Haskell Study Notes

Isabelle

May 11, 2021

Contents

1 Types

1.1 Basic types

Haskell has many primitive types such as strings, characters, integers and floating point numbers.

```
"hello world" -- string
1234          -- integer
3.14          -- float
```

1.2 Typeclasses

Typeclasses are a way of sharing specific functionality between types. We can either implement our own instances of typeclasses, or let haskell *derive* them automatically.

They are kind of like Java interfaces.

Num is the generic base typeclass that all numbers derive from.

1. Integral numbers (*Integral* typeclass)
    - *Int*: fixed precision integer with a min and maximum size
    - *Integer*: supports **very** large integers
2. Floating point numbers (*Fractional* typeclass)
    - *Float*: single precision floating point number
    - *Double*: double precision floating point number

1.3 More on typeclasses

Anything that derives from

- *Show* can be printed
- *Read* can be read as a value
- *Eq* can be compared for equality with $==$ and $/=$
- *Ord* can be compared and ordered with ¡ and ¿

```haskell
{-# LANGUAGE DuplicateRecordFields #-}

data Worker = Worker { name :: String, job :: String } deriving (Show)
data Student = Student { name :: String, school :: String } deriving (Show)

class Person a where
    getName :: a -> String
    getOccupation :: a -> String

instance Person Worker where
    getName x = name (x :: Worker)
    getOccupation x = "Working on " ++ job x

instance Person Student where
    getName x = name (x :: Student)
    getOccupation x = "Studying at " ++ school x
```

Figure 1: Demonstration of a custom typeclass for Person

2 More on types

2.1 Record types

Haskell has support for record types which can basically be seen as Haskells version of a C struct.

Let's create a type to represent a *Person*.

```
--                     fName  lName  age gender height
data Person = Person String String Int String Float

firstName :: Person -> String
firstName (Person s _ _ _ _) = s

lastName :: Person -> String
lastName (Person _ s _ _ _) = s

...
```

Yikes thats not readable, lets write it in the more readable *record syntax* which is **identical** to the above syntax but much more organized.

```
data Person = Person { firstName :: String
                     , lastName  :: String
                     , age :: Int
                     , gender :: String
                     , height :: Float
                     } deriving (Show)
```

3 Lists and Tuples

3.1 Lists

Haskell lists are represented as linked lists. A node in a linked list can either be *Nil* or a pointer to the *next node*. Lists have a *head* and a *tail*. Because of Haskell being lazily evaluated, lists can be infinite. *take* takes n items from a list, and *drop* drops n items from a list.

list = [1, 2, 3, 4, 5]
In the above list, the *head* is 1, and the *tail* is [2, 3, 4, 5]

Lists are made up of individual cons cells. The implementation for the *[]* type in Haskell is:

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

-- Defining it ourselves
data List a = Nil | Cons a (List a)

-- Creating a list using our list type
Cons 1 (Cons 2 (Cons 3 Nil))

-- So a list in Haskell is basically just 1:2:3:[] and
    [1,2,3] is syntactic sugar
```

[1]

The *splitAt* function splits a list into two parts at the element specified. Lists can be indexed using the *!!* operator.

3.1.1 List Comprehensions

List comprehensions are similar to how they work in Python. They must have at least one list that is the generator, which provides the input.

[*operation* | $x \leftarrow$ *list*]

An example of a list comprehension:

`[x ^ 2 | x <- [1..10]]`

List comprehensions can have predicates (conditions)

`[x ^ 2 | x <- [1..10], x ‘mod‘ 2 == 0]`

... and they can pull from multiple generators/inputs too

`[(x, y) | x <- [1,2,3], y <- [’a’, ’b’]]`

List comprehensions can also be bound to *variables*.

[1] The spine is a way to refer to the structure that glues a collection of values together. In the list datatype it is formed by the recur- sive nesting of cons cells.

```haskell
let square' = [x ^ 2 | x <- [1..10]]
```

[2]

3.2 Tuples

Haskell has tuples, triples, and *n-tuples*. Tuples have a *fst* and *snd* function which respectively gets either the first or second item.

swap (defined in **Data.Tuple**) swaps the items in a *tuple*

```haskell
("char", 20)

-- a triple
("char", 20, "turtles")
```

[2]When we say variables, the mathematical meaning of variables is meant. This means we can bind values to identifiers, but we can **never** change the values of those bindings.

4 More operations on lists

4.1 Filtering

`filter` has the following type signature *filter* :: (a → b) → [a] → [a]. *filter*

```
filter _ []    = []
filter pred (x:xs)
  | pred x     = x : filter pred xs
  | otherwise = filter pred xs
```

Filter all the **even** numbers from the list, removing the odd ones.

```
filter even [1..10]
-- [2,4,6,8,10]
```

Filters can have anonymous lambda syntax instead of directly passing higher-order functions to it.

```
filter (\x -> (x `rem` 2 == 0)) [1..20]
-- [2,4,6,8,10,12,14,16,18,20]
```

4.2 Zipping Lists

Zipping lists is a means of **combining** values from multiple lists into a single list. `zipWith` allows you to use a combining function to product a list of results by zipping two lists together.

```
zip [1, 2, 3] [4, 5, 6]
-- [(1,4), (2,5), (3,6)]

-- The lists can be different types
zip [1, 2, 3] ['a', 'b', 'c']
-- [(1,'a'), (2,'b'), (3,'c')]
```

We can use `unzip` to unzip a zipped list and recover the contents of it before it was zipped with `zip` or `zipWith`.

zipWith :: (a → b → c) → [a] → [b] → [c])

5 Conditionals

5.1 If expressions

Haskell doesnt have if statements, however it has if expressions instead.

```
-- stolen from the haskell book
let x = 0
if (x + 1 == 1) then "AWESOME" else "wut"
```

5.2 Case expressions

Case expressions are similar to switch-case from languages like Java, and C++. Case expressions begin with `case x of` and their body contains all the different cases in the format *value* → *return-value*.

```
pal xs =
  case y of
    True  -> "yes"
    False -> "no"
  where y = xs == reverse xs
```

It can also be used to *pattern match* against data types

```
data Animal = Cat | Dog
speak a =
  case a of
    Cat -> "meow"
    Dog -> "bork"
```

5.3 Guards

There are also guards which can provide a nicer way of pattern matching instead of writing if-else expressions or case blocks. Guard blocks are written as a series of cases, along with a fallback case called `otherwise`

Cases are written in the format: "| *condition* = *value*."

```haskell
abs n
  | n < 0      = -n
  | otherwise = n
```

5.4 Pattern matching

Pattern matching is very common in Haskell. It allows great flexibility since you can pattern match against anything to extract values etc.

_ or x means a catch-all/fallback pattern to match on if all else fails

```haskell
isItOne :: Int -> Bool
isItOne 1 = True
isItOne _ = False
```

5.4.1 Matching against data constructors

Pattern matching against data constructors is possible. The example defines two functions called `getName` and `getAccNumber` that pattern match on the *User* data constructor to extract either the name <u>or</u> account number.

```haskell
data User = User String Int

getName :: User -> String
getName (User name _) = name

getAccNumber :: User -> Int
getAccNumber (User _ acc) = acc
```

[3] [4]

[3] If we don't use a field, we can use an _ just like pattern matching with functions to signify the field is ignored

[4] Pattern matching against data constructors can get real old if you have a lot of parameters. So an alternative to this that will be covered later is record types.

6 Functions

Functions are defined like add x y = x + y. All functions are pure, unless stated otherwise which means they cannot modify state, or perform side effects like input output, writing to a database, etc

6.1 Currying

By default, all functions are curried in Haskell. Given the following type signature

```
-- add takes two ints and returns an int
add :: Int -> Int -> Int
```

we would say that the function `add` takes two integers, and returns an integer. However, because of how currying works it actually means.

- `add` takes a *single* integer parameter
- and returns a function that takes another integer parameter, and eventually will return an integer itself.

Currying is useful because it means we can create new functions by *partially applying* functions to parameters.

addOne 5 evaluates to *((add 1) 5)* which is then further reduced to the normal form .

```
add :: Int -> (Int -> Int)

addOne :: Int -> Int
addOne x = add 1

addOne 5
((add 1) 5)
6
```

6.2 Higher-order functions

Higher order functions are functions that can accept functions as parameters `flip` is an example of a higher-order function.

Its type signature is *flip :: (a* $\rightarrow$ *b* $\rightarrow$ *c)* $\rightarrow$ *b* $\rightarrow$ *a* $\rightarrow$ *c* and it can be partially applied like so

```
flipOne = flip 1
partialApply = flipOne 2 -- ((flip 1) 2)
```

6.3 Function composition

Composing functions is like a *right to left* pipeline. **f . g** can be read as **f** after **g**. In Haskell . is used since ∘ is not a valid ASCII character.

So (f ∘ g) x = f (g x)

1. first applies g
2. then applies f to the result of applying g
3. and makes a new function which takes a parameter x

Example of function composition in action

```haskell
negate . sum $ [1,2,3,4,5]

-- which is equivalent to
negate (sum [1,2,3,4,5])
negate (15)
```

[5]

[5] $ is used since function application has the highest precedence, so Haskell will think we mean negate . 15. Alternatively, you can wrap it in brackets like **negate . (sum [1,2,3,4,5])**

7 Folds

Folds as a general concept are called catamorphisms. Catamorphisms are a means of deconstructing data. If the spine of a list is the structure of a list, then a fold is what can reduce that structure.

7.1 Foldr

Foldr is short for "fold right". This is the most common fold that you will want to use often with lists. The type signature is `foldr :: Foldable t => (a -> b -> b) -> b -> ta -> b` in GHC 7.10 and newer.

GHC 7.10 abstracted out the list-specific part of folding into a typeclass called `Foldable` to allow you to reuse the same folding functions for any data type that can be folded.

```
-- Remember how map worked?
map :: (a -> b) -> [a] -> [b]

map (+1) 1 :      2 :      3 : []
map (+1) 1 : (+1) 2 : (+1) 3 : []

-- foldr works similar
foldr (+) 0 (1   :   2   :   3   :   [])
1   + ( 2   +   (3  +   0))
```

map applies a function to each item of a list and returns a list, whereas a *fold* replaces the cons constructors with the function and **reduces** the list.

7.1.1 Associativity

Foldr is right associative, which means that it associates to the right.

In `foldr (+) 0 [1..10]`, *0* is the **identity** for the function. If this were to be implemented recursively like:

```
sum :: [Int] -> Int
sum []     = 0 -- the base case translates to the identity for foldr
sum (x:xs) = x + sum xs
```

7.1.2 Reducing

One way to think about the way Haskell evaluates folds is that its like a text rewriting system. Our expression has rewritten itself from `foldr (+) 0 [1, 2, 3]` into:

```
(+) 1 ( (+) 2 ( (+) 3 0) )
```

which can be reduced by evaluating the inner-most parentheses:

```
1 + (2 + (3 + 0))
1 + (2 + 3)
1 + 5
6
```

```
-- we're reducing:
foldr (+) 0 [1, 2, 3]

-- first step, whats ‘xs’ in our case expression?
foldr (+) 0 [1, 2, 3] =
case [1, 2, 3] of
[]      -> 0
(x:xs)  -> f x (foldr f z xs) <-- this matches

-- next, what are f, x, xs, and z in that branch of the case?
foldr (+) 0 [1, 2, 3] =
case [1, 2, 3] of
[]              -> 0
(1 : [2,3])     -> (+) 1 (foldr (+) 0 [2, 3])

-- there is (+) 1 implicitly wrapped around this
-- continuation of the recursive fold
foldr (+) 0 [2, 3] =
case [2, 3] of
[]          -> 0 -- this didn't match again
(2 : [3]) -> (+) 2 (foldr (+) 0 [3])

-- next recursion
foldr (+) 0 [2] =
case [3] of
[]          -> 0 -- this didn't match again
(3 : []) -> (+) 3 (foldr (+) 0 [])

-- there is (+) 1 ( (+) 2 ( (+) 3 ..) ) implicitly wrapped around
-- this continuation of the fold

-- last recursion, end of the spine
foldr (+) 0 [] =
case [] of
[]      -> 0 -- finally matches!
-- ignore other case, it didnt happen
```

7.2 Foldl

Because of the way lists work, folds must *first* recurse over the spine of the list from the beginning to the end. Left folds traverse the spine in the same direction as right folds, but their folding process is left associative and proceeds in the opposite direction as that of foldr.

A simple definition of `foldl` could look like:

```
foldl :: (b -> a -> b) -> b -> [a] -> b
foldl f acc []       = acc
foldl f acc (x:xs)   = foldl f (f acc x) xs

-- Given the list
foldl (+) 0 (1 : 2 : 3 : [])

-- foldl associates like
(  ( 0 + 1 ) + 2 ) + 3 )

-- in contrast to foldr being
( 3 + ( 2 + ( 1 + 0 ) ) )
```

7.3 Scans

Scans are similar to folds except that scans return a list of all the intermediate stages of the fold.

```
Prelude> foldr (+) 0 [1..5]
15

Prelude> scanr (+) 0 [1..5]
[15, 14, 12, 9, 5, 0]

Prelude> foldl (+) 0 [1..5]
15

Prelude> scanl (+) 0 [1..5]
[0, 1, 3, 6, 10, 15]
```

The relationship between the scans and folds are:

```
last (scanl f z xs) = foldl f z xs
head (scanr f z xs) = foldr f z xs

foldr :: (a -> b -> b) -> b -> [a] -> b
scanr :: (a -> b -> b) -> b -> [a] -> [b]

foldl :: (b -> a -> b) -> b -> [a] -> b
scanl :: (b -> a -> b) -> b -> [a] -> [b]
```

The *primary* difference bwteen folds and scans is that scans **always** return a list. Folds can return a list as a result too, but they don't always.

Because scans return a list, they are *not* catamorphisms and are not folds at all! The type signatures are similar and the routes of traversing the spine and evaluating the cons cells are similar. This means we can use scans in places that we *cant* use a fold – because scans return a *list* of results rather than reducing the spine.

```
scanr (+) 0 [1..3]
[1 + (2 + (3 + 0)), 2 + (3 + 0), 3 + 0, 0]
[6, 5, 3, 0]

scanl (+) 0 [1..3]
[0, 0 + 1, 0 + (1 + 2), 0 + ((1 + 2) + 3)]
[0, 1, 3, 6]
```

7.4 Differences

A left fold has the sucessive steps of the fold as its first argument. The next recursion of the spine isn't intermediated by the folding function (compared to **foldr**) – which means recursion of the spine of the list is unconditional.

What this means, is that having a function that doesn't **force** evaluation of either of its arguments won't change anything.

```
Prelude> const 1 undefined
1

Prelude> (flip const) 1 undefined
*** Exception: Prelude.undefined

Prelude> (flip const) undefined 1
```

vs

```
Prelude> foldr const 0 ([1..5] ++ undefined)
1

Prelude> foldr (flip const) 0 ([1..5] ++ undefined)
*** Exception: Prelude.undefined

...
```

7.5 Use foldl' not foldl

However while `foldl` unconditionally evaluates the spine, you can selectively evaluate the values in the list. This feature means foldl is generally inappropriate for lists that are infinite – as well as large lists.

When you need a left fold, you should use `foldl'` instead. This function works the same as `foldl` but it is **strict**. In other words, it forces evaluation of the evaluates inside cons cells as it traverses the spine.

7.6 Evaluation is just substitution

The fundamental way to think about evaluation in Haskell is as substituting a value.

When we use a *right fold* on a list with a function *f*, and a start value of z, we are replacing the **cons** constructors with our function *f* and the empty list constructor with the start value of z.

```
[1..3] = 1 : 2 : 3 : []
```

```
foldr f z [1, 2, 3] =

1 `f` (foldr f z [2, 3])
```

```
1 `f` (2 `f` (foldr f z [3]))
1 `f` (2 `f` (3 `f` (foldr fz[])))
=
1 `f` (2 `f` (3 `f` z))
```

7.7 Laziness

```
foldr f z (x:xs) = f x (foldr f z xs)
-- rest of the fold    ^^^^^^^^^^^^^^
```

Folding happens in two stages — traversal and folding.

- Traversal is the stage in which the fold recurses over the spine
- Folding is the stage where the values in a list are evaluated/reduced using a function

Foldr is *lazy* which means that if *f* doesn't evaluate it's second argument (rest of the fold), no more of the spine will be forced. One of the consequences of this is that foldr can avoid evaluating not just some or all of the values in the list, but some or all of the list's spine as well!

This means `foldr` works on infinite sized lists too.

7.8 Creating folding functions

When we write folds we first think about what the starting value is for the fold.

This is the *identity* for the function. Examples given below:

- Identity for a list is the empty list ([])
- Identity for summation is 0
- Identity for multiplication is 1
- Identity for strings is either "" (empty string) *or* [] (empty list)

[6]

7.9 Summary

7.9.1 foldr

The rest of the fold (recursive invocation of foldr) is an *argument* to the folding function. It doesn't directly self call as a tail call like `foldl` does.

You can think of it as alternating between applications of `foldr` and the folding function *f*. The next invocation of `foldr` is **conditional** on *f* having asked for more of the results after having folded the list.

[6] Sometimes the fold/recursive function will never reduce to a result. This is called bottoming or reaching the bottom (symbolized as $\perp$).

```haskell
foldr :: (a -> b -> b) -> b -> [a] -> b
```

1. The 'b' we're pointing to in `(a -> b -> b)` is the rest of the fold. Evaluating that evaluates the next application of foldr.
2. *foldr* associates to the right.
3. Works with infinite lists.
4. Is a good choice when you want to transform finite/infinite data structures

7.9.2 foldl

1. Self-calls (tail call) through the list, only beginning to produce values after its reached the end
2. Associates to the left (is left associative)
3. Cannot be used with infinite lists
4. Is neatly useless and should always be replaced with `foldl'` which is strictly evaluated.

8 Algebraic Data Types

8.1 How Datatypes are constructed

```
  data Bool = False  |  True
-- [1] [2] [3] [4]  [5]  [6]
```

In the above example for a *Bool* datatype there are the parts broken down:

1. The keyword *data* which signals that what follows is defining a datatype
2. Type constructor (with no arguments)
3. Equals sign which seperates the *type constructor* from the *data constructor*
4. Data constructor. A data constructor that takes *zero* arguments is called a **nullary constructor**.
5. The pipe which denotes a **sum** type, which indicates a logical disjunction (or).
6. Constructor for the value `True` – another **nullary constructor**.

Let's look at another example taken from the Haskell Book. We will look at the *data constructor* and *type constructors* for the list type.

```
data [] a = [ ]  | a  :  [a]
--   [ 1 ]  [2]       [3]
```

1. Type constructor with an argument. The argument here is a polymorphic type variable, so the lists argument can be of different types.
2. Data constructor for the empty list.
3. Data constructor that takes two arguments: an *a* and also a list of *a* aka *[a]*.

8.1.1 Kinds

If we look closer at the list data type, we can tell based on the data type that it must be applied to a concrete type, like *Integer* or *String* before we have a list of that thing.

Kinds are the types of types, or types one level up. They are represented with *. When something is a fully applied, concrete type its kind is *. When it is `* -> *` it is waiting to be applied.

We can query the kind signature of a type constructor in GHC using `:kind` or `:k`.

```
Prelude> :k Bool
Bool :: *

Prelude> :k [Int]
```

```
[Int] :: *

Prelude> :k []
[] :: * -> *
```

Both `Bool` and `[Int]` are fully applied, concrete types so their kind has no arrows. However, the kind of `[]` is `* -> *` since it still needs to be applied to a concrete type before it itself is a concrete type.

8.1.2 Nullary, unary and product types

A type can be thought of as an enumeration of constructions that have zero or *more* arguments.

All of the following are valid data declarations:

```haskell
-- nullary
data Example0 =
  Example0 deriving (Eq, Show)

-- unary
data Example1 =
  Example1 Int deriving (Eq, Show)

-- product of Int and String
data Example2 =
  Example2 Int String deriving (Eq, Show)
```

8.2 What makes them algebraic

Algebraic datatypes (ADTs) in Haskell are algebraic because we can describe the patterns of their argumement structures using two basic operations: sum and product. The most direct way to explain why they're called sum and product is to demonstrate sum and product in terms of cardinality.

The cardinality of a datatype is the number of possible values it defines. It can be as small as zero, or large as infinite.

The cardinality of `Bool` is 2 since there are two possible values (True or False), and the cardinality of `Int8` is 256 since it can range from -128 to 127 so 128 + 127 + 1 = 256.

8.3 Determining cardinality

8.3.1 Nullary data constructors

```haskell
data Example = MakeExample deriving Show
```

Since *MakeExample* takes no type arguments it's a nullary constructor. Nullary data constructors are constants that only represent themselves as values – so they have a cardinality of 1.

Since `MakeExample` is a single nullary value for the type `Example`, the cardinality of the type is 1.

8.3.2 Unary constructors

A unary data constructors takes exactly one argument. Instead of the data constructor being a constant, known value like in nullary constructors – the value will be constructed at run time from the argument we applied it to.

Datatypes that only contain a unary constructor **always** have the same cardinality as the type they contain.

```haskell
data Goats = Goats Int deriving (Eq, Show)
```

Anything that is a valid Int must **also** be a valid argument to the `Goats` constructor. For cardinality, this means unary constructors are the *identity* function.

8.3.3 Sum types

To determine the cardinality of sum types, we add the cardinalities of their data constructors. `True` and `False` take no type arguments and are nullary constructors, each with a cardinality of 1.

Now we do some arithmetic. Nullary constructors are 1, and sum types are addition of + when we are talking about the cardinality.

```
-- how many values inhabit bool?
data Bool = False | True

True | False = C

-- given that |, the sum type syntax is addition
True + False = C

-- and that False and True are both == 1
1 + 1 = C
1 + 2 = 2

-- the cardinality is 2
C = 2
```

8.3.4 Product types

A product types cardinality is the **product** of the cardinalities of its inhabitants. A product type is like a struct.

```haskell
data QuantomBool = QuantomTrue
                 | QuantomFalse
                 | QuantomBoth deriving (Eq, Show)

data TwoQs =
  MkTwoQs QuantomBool QuantomBool
  deriving (Eq, Show)
```

The datatype `TwoQs` has one data constructor, `MkTwoQs` that takes two arguments – making it a product of the two types that inhabit it.

Each argument is a `QuantomBool` which has a cardinality of 3. Product types can also be defined in **record** syntax.

```haskell
-- using a data constructor
data Person = MkPerson String Int deriving (Eq, Show)

-- using record syntax
data Person = Person { name :: String
                     , age  :: Int
                     } deriving (Eq, Show)

-- both of these are identical but the record syntax
-- generates name and age functions
```

Product types are distributive over sum types.

`a * (b + c) -> (a * b) + (a * c)`

```haskell
data Fiction = Fiction deriving Show
data Nonfiction = Nonfiction deriving Show

data BookType = FictionBook Fiction
              | NonfictionBook Nonfiction
              deriving Show

-- We define two constant/nullary types Fiction and Nonfiction.
-- We have a sum type BookType which has two constructors that takes either one of our types
-- as arguments.

type AuthorName = String
data Author1 = Author1 (AuthorName, BookType)

-- This *isnt* a sum of products so it isn't normal form.

data Author2 =
    Fiction AuthorName
  | Nonfiction AuthorName
  deriving (Eq, Show)
```

```

-- This is *distributive*. Products distribute over sums. Just as we can break a * (b + c) into a * b + b * c,
-- we can break the values into a sum of products.

-- Now it's in normal form since no further evaluation on the constructors
-- can be done until some computation / operation is done with these types.
```

8.3.5 newtype

We use the `newtype` keyword to mark a type that can only ever have a single unary data constructor. A `newtype` cannot be a product type, sum type or contain any nullary constructors.

However, it has some advantages of a vanilla `data` declaration. It has no runtime overhead as it reuses the representation of the type it contains (which is allowed since it cannot be a record (*product type*) or a tagged union (*sum type*).

```haskell
newtype Goats =
  Goats Int deriving (Eq, Show)

newtype Cows =
  Cows Int deriving (Eq, Show)

-- Now we can't accidentally mix up the Goats and Cows
tooManyGoats :: Goats -> Bool
tooManyGoats (Goats n) = n > 42
```

A `newtype` is similar to a type alias in that the distinction between them is gone by compile time. So a *String* really is a *Char[]* and *Goats* above is really an *Int*.

However, one key difference bwteen a `newtype` and a type alias is that you can define typeclass instances for `newtypes` that differ from the instances for their underlying type.

```haskell
class TooMany a where
  tooMany :: a -> Bool

instance TooMany Int where
  tooMany n = n > 42
```

We can use it in the REPL but only if we assign the type `Int` to whatever number we pass in since numeric literals are polymorphic.

So this works (if we remove the `(42 :: Int)` it will blow up since we only defined the instance for `Int`).

`tooMany (42 :: Int)`

Under the hood, `Goats` is still `Int` **but** the *newtype* declaration will allow us to define a custom instance.

```haskell
newtype Goats = Goats Int deriving Show

instance TooMany Goats where
  tooMany (Goats n) = n > 43
```

We can use the language extension `GeneralizedNewtypeDeriving` to let us derive from a user-defined typeclass.

Without it, we would have to write

```haskell
class TooMany a where
  tooMany :: a -> Bool

instance TooMany Int where
  tooMany n = n > 42

newtype Goats = Goats Int deriving (Eq, Show)

-- will do the same thing as the Int instance, but we still have to
-- define it seperately without using the extension.
instance TooMany Goats where
  tooMany (Goats n) = tooMany n
```

But with the extension we can write

```haskell
{-# LANGUAGE GeneralizedNewtypeDeriving #-}

class TooMany a where
  tooMany :: a -> Bool

instance TooMany Int where
  tooMany n = n > 42

-- we just deriving from TooMany
newtype Goats =
  Goats Int deriving (Eq, Show, TooMany)
```

9 Weak head normal form

Values in Haskell are reduced to Weak head normal form (WHNF). Weak head normal form is a type of normal form which contains both the possibility the expression has been fully evaluated/reduced (normal form) as well as the possibility that the expression has been evaluated to the point of arriving at a data constructor/lambda waiting for an argument.

These expressions **are** in WHNF:

```
-- https://stackoverflow.com/questions/6872898/
(1, 1 + 1)          -- outermost part is the data constructor (,)
\x -> 2 + 2         -- outermost part is a lambda abstraction
'h' : ("e" + "llo") -- outermost part is the data constructor (:)
```

These examples **are not** in WHNF

```
1 + 2               -- the outermost part here is an application of (+)
(\x -> x + 1) 2     -- the outermost part is an application of (\x -> x + 1)
"he" ++ "llo"       -- the outermost part is an application of (++)
```

We can use `:sprint` in GHCI to print out the representation of the value in memory. Due to laziness, polymorphic types are unevaluated (thunked) until we use them. This is marked with an underscore (_).

```
Prelude> let nums = [1..5]

Prelude> :sprint nums
nums = _

Prelude> take 2 nums
Prelude> :sprint nums
nums = 1 : 2 : _
```

[7]

[7] The examples on WHNF were taken directly from the amazing Haskell Book