Haskell: From Basic to Advanced

Part 2 – Type Classes, Laziness, IO, Modules

Qualified types

- In the types schemes we have seen, the type variables were *universally quantified*, e.g.

```
++ :: [a] -> [a] -> [a]
map :: (a -> b) -> [a] -> [b]
```

- In other words, the code of **`++`** or **`map`** could assume *nothing* about the corresponding input

- What is the (principal) type of **`qsort`**?
  - we want it to work on *any list whose elements are comparable*
  - but nothing else
- The solution: qualified types

The type of **qsort**

```
-- File: qsort2.hs
qsort [] = []
qsort (p:xs) =
  qsort lt ++ [p] ++ qsort ge
    where lt = [x | x <- xs, x < p]
          ge = [x | x <- xs, x >= p]
```

```
Prelude> :l qsort2.hs
[1 of 1] Compiling Main             ( qsort2.hs, interpreted )
Ok, modules loaded: Main.
*Main> :t qsort
qsort :: Ord a => [a] -> [a]
```

- The type variable a is *qualified* with the type class **`Ord`**
- **`qsort`** works only with any list whose elements are instances of the **`Ord`** type class

Note: A type variable can be qualified with more than one type class

Type classes and instances

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

defines a *type class* named **`Ord`**

```
data Student = Student Name Score
type Name = String
type Score = Integer

better :: Student -> Student -> Bool
better (Student n1 s1) (Student n2 s2) = s1 > s2
```

Note: we can use the same name for a new data declaration and a constructor

```
instance Ord Student where
  x > y = better x y
  x <= y = not (better x y)
```

makes **`Student`** an *instance* of **`Ord`**

Note: The actual **`Ord`** class in the standard **`Prelude`** defines more functions than these two

Type classes

- Haskell's type class mechanism has some parallels to Java's interface classes
- **Ad-hoc polymorphism** (also called **overloading**)
  - for example, the `>` and `<=` operators are overloaded
  - the `instance` declarations control how the operators are implemented for a given type

Some standard type classes

`Ord`	used for totally ordered data types
`Show`	allow data types to be printed as strings
`Eq`	used for data types supporting equality
`Num`	functionality common to all kinds of numbers

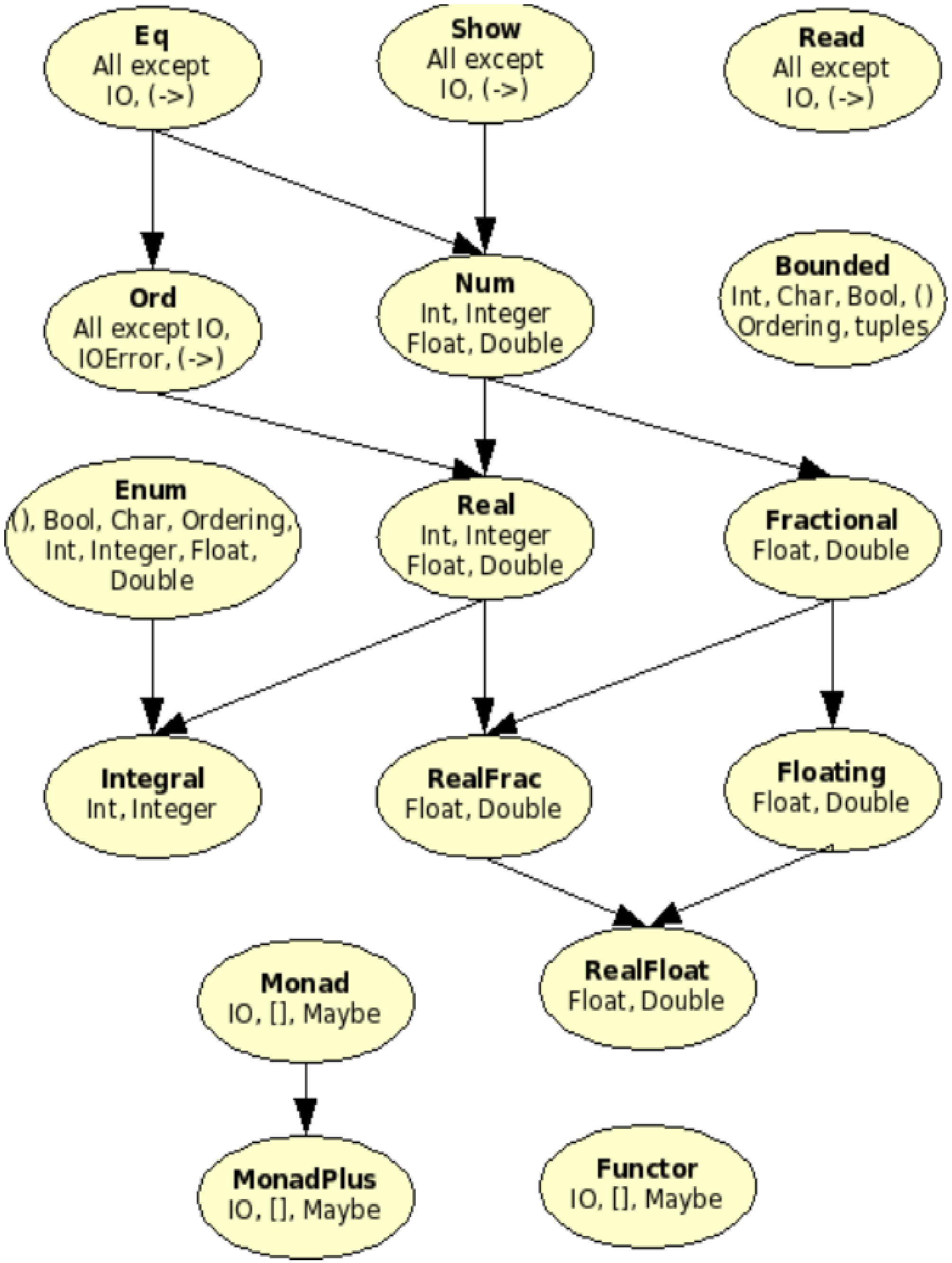
Example: equality on Booleans

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

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

```
instance Eq Bool where
   True == True = True
   False == False = True
   _ == _ = False
   x /= y = not (x == y)
```

Predefined classes and instances



Referential transparency

- *Purely functional* means that *evaluation has no side-effects*
  - A function maps an input to an output value and does nothing else (i.e., is a "real mathematical function")

- **Referential transparency**:

  "*equals can be substituted with equals*"

We can disregard evaluation order and duplication of evaluation

`f x + f x` is always same as `let y = f x in y + y`

Easier for the programmer (and compiler!) to reason about code

Lazy evaluation

```haskell
-- a non-terminating function
loop x = loop x
```

```
Prelude> :l loop
[1 of 1] Compiling Main             ( loop.hs, interpreted )
Ok, modules loaded: Main.
*Main> length [fac 42,loop 42,fib 42]
3
```

- We get a “correct” answer immediately
- Haskell is lazy: computes a value only when needed
  - none of the elements in the list are computed in this example
  - functions with undefined arguments might still return answers
- Lazy evaluation can be
  - efficient since it evaluates a value at most once
  - surprising since evaluation order is not “the expected”

Lazy and infinite lists

- Since we do not evaluate a value until it is asked for, there is no harm in defining and manipulating infinite lists

```
from n = n : from (n + 1)

squares = map (\x -> x * x) (from 0)

even_squares = filter even squares

odd_squares = [x | x <- squares, odd x]
```

```
Prelude> :l squares
[1 of 1] Compiling Main             ( squares.hs, interpreted )
Ok, modules loaded: Main.
*Main> take 13 even_squares
[0,4,16,36,64,100,144,196,256,324,400,484,576]
*Main> take 13 odd_squares
[1,9,25,49,81,121,169,225,289,361,441,529,625]
```

- Avoid certain operations such as printing or asking for the length of these lists...

Programming with infinite lists

- The (infinite) list of all Fibonacci numbers

```
fibs = 0 : 1 : sumlists fibs (tail fibs)
  where sumlists (x:xs) (y:ys) = (x + y) : sumlists xs ys
```

```
Prelude> :l fibs
[1 of 1] Compiling Main             ( fibs.hs, interpreted )
Ok, modules loaded: Main.
*Main> take 15 fibs
[0,1,1,2,3,5,8,13,21,34,55,89,144,233,377]
*Main> take 15 (filter odd fibs)
[1,1,3,5,13,21,55,89,233,377,987,1597,4181,6765,17711]
*Main> take 13 (filter even fibs)
[0,2,8,34,144,610,2584,10946,46368,196418,832040,3524578,14930352]
```

- Two more ways of defining the list of Fibonacci numbers using variants of **`map`** and **`zip`**

```
fibs2 = 0 : 1 : map2 (+) fibs2 (tail fibs2)
  where map2 f xs ys = [f x y | (x,y) <- zip xs ys]
-- the version above using a library function
fibs3 = 0 : 1 : zipWith (+) fibs3 (tail fibs3)
```

Lazy and infinite lists

`[n..m]` shorthand for a list of integers from **n** to **m** (inclusive)

`[n..]` shorthand for a list of integers from **n** upwards

We can easily define the list of all prime numbers

```
primes = sieve [2..]
  where sieve (p:ns) = p : sieve [n | n <- ns, n `mod` p /= 0]
```

```
Prelude> :l primes
[1 of 1] Compiling Main             ( primes.hs, interpreted )
Ok, modules loaded: Main.
*Main> take 13 primes
[2,3,5,7,11,13,17,19,23,29,31,37,41]
```

Infinite streams

- A *producer* of an infinite stream of integers:

```
fib  = 0 : fib1
fib1 = 1 : fib2
fib2 = add fib fib1
  where add (x:xs) (y:ys) = (x+y) : add xs ys
```

- A *consumer* of an infinite stream of integers:

```
consumer stream n =
  if n == 1 then show head
  else show head ++ ", " ++ consumer tail (n-1)
    where head:tail = stream
```

consumer fib 10 ⇒ ... ⇒ "0, 1, 1, 2, 3, 5, 8, 13, 21, 34"

Drawbacks of lazy evaluation

- More difficult to reason about performance
  - especially about space consumption
- Runtime overhead

The five symptoms of laziness:

1.

Side-effects in a pure language

- We really need side-effects in practice!
  - I/O and communication with the outside world (user)
  - exceptions
  - mutable state
  - keep persistent state (on disk)
  - ...

- How can such *imperative* features be incorporated in a *purely functional language*?

Doing I/O and handling state

- When doing I/O there are some desired properties
  - It should be done. Once.
  - I/O statements should be handled in sequence

- Enter the world of **Monad**s* which
  - encapsulate the state, controlling accesses to it
  - effectively model *computation* (not only sequential)
  - clearly separate pure functional parts from the impure

Do it in a Monad

... and remain pure!

— with Haskell, the 100% pure, lazy functional programming language

* A notion and terminology adopted from **category theory**

The `IO` type class

- **Action**: a special kind of value
  – e.g. reading from a keyboard or writing to a file
  – must be ordered in a well-defined manner for program execution to be meaningful
- **Command**: expression that evaluates to an action

- **`IO T`**: a type of command that yields a value of type **`T`**
  – `getLine :: IO String`
  – `putStr  :: String -> IO ()`
- Sequencing IO operations (the *bind* operator):

```
(>>=) :: IO a -> (a -> IO b) -> IO b
```

current state second action new state

Example: command sequencing

- First read a string from input, then write a string to output

```
getLine >>= \s -> putStr ("Simon says: " ++ s)
```

- An alternative, more convenient syntax:

```
do s <- getLine
   putStr ("Simon says: " ++ s)
```

- This looks very “imperative”, but all side-effects are controlled via the **IO** type class!
  - **IO** is merely an instance of the more general type class **Monad**

```
(>>=) :: Monad m => m a -> (a -> m b) -> m b
```

  - Another application of **Monad** is simulating mutable state

Example: copy a file

- We will employ the following functions:

```
Prelude> :info writeFile
writeFile :: FilePath -> String -> IO ()    -- Defined in `System.IO'
Prelude> :i FilePath
type FilePath = String                      -- Defined in `GHC.IO'
Prelude>:i readFile
readFile :: FilePath -> IO String           -- Defined in `System.IO'
```

- The call **`readFile "my_file"`** is not a String, and no String value can be extracted from it
- But it can be used as part of a more complex sequence of instructions to compute a String

```
copyFile fromF toF =
  do contents <- readFile fromF
     writeFile toF contents
```

Monads

- As we saw, Haskell introduces a **do** notation for working with monads, i.e. introduces sequences of computation with an implicit state

```
do expr1; expr2; ...
```

- An "assignment" "expands" to

```
do x <- action1; action2
```

```
action1 >>= \x -> action2
```

- A monad also requires the **return** operation for returning a value (or introducing it into the monad)
- There is also a sequencing operation that does not take care of the value returned from the previous operation

  Can be defined in terms of bind: `x >> y = x >>= (\_ -> y)`

Modules

- Modularization features provide
  - *encapsulation*
  - *reuse*
  - *abstraction*

  (separation of name spaces and information hiding)
- A module *requires* and *provides* functionality

```
module Calculator (Expr,eval,gui) where
import Math
import Graphics
...
```

- It is possible to export everything by omitting the export list

Modules: selective export

- We need not export all constructors of a type
- Good for writing ADTs: supports hiding representation

```
module AbsList (AbsList, empty, isempty,
                cons, append, first, rest) where

data AbsList a = Empty
               | Cons a (AbsList a)
               | App (AbsList a) (AbsList a)

empty = Empty
cons x l = Cons x l
append l1 l2 = App l1 l2
...
```

- Here we export only the type and abstract operations

Modules: import

- We can use **import** to use entries from another module

```
module MyMod (...) where
import Racket (cons, null, append)
import qualified Erlang (send, receive, spawn)

foo pid msg queue = Erlang.send pid (cons msg queue)
```

- Unqualified import allows to use exported entries as is
  - \+ shorter symbols
  - − risk of name collision
  - − not clear which symbols are internal or external
- Qualified import means we need to include module name
  - − longer symbols
  - \+ no risk of name collision
  - \+ easy distinction of external symbols

A better quick sort program

- Recall the **`qsort`** function definition

```
qsort [] = []
qsort (p:xs) = qsort lt ++ [p] ++ qsort ge
    where lt = [x | x <- xs, x < p]
          ge = [x | x <- xs, x >= p]
```

- We can avoid the two traversals of the list by using an appropriate function from the **`List`** library

```
import Data.List (partition)

qsort [] = []
qsort (p:xs) = qsort lt ++ [p] ++ qsort ge
    where (lt,ge) = partition (<p) xs
```

Exercise: sort a file (with its solution)

- Write a module defining the following function:

```
sortFile :: FilePath -> FilePath -> IO ()
```

- **`sortFile file1 file2`** reads the lines of **`file1`**, sorts them, and writes the result to **`file2`**

- The following functions may come handy

```
lines   :: String -> [String]
unlines :: [String] -> String
```

```
module FileSorter (sortFile) where
import Data.List (sort)       -- or use our qsort

sortFile f1 f2 =
  do str <- readFile f1
     writeFile f2 ((unlines . sort . lines) str)
```

Summary so far

- **Higher-order functions**, **polymorphic functions** and **parameterized types** are useful for building abstractions
- **Type classes** and **modules** are useful mechanisms for structuring programs
- **Lazy evaluation** allows programming with infinite data structures
- Haskell is a **purely** functional language that can avoid redundant and repeated computations
- Using **monads**, we can control side-effects in a purely functional language