for built-in arithmetic operators: +, *, -, ...
- Sometimes supported for user defined functions (Java, C++, ...)
- C++, Haskell allow overloading of primitive operators
- The code to execute is determined by the type of the arguments, thus
  - **early binding** in statically typed languages
  - **late binding** in dynamically typed languages
Overloading: an example

- Function for squaring a number:
  ```c
  int sqr(int x) { return x * x; }
  ```
- Typed version (like in C):
  ```c
  int sqr(int x) { return x * x; }
  ```
- Multiple versions for different types:
  ```c
  int sqrInt(int x) { return x * x; }
  double sqrDouble(double x) { return x * x; }
  ```
- Overloading (Java, C++):
  ```c
  int sqr(int x) { return x * x; }
  double sqr(double x) { return x * x; }
  ```
- But which type can be inferred by ML/Haskell?
  ```c
  > sqr x = x * x
  ```
Overloading besides arithmetic

• Some functions are "fully polymorphic"

\[
\text{length} :: [w] \rightarrow \text{Int}
\]

• Many useful functions are less polymorphic

\[
\text{member} :: [w] \rightarrow w \rightarrow \text{Bool}
\]

• Can list membership work for any type?
  – No! Only for types \( w \) that support equality.

\[
\text{sort} :: [w] \rightarrow [w]
\]

• Can list sorting work for any type?
  – No! Only for types \( w \) that support ordering.
Overloading Arithmetic, Take 1

• 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:

```haskell
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

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

• Type of list membership function

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

• **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(==) -> [a(==)] -> Bool
member 4         [2,3] :: Bool
member 'c'       ['a', 'b', 'c'] :: Bool
member (\y -> y*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 "\(a\) (while normal type variables are 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:

```haskell
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)
```

– This allows the function to be parametric

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

• Consider the “overloaded” parabola function

parabola \( x = (x \times x) + x \)

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

parabola’ (plus, times) \( x = \) plus (times \( x \times x \)) \( x \)

  – 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:

\[
\text{y} = \text{parabola’}(\text{intPlus}, \text{intTimes}) \ 10 \\
\text{z} = \text{parabola’}(\text{floatPlus}, \text{floatTimes}) \ 3.14
\]
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
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: `Eq a` type class
    • operations `==` and `\=` with `type a -> a -> Bool`

• **Type class instance declarations**
  – Specify the implementations for a particular type
  – For `Int` instance, `==` is defined to be integer equality

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

  \[
  \text{member} :: \text{Eq } w \Rightarrow w \rightarrow [w] \rightarrow \text{Bool}
  \]
If a function works for every type with particular properties, the type of the function says just that:

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

Otherwise, it must work for any type:

- `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 = intPlus a b
    a * b = intTimes a b
    negate a = intNeg a
    ...etc...

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

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

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

When you write this...

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

...the compiler generates this

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

The “Num n =>” turns into an extra value argument to the function. It is a value of data type Num n and it represents a dictionary 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...

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

...the compiler generates this

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

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

```
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
```

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

When you write this...

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

...the compiler generates this

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

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

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

An instance declaration 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)

Note the concise type for the squares function!

Pass appropriate dictionary on to each square 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:

\[
\text{sumSq} :: \text{Num} \ n \Rightarrow n \rightarrow n \rightarrow n
\]

\[
\text{sumSq} \ x \ y = \text{square} \ x + \text{square} \ y
\]
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

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

```hs
Main>:t show
show :: Show a => a -> String
Main> show Red
"Red"
Main> Red < Green
True
```

```hs
Main>:t read
read :: Read a => String -> a
Main> let c :: Color = read "Red"
Main> c
Red
```

• **Ad hoc**: derivations apply only to types where derivation code works
class Num a where
  (+) :: a -> a -> a
  (-) :: a -> a -> a

fromInteger :: Integer -> a
...

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

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

“1” means “fromInteger 1”
Type Inference with overloading

- In presence of overloading (Type Classes), 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:

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

  - Type $T$ is $a \rightarrow [a] \rightarrow \text{Bool}$
  - Constraint $Q$ is $\{ \text{Ord } a, \text{Eq } a, \text{Eq } [a] \}$

  Ord $a$ because $y > z$
  Eq $a$ because $y == z$
  Eq $[a]$ because $ys == [z]$
Simplifying Type Constraints

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

• Applying these rules,
  – (Ord a, Eq a, Eq[a]) simplifies to  Ord a
Type Inference with overloading

• Putting it all together:

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

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

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

Prelude> 'a' + 1
<interactive>:33:1: error:
  • No instance for (Num Char) arising from a use of ‘+’
  • In the expression: 1 + 'a'
  In an equation for ‘it’: it = 1 + 'a'

Prelude> (\x -> x)
<interactive>:34:1: error:
  • No instance for (Show (p0 -> p0)) arising from a use of ‘print’
    (maybe you haven't applied a function to enough arguments?)
  • In a stmt of an interactive GHCi command: print it