301AA - Advanced Programming [AP-2017]

Lecturer: Andrea Corradini andrea@di.unipi.it
Tutor: Lillo Galletta galletta@di.unipi.it

Department of Computer Science, Pisa
Academic Year 2017/18

**AP-2017-20**: 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
- Inclusion
- Coercion
- Explicit
- Bounded
- Covariant
- Invariant
- Contravariant

- Overloading
- Type classes

Type Inference
- Implicit

Ad hoc
Why Overloading?

• Many useful functions are not parametric
• Can list membership work for any type?
  – No! Only for types w that support equality.
• Can list sorting work for any type?
  – No! Only for types w that support ordering.

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

\[
\text{sort :: } [w] \to [w]
\]
Overloading Arithmetic, Take 1

• Allow functions containing overloaded symbols to define multiple functions:

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

• But consider:

```plaintext
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.

```plaintext
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 ‘==‘:

```plaintext
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**:

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

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

• Now we can type the member function:

```haskell
member :: a(==) \to [a(==)] \to \text{Bool}
member 4 [2,3] :: \text{Bool}
member 'c' ['a', 'b', 'c'] :: \text{Bool}
member (\y\to y \ast 2) [\x\to x, \x\to 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:

```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

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

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

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

– 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:

\[
y = \text{parabola’ (intPlus, intTimes) } 10 \\
z = \text{parabola’ (floatPlus, floatTimes) } 3.14
\]
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: `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

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

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

Works for any type ‘n’ that supports the Num operations

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

The class declaration says what the Num operations are

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

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

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

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

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

The “Num \; n \Rightarrow” 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

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 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.

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)

Note the concise type for the squares function!
Compositionality

Overloaded functions can be defined from other overloaded functions:

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

\[
\text{sumSq} x y = \text{square } x + \text{square } y
\]
Build compound instances from simpler ones:

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

instance Eq Int where
    (==) = intEq  -- intEq 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
```
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

\[
\text{memsq :: (Eq a, Num a) => a -> [a] -> Bool}
\]
\[
\text{memsq x xs = member (square x) xs}
\]

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

• A subclass declaration expresses this relationship:

\[
\text{class Eq a => Num a where}
\]
\[
(+) :: a -> a -> a
\]
\[
(*) :: a -> a -> a
\]

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

\[
\text{memsq :: Num a => a -> [a] -> Bool}
\]
\[
\text{memsq x xs = member (square x) xs}
\]
Default Methods

• Type classes can define “default methods”

```
-- 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

```haskell
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.
Type Inference with overloading

- In presence of overloading (Type Classes), 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 $a \rightarrow [a] \rightarrow \text{Bool}$
  - Constraint $Q$ is $\{ \text{Ord } a, \text{Eq } a, \text{Eq } [a] \}$

  - $\text{Ord } a$ because $y > z$
  - $\text{Eq } a$ because $y == z$
  - $\text{Eq } [a]$ 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])
```

– $T = a \rightarrow [a] \rightarrow \text{Bool}$
– $Q = (\text{Ord } a, \text{Eq } a, \text{Eq } [a])$
– $Q$ simplifies to $\text{Ord } a$
– example :: $\text{Ord } a \Rightarrow a \rightarrow [a] \rightarrow \text{Bool}$
Detecting Errors

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

```haskell
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
```
Towards Type Constructor Classes and Monads:
The **Maybe** type constructor
The **Maybe** type constructor

- **Type constructor**: a generic type with one or more type variables

```haskell
data Maybe a = Nothing | Just a
```

- A value of type **Maybe a** is a possibly undefined value of type **a**

- A function `f :: a -> Maybe b` is a partial function from **a** to **b**

```haskell
max [] = Nothing
max (x:xs) = Just (foldr (\y z -> if y > z then y else z) x xs)
max :: Ord a => [a] -> Maybe a
```
Composing partial function

father :: Person -> Maybe Person  -- partial function
mother :: Person -> Maybe Person  -- (lookup in a DB)

maternalGrandfather :: Person -> Maybe Person
maternalGrandfather p =
    case mother p of
        Nothing -> Nothing
        Just mom -> father mom  -- Nothing or a Person

bothGrandfathers :: Person -> Maybe (Person, Person)
bothGrandfathers p =
    case father p of
        Nothing -> Nothing
        Just dad ->
            case father dad of
                Nothing -> Nothing
                Just gf1 ->
                    case mother p of
                        Nothing -> Nothing
                        Just mom ->
                            case father mom of
                                Nothing -> Nothing
                                Just gf2 ->
                                    Just (gf1, gf2)
                                      -- found second grandfather
The **Monad** type class and the **Maybe** monad

```haskell
class Monad m where
  return :: a -> m a
  (>>=)  :: m a -> (a -> m b) -> m b -- "bind"
  ...

instance Monad Maybe where
  return :: a -> Maybe a
  return x = Just x
  (>>=)  :: Maybe a -> (a -> Maybe b) -> Maybe b
  y >>= g = case y of
    Nothing  -> Nothing
    Just x  -> g x
```

- `m` is a type constructor
- `m a` is the type of monadic values

- **`bind (>>=)`** shows how to “propagate” undefinedness
Composing partial functions

• We introduce a higher order operator to compose partial functions in order to “propagate” undefinedness automatically.

\[
y >>= g = \begin{cases} 
\text{Nothing} & \rightarrow \text{Nothing} \\
\text{Just } x & \rightarrow g \ x 
\end{cases}
\]

\textbf{y \gg=} g = \begin{case} 
\text{Nothing} & \rightarrow \text{Nothing} \\
\text{Just } x & \rightarrow g \ x 
\end{case}

\( (\gg=) \quad :: \text{Maybe } a \rightarrow (a \rightarrow \text{Maybe } b) \rightarrow \text{Maybe } b \)

• The \textit{bind} operator will be part of the definition of a \textit{monad}. 
Use of **bind** to compose partial functions

father :: Person -> Maybe Person  -- partial function
mother :: Person -> Maybe Person  -- (lookup in a DB)

maternalGrandfather :: Person -> Maybe Person
maternalGrandfather p =
case mother p of
  Nothing -> Nothing
  Just mom -> father mom

maternalGrandfather p = mother p >>= father

bothGrandfathers p =
  father p >>=
    (\dad -> father dad >>=
      (\gf1 -> mother p >>=
        (\mom -> father mom >>=
          (\gf2 -> return (gf1,gf2) )))))