Types, Data Types, and Polymorphism

Winter 2015-2016

Table of Contents

Review: Basic Haskell Types

- booleans: **Bool** ; values: **True** , **False**
- integers: **Int** , **Integer**
- floating point numbers: **Float** , **Double**
- characters: **Char** : values: eg, 'a' , 'b'
- strings: **String** : eg, "abc" , ""
- Haskell type names and values are Capitalized
- Every value, expression has a type

Function types

- A function maps input type(s) to an output type
- The type operator `->` is pronounced "to"
- A function's type has at least one `->`

```
square :: Integer -> Integer
square a = a * a

maxOf2 :: Int -> Int -> Int
maxOf2 x y | x > y     = x
           | otherwise = y
```

Type Synonyms

- A simple notation for giving a new name to a type
- Example follows (not using synonyms). . .
- Similar to a previous example

```haskell
-- compute both roots of quadratic equation
-- a * x * x + b * x + c = 0

quad :: (Float, Float, Float) -> (Float, Float)

quad (a,b,c)
  | a == 0      = error "not quadratic"
  | disc < 0    = error "not real"
  | otherwise   = (r1, r2)
   where disc = b*b - 4*a*c
         r1   = (-b + (sqrt disc))/(2 * a)
         r2   = (-b - (sqrt disc))/(2 * a)
```

- `quad`'s argument, the triple `(a,b,c)`, was declared to have type **(Float, Float, Float)**
- A type synonym is a way of giving another name for a type E.g.,

```
type Coefs = (Float, Float, Float)
type Roots = (Float, Float)
```

```
type Coefs = (Float, Float, Float)
type Roots = (Float, Float)

-- compute both roots of quadratic equation
-- a * x * x + b * x + c = 0

quad :: Coefs -> Roots
quad (a,b,c)
  | a == 0      = error "not quadratic"
  | disc < 0    = error "not real"
  | otherwise   = (r1, r2)
   where disc = b*b - 4*a*c
         r1   = (-b + (sqrt disc))/(2 * a)
         r2   = (-b - (sqrt disc))/(2 * a)
```

Creating type synonyms

- Syntax: **type** <Name> = <signature>
- The Name must begin with a capital letter
  (all type names begin with a capital letter: Float, Integer, Char...)
- Semantics: The name is associated with the signature
- This is a good way to make signatures more meaningful and concise!
- However, nothing new is being created, except a name
- Type synonyms inherit properties of the underlying types

Data types

- Haskell's **type** declaration allows programmers to define new names for existing types.
- Nothing new is created.
- Haskell provides a tool to create data types that are completely new
- The keyword **data** is used for this.

Example: Booleans

- The type **Bool** is defined in Haskell's Prelude:

```
data Bool = False | True
```

- The keyword **data** tells Haskell a new data type is being defined
- The **Bool** is the name for the new datatype
- There are exactly 2 values in this type: **True** and **False**
- The $=$ separates the name from the values
- The $|$ separates different values of the type

Creating Datatypes

- The type **Bool** is defined in Haskell's Prelude:

```
data Bool = False | True
```

- Unlike **type**, a **data** definition creates something new
- All data values, e.g., **True**, **False**, must be capitalized! (all data types and type values must be capitalized)
- The values **True**, **False** are also called constructors
They "construct" the values of **Bool**.

Defining functions on Datatypes

- Haskell allows programmers to define functions on datatypes using patterns and constructors.
- Other languages require new types to be constructed out of old types

Example: Boolean negation

- Haskell defines **not**, the code looks like this:

```
not :: Bool -> Bool

not False = True
not True  = False
```

- There is a "pattern" for each constructor.

Evaluating function calls with patterns

- Evaluating **not** <expression>:
  1. <expression> is evaluated
  2. the result is compared to **False** – the first pattern
  3. If it matches, the right side of the first pattern is evaluated and returned
  4. If it doesn't match, the next pattern is tried...
  5. If no pattern match, a runtime error is generated
- There were no builtin functions used to define **not**

Example: Days of a week

data Day = Monday | Tuesday | Wednesday | Thursday
| Friday | Saturday | Sunday

- There are no built-in operations defined on Day
- How to define equality? $\leq$? Any other function?
- Solution: Pattern matching

Example: Days of our lives

```
cheer :: Day -> String

cheer Monday    = "I don't like Mondays"
cheer Tuesday   = "Goodbye, Ruby!"
cheer Wednesday = "Wednesday morning at five o'clock."
cheer Thursday  = "I could never get the hang of Thursdays."
cheer Friday    = "It's Friday, Friday!"
cheer Saturday  = "Saturday night's alright alright alright"
cheer Sunday    = error "Couldn't find anything interesting."
```

Evaluating function calls with patterns

- Evaluating `cheer <expression>`:
  1. `<expression>` is evaluated
  2. the result is compared to `Monday` – the first pattern
  3. If it matches, the right side of the first pattern is returned
  4. If it doesn't match, the next pattern is tried...
  5. Each pattern is tried from top down; the first one to match is used
  6. If no pattern match, a runtime error is generated

Benefits of pattern matching

- With pattern matching, the programmer is free to define types independent of other existing types
- Powerful ability not found in C, C++, Java
  (In C/Java, you always have to base your new classes on existing classes)
- Function definitions can be short and clear

Another example

- Haskell defines boolean operator && (and) and || (or)
- But let's look at one way of doing it ourselves:

```
and :: Bool -> Bool -> Bool

and False x = False
and True x = x
```

Example: how does evaluation work?

Assume the expression to be evaluated is **and** <e1> <e2>

1. Use first pattern, evaluate <e1> and compare the result to **False**
2. If the result matches **False** return the RHS (i.e., **False**)
3. If <e1> evaluates to **True**, the first eqution fails to match, so the second pattern must be used, and in this case, <e2> is returned

Anonymous variables

- If you are defining a function using patterns, you can use _ as an anonymous variable. It is useful when the argument is only used on the LHS of an equation, e.g.

```
and :: Bool -> Bool -> Bool

and False _ = False
and       _ x = x
```

- The benefit is that you don't need to clutter your definition with names you use only once in an equation.

Example: Representing Shapes I

```
data Shape = Circle Float                -- radius
           | Square Float                -- side length
           | Rectangle Float Float       -- width, height
```

- The constructors are `Circle`, `Square` and `Rectangle`
- When you want to represent a circle whose radius is 1.5, you write `Circle 1.5`
- To represent a rectangle of width 1 and height 2: `Rectangle 1 2` (no commas!)
- Constructors are functions:
  - `Circle :: Float -> Shape`
  - `Square :: Float -> Shape`
  - `Rectangle :: Float -> Float -> Shape`

Computing the area of a Shape

```
data Shape = Circle Float |
             Square Float |
             Rectangle Float Float

area :: Shape -> Float
area (Circle r) = pi * r * r
area (Square l) = l * l
area (Rectangle w h) = w * h
```

Simple: a pattern for each constructor.

Generally: have a pattern for each constructor.

The dual nature of Constructors

- When a constructor is used on the LHS, Haskell does pattern matching against the constructor syntax.
- In other words, a constructor is treated as a data structure.
- However, the constructor acts as a function when it is used on the RHS.
- In other words, it has arguments and returns a value when called (the value is itself).

Example: changing shapes

```
convert   :: Shape -> Shape

convert ( Circle  x) = Square x      -- change circle to square
convert (Square x) = Circle  x       -- change square to circle
convert s = s                        -- nothing else changes
```

- On the LHS of the first equation, Circle x is used in pattern matching.
- On the RHS of the second equation, Circle x is used to construct a value of type Shape.

Recursive data types

Peano numbers

```
data Peano = Zero | S Peano
```

- We represent 0 by `Zero`
- We represent 1 by `S Zero`
- We represent 2 by `S (S Zero)`

Constructor `S` really is a function!

Polymorphism in general

- Polymorphism is the property that a definition can be applied to many different types.
- Benefits: code re-use. Write one polymorphic function, and use it in many different situations.

Polymorphism in Haskell

- In Haskell, polymorphism is achieved by using type variables
- A type variable is just a lower case symbol, often just a single letter
- A type variable is used to represent "many types"
- A type variable may appear in several places:
    - Type signatures
    - Type synonyms
    - Datatype definitions
    - Type class definitions
- A type variable never appears in a function definition

Example: not polymorphic type synonyms

- Suppose we wanted a pair of integers: we could define:

```
type PairOfInt = (Int, Int)
```

- If we wanted a pair of single precision floating point values:

```
type PairOfFloat = (Float, Float)
```

- The only difference is the type being contained, i.e., **Float** vs. **Int**

Polymorphic type synonyms

- Type synonyms can be polymorphic (parameterized with type variables)
- Example:

```
type PairOf a = (a, a)

p1 :: PairOf Int
p1 = (0,0)

p2 :: PairOf Char
p2 = ('a','b')

p3 :: PairOf Peano
p3 = (Zero, S Zero)
```

- The type PairOf a is polymorphic
- Two of the same kind of thing
- We can have pairs of any type!

Polymorphic data types

- Data types can also be polymorphic
- We parameterize the datatype name with a type variable.
- E.g., shapes:

data Shape a = Circle a | Square a | Rectangle a a

- This says
  - Shape a is a polymorphic datatype
  - It has 3 polymorphic type constructors;
  - The type variable a stands in place of any type.
  - The constructors Circle and Square both take one argument of type a (whatever you decide to use),
  - The constructor Rectangle takes two arguments, both of type a .

Example: Making polymorphic shapes

```
data Shape a = Circle a | Square a | Rectangle a a
```

- This allows us to specialize shapes for different purposes.
  - Shape **Int**
  - Shape **Float**
  - Shape **Double**
  - Shape Peano
- However, we can also specialize less useful types:
  - Shape **Bool**
  - Shape **Char**
- A data declaration cannot use a type restriction, so our functions have to!

Example: Calculating the area of a shape

```
data Shape a = Circle a | Square a   | Rectangle a a

area :: Floating a => Shape a -> a
area ( Circle  r) = pi * r * r
area (Square l) = l * l
area (Rectangle w h) = w * h
```

- A data declaration cannot use a type restriction, so our functions have to!
- Haskell cannot calculate `area ( Circle 'a')` even though `Circle 'a'` is a perfectly valid instance of a shape.

Lists: Examples

- In Haskell, a list is a polymorphic datatype.
- Lists are singly-linked.
- Examples:

```
list1 :: [Int]
list1 = [1, 2, 3]

list2 :: [Char]
list2 = ['a', 'b', 'c', 'd']

list3 :: [String]
list3 = ["doh", "re", "me"]
```

Lists: Syntax

- The empty list is `[]`.
- The square brackets are used in 2 ways
  1. In type signatures
  2. For literal lists

```
list1 :: [Int]
list1 = [1, 2, 3]
```

- NOTE: The notation for specifying list parameters to functions is different!

The ugly truth about lists

- The list is a recursive and polymorphic datatype.
- Lists are built-in, but they could be defined as follows:

```
data [a] = [] | a : [a]
```

- This says: A List of elements of type `a` is either
  - the empty list, `[]`
  - or a construction of a value of type `a` with a list of type `[a]`
- Example: `1 : 2 : 3 : []`
- `:` is an infix operator, associates right
- Equivalent lists:
  `[1,2,3] = 1:[2,3] = 1:2:[3] = 1:2:3:[]`

List Operations: Append

- Note the use of `(x:xs)` to specify the head/tail of a Haskell list:

```
append :: [a] -> [a] -> [a]

append [] ls = ls
append (x:xs) ls = x : (append xs ls)
```

- The notation `[a]` in the type signature means “a list of elements all of type `a`”
- Polymorphic: one definition that can be applied to lists of any type.
- There is a built-in version of this function: `(++)`.

List Operations: Length

- Example: length

```
length :: [a] -> Int

length [] = 0
length (x:xs) = 1 + (length xs)
```

- Polymorphic: it doesn't matter what kind of list it is.
- Note that the type variable a is lower case!
- The parentheses in (x:xs) show precedence. The ( ) are not part of the list!

Common list syntax errors

Using [ ] badly:

```
mylength :: [a] -> Int

mylength [] = 0
mylength [x:xs] = 1 + (mylength xs)
```

```
mylength.hs:2:23:
    Occurs check: cannot construct the infinite type: t = [t]
      Expected type: [t] -> t1
       Inferred type: [[t]] -> t2
    In the second argument of '(+)', namely '(mylength xs)'
    In the expression: 1 + (mylength xs)
```

Common list syntax errors

Forgetting to indicate precedence:

```
mylength :: [a] -> Int

mylength [] = 0
mylength x:xs = 1 + (mylength xs)
```

```
mylength.hs:2:0: Parse error in pattern
```

List Operations: reverse

- Example: reverse: return the list in the reverse order

```
reverse :: [a] -> [a]

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

- Again, one implementation, many uses!
- It can reverse any kind of list.
- It's inefficient to reverse a list this way! Use tail recursion instead.

When polymorphism isn't enough

- What type should `+` be?
    1. Too restrictive: `(+) :: Integer -> Integer -> Integer`
    2. Too loose: `(+) :: a -> a -> a`
- We don't want `+` to be polymorphic, because
    - the implementation would change depending on the type of the arguments,
    - e.g., adding complex numbers is different from adding integers
    - but they are related enough to use the same syntax, `+`

Another example: Equality

- We have seen `==` used with numbers
- How can we use `==` with other types?
  - What type would it have?
  - What should `==` compute?
- A polymorphic definition won't work
- Each type may need a different implementation associated with `==`

Overloading

- An operator is overloaded if
  1. A single syntax can be applied to many types
  2. For each type, the implementation is different
- Example: In Java, the `+` can be used to
  - Add numbers: `3 + 4`
  - Append strings: `"this"+"that"`
  - The same operator, different computation
  - The same syntax, different semantics
- Excellent Exam question: What's the difference between an overloaded operator and a polymorphic operator?

Type classes implement overloading

- In Haskell, operator overloading is implemented with type classes
- Every type class defines one or more operators, which are overloaded for each member of the type class, eg:
  - `==` defined for each type that is in the class **Eq**
  - `<` defined for each type that is in the class **Ord**
  - **show** defined for each type that is in the class **Show**

More Existing Type Classes

- There are several type classes pre-defined in Haskell
  - class **Show**: values that can be displayed
  - class **Eq**: values that can be tested for equality $==$
  - class **Enum**: values that can be counted
  - class **Num**: values that can be used in numeric expressions
  - class **Fractional**: subclass of **Num** for operations that need floating point (**Float** or **Double**)
  - class **Ord**: values that can be ordered using $<$
  - and lots of others
  - Know these basic ones; be aware that others exist (especially when interpreting error messages)

Defining Type Classes I

- A type class is a set of types, all of which overload a number of functions for the class.
- The class associated with equality `==` is **Eq**
- Defined in the Haskell Prelude, as follows

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

- **class** tells Haskell to define a new class
- **Eq** a is the polymorphic class name (must be polymorphic!)
- Declares 2 operators with polymorphic signatures
  1. Equality (`==`)
  2. Inequality (`/=`)
- No definitions for the operators are given yet. Each instance defines its own implementation!.

Overview of the type Class **Ord**

- The class of types whose elements can be ordered
- Eg: numbers, characters, etc.
- The **Ord** class is associated with the operators
  $<$: less than
  $>$: greater than
  $<=$: less or equal
  $>=$: greater or equal
- Note:
  - The $<=$ implies an equality test
  - So an instance of **Ord** must also be an instance of **Eq**

Definition of the type Class **Ord**

```
class (Eq a) => (Ord a) where
  (<),(<=),(>),(>=) :: a -> a -> Bool
```

- **Ord** a is a subclass **Eq** a
- **Ord** is polymorphic
- One line to give type signature for 4 functions

Definition of Type Class Enum

- An enumeration is a "numbering" of values
- The class **Enum** describes types whose elements can be numbered; each value can correspond to a unique integer

```
class Enum a where
  fromEnum :: a -> Int
  toEnum   :: Int -> a
```

Other type classes you need to know

- **Show** a : the class of objects that can be displayed on the screen.
  - Most types should be in this class.
  - Functions are not in this class. They cannot be displayed.
- **Num** a : the class of numeric values
  - All the number types are in this class: **Int** **Integer** **Float** **Double**

Note

- The type classes we have seen are built in to Haskell
- I have shown you the definitions, but you do not have to type them in to use them.
- They are already defined in the Haskell Prelude.

Instantiating a type class

- Type classes are collections of types with common overloaded operators.
- Any data type defined by **data** can be included in any type class, using the **instance** declaration.

Example of Instantiating a Type Class

```
data Day = Monday | Tuesday | Wednesday
         | Thursday | Friday | Saturday | Sunday

instance Eq Day where
  (==) = dayEq

dayEq  Monday    Monday    = True
dayEq  Tuesday   Tuesday   = True
dayEq  Wednesday Wednesday = True
dayEq  Thursday  Thursday  = True
dayEq  Friday    Friday    = True
dayEq  Saturday  Saturday  = True
dayEq  Sunday    Sunday    = True
dayEq  x         y         = False
```

Type Class Default Implementations I

- We defined `(==)` on `Day`
- But we did not define `(/=)`
- Trivial implementation: `x /= y = not (x == y)`
- Haskell feature: A type class can define "default" implementations:

```
-- desert island implementation of Eq
class Eq a where
  (==) :: a -> a -> a
  x /= y = not (x == y)
```

- The function `(==)` is given a signature, but not an implementation
  - The programmer must provide the implementation for any instance

Type Class Default Implementations II

- The function `(/=)` is defined in terms of `(==)`
  - This is a "default" implementation
  - An instance of **Eq** must define `(==)`
  - An instance of **Eq** may define `(/=)`
    (if the default is unsuitable)

Default implementations for **Ord**

```
class (Eq a) => (Ord a) where
  (<),(<=),(>),(>=) :: a -> a -> Bool
  (x <= y)          = (x < y) || (x == y)
  (x >= y)          = (x > y) || (x == y)
  (x > y)           = not (x <= y)
```

- To make an instance of **Ord**, the programmer must implement $<$
- A programmer can over-ride any default definition.

Example: Bool I

- Example: **Bool** can be declared an instance of **Ord** and **Enum** as follows:

```
instance Enum Bool where
  toEnum  False = 0
  toEnum  True  = 1
  fromEnum 0    = False
  fromEnum 1    = True

instance Ord Bool where
  False < False    = False
  False < True     = True
  True < False     = False
  True < True      = False
```

- We had to implement **<** on **Bool**

Example: Bool II

- In this example, $<$ means "appears earlier in the ordering"
- This declaration defines only $<$; the others are default implementations

Tedious Instance Declarations

- Some instance declarations are tedious
- Example: recall Day

```
data Day = Monday | Tuesday | Wednesday
         | Thursday | Friday  | Saturday
         | Sunday
```

- To make this an instance of **Eq**, **Ord**, and **Enum** would be quite tedious

Automatic Instance Declarations I

- Haskell compilers can generate automatic instance declarations

```
data Day = Monday | Tuesday | Wednesday
         | Thursday | Friday | Saturday
         | Sunday
  deriving (Eq, Ord, Enum, Show)
```

- Using the **deriving** clause forces the compiler to generate sensible code for the data-type automatically.
  - The "automatic code" is sensible
    eg, Monday == Monday = **True**, etc
    eg, Monday < Monday = **False**, etc
  - For **Enum**, the values are mappable to numbers in order, starting from 0

Perspective

- Type synonyms make programs more readable.
- Type variables allow programmers to define polymorphic functions.
- Datatypes allow Haskell programmers to define new data.
- To write functions on new datatypes, use function definitions based on patterns
- Haskell provides type classes to define common functions on members of the class through overloading
- Haskell takes the drudgery out of programming by allowing for automatic instance declarations.