Learn You a Physics for Great Good!

>>> WORK IN PROGRESS <<<

Dimensions / Quantities

[src: Dimensions/Quantity.lhs] Previous: Type-level dimensions Table of contents Next: Testing of Quantity

Quantities

We'll now create a data type for quantities and combine dimensions on value-level and type-level. Just as before, a bunch of GHC-extensions are necessary.

```
{-# LANGUAGE UndecidableInstances #-}
{-# LANGUAGE FlexibleInstances #-}
{-# LANGUAGE GADTs #-}
{-# LANGUAGE DataKinds #-}
{-# LANGUAGE TypeOperators #-}
{-# LANGUAGE KindSignatures #-}
```

TODO: The module header is overly long. Please group several (related) operations on each line

```
module Dimensions.Quantity
( Quantity
, length
, mass
, time
, current
, temperature
, substance
, luminosity
, one
, velocity
```

```
  , acceleration
  , force
  , momentum
  , (~=)
  , isZero
  , (#)
  , (+#)
  , (-#)
  , (*#)
  , (/#)
  , sinq
  , cosq
  , asinq
  , acosq
  , atanq
  , expq
  , logq
  )
  where


  import qualified Dimensions.ValueLevel as V
  import           Dimensions.TypeLevel  as T
  import           Prelude               as P hiding (length, div)
```

First we create the data type for quantities.

TODO: A bit confusing to use the same name for both the type constructor and the value constructor.

TODO: Please explain the purpose of Quantity? (semantics, intended meaning or use, problem avoided, etc.).

TODO: Perhaps also note that the intended use is with a side-condition (which you don't check) that the value-level dimension matches the type-level dimensions.

```
  data Quantity (d :: T.Dim) (v :: *) where
    Quantity :: V.Dim -> v -> Quantity d v
```

`data Quantity` creates a *type constructor*. Which means it takes two *types* (of certain *kinds*) to create another *type* (of a certain *kind*). For comparsion, here's a *value constructor* which takes two *values* (of certain *types*) as input to create another *value* (of a certain *type*).

```
  data MyDataType = MyValueConstructor String Int
```

TODO: MyValueConstructor is the value constructor, not MyDataType

```
ghci> :t MyValueConstructor
MyDataType :: String -> Int -> MyDataType
```

So this `Quantity` type constructor takes two *types* (of certain *kinds*) as input to create another *type* (of a certain *kind*). See the parallell to the value constructor? It's the same thing, but "one step up" to the type-level.

```
ghci> :k Quantity
Quantity :: T.Dim -> * -> *
```

Instead of checking the *type* (like we do on the value constructor above), we check the *kind*. The kind of `Quantity` shows, just as we said, that it takes two types as input. We also see that the *kind* of the first *type* must be `T.Dim`, i.e., a type-level dimension. The kind of the second type is `*`, which is the kind of types that can have values. Such types are for instance `Double`, `Integer` and `String` (but not `T.Dim`).

The definition of `Quantity` tells us this too. In it, `(d :: T.Dim) (v :: *)` signifies that the type constructor `Quantity` takes a type u of kind `T.Dim` and a type of kind `*`. In this context, `u :: T.Dim` is read as "u has kind `T.Dim`".

So the first row was a *type constructor*. The second row is a *value constructor*. It takes two values, one of type `V.Unit` and the other of type v (v will be bound when the type of the whole thing is decided). This is the same deal as "your average data type" value constructor, like the `MyDataType` example above.

Both the type constructor and the value constructor have the same name `Quantity`. This is legal syntax. Just like the following is legal:

```
data YourAverageDataType = YourAverageDataType Double
```

The left-hand-side type constructor (that happens to take no arguments) and the right-hand-side value constructor have the same name.

Okay, so how does all this tie together? First the *type* is decided, for instance

```
type ExampleType = Quantity T.Length Double
```

then a *value* of that type is created

TODO: This examples comes too late (too far from the type definition). I suggest you start in the opposite order, from example values, and only then introduce the “heavier machinery”. This would also save you from inventing a phony type “MyDataType” - that could become “MyLength” or something like that.

```
exampleValue :: ExampleType
exampleValue = Quantity V.length 5.3
```

TODO: Please rephrase. The definition of “ExampleType” uses the type level and “exampleValue” uses the value level.

TODO: (here and elsewhere) use “print” or “pretty print” or “pretty-print”, not “print prettily”.

Note that the `Quantity` data type has both value-level and type-level dimensions. As previosuly mentioned, value-level in order to print prettily and type-level to only permit legal operations.

Exercise. Create a data type which has two type-level dimensions, always use `Rational` as the value-holding type (the role of `v`) but which has no value-level dimensions.

TODO: What is the purpose (semantics, intended meaning or use) of Quantity’?

Solution.

```
data Quantity' (d1 :: T.Dim) (d2 :: T.Dim) where
  Quantity' :: Rational -> Rational -> Quantity' d1 d2
```

A taste of types

We’ll implement all arithmetic operations on `Quantity`, but for now, to get a taste of types, we show here addition and multiplication and some examples of values of type `Quantity`.

```
quantityAdd :: (Num v) => Quantity d v ->
                          Quantity d v ->
                          Quantity d v
quantityAdd (Quantity d v1) (Quantity _ v2) = Quantity d (v1+v2)
```

The type is interpreted as follows: two values of type `Quantity d v` is the input, where `d` is the type-level dimension. The output is also a value of type `Quantity d v`.

The type of the function forces the inputs to have the same dimensions. For this reason, the dimension on value-level doesn't matter on one of the arguments, because they will be the same. (It's possible to create values where the dimensions on value-level and type-level don't match. We'll come back to this later. TODO: perhaps this comment needs to come earlier to give the reader a feeling for where it is all heading.)

`Quantity d v` in the type and `Quantity d v1` as a value pattern shouldn't be confused (TODO: then rename one!). The first `d` is a type variable, holding the type-level dimension while the second `d` is a (value) variable holding the value-level dimension. Similary, the first `v` is a type variable holding the type of values (e.g. `Double`) and the second `v` holds the actual value.

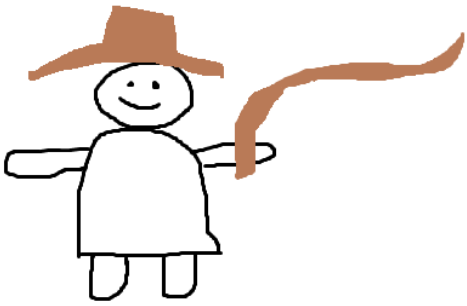
Multiplication is

```
quantityMul :: (Num v) => Quantity d1 v ->
                          Quantity d2 v ->
                          Quantity (d1 `Mul` d2) v
quantityMul (Quantity d1 v1) (Quantity d2 v2) =
  Quantity (d1 `V.mul` d2) (v1*v2)
```

The type has the following interpretation: two values of type `Quantity dx v` is input, where `dx` are two types representing (potentially different) dimensions. As output, a value of type `Quantity` is returned. The type in the `Quantity` will be the type that is the product of the two dimensions in.

Exercise. Implement subtraction and division.

Are you a cowboy?



Solution. Hold on cowboy! Not so fast. We'll come back later to subtraction and division, so check your solution then.

Now on to some example values and types.

```
width :: Quantity T.Length Double
width = Quantity V.length 0.5


height :: Quantity T.Length Double
height = Quantity V.length 0.3


type Area = Mul T.Length T.Length
```

The following example shows that during a multiplication, the types will change, as they should. The dimensions change not only on value-level but also on type-level.

```
area :: Quantity Area Double
area = quantityMul width height


ghci> width
0.5 m
ghci> :t width
width :: Quantity Length Double
ghci> area
0.15 m^2
ghci> :t area
area
 :: Quantity
      ('Dim
         'Numeric.NumType.DK.Integers.Pos2 -- The interesting line
         'Numeric.NumType.DK.Integers.Zero
         'Numeric.NumType.DK.Integers.Zero
         'Numeric.NumType.DK.Integers.Zero
         'Numeric.NumType.DK.Integers.Zero
         'Numeric.NumType.DK.Integers.Zero
         'Numeric.NumType.DK.Integers.Zero)
      Double
```

Which is the same as `Area`.

The type-level dimensions are used below to enforce, at compile-time, that only allowed operations are attempted.

```
-- Doesn't even compile
weird = quantityAdd width area
```

If the dimensions had been value-level only, this error would go undetected until run-time.

Exercise. What is the volume of a cuboid with sides $1.2\ m$, $0.4\ m$ and $1.9\ m$? Create values for the involved quantities. Use `Float` instead of `Double`.

Solution.

```
side1 :: Quantity Length Float
side1 = Quantity V.length 1.2
```

```haskell
side2 :: Quantity Length Float
side2 = Quantity V.length 0.4


side3 :: Quantity Length Float
side3 = Quantity V.length 1.9


type Volume = Length `Mul` (Length `Mul` Length)


volume :: Quantity Volume Float
volume = quantityMul side1 (quantityMul side2 side3)
```

Pretty-printer

Let's do a pretty-printer for quantities. Most of the work is already done by the value-level dimensions.

```haskell
showQuantity :: (Show v) => Quantity d v -> String
showQuantity (Quantity d v) = show v ++ " " ++ show d


instance (Show v) => Show (Quantity d v) where
  show = showQuantity
```

Comparsions

It's useful to be able to compare quantities. Perhaps one wants to know which of two amounts of energy is the largest. But what's the largest of `1 J` and `1 m`? That's no meaningful comparsion since the dimensions don't match. This behaviour is prevented by having type-level dimensions.

```haskell
quantityEq :: (Eq v) => Quantity d v -> Quantity d v -> Bool
quantityEq (Quantity _ v1) (Quantity _ v2) = v1 == v2


instance (Eq v) => Eq (Quantity d v) where
  (==) = quantityEq
```

Exercise. Make `Quantity` an instance of `Ord`.

Solution.

```
quantityCompare :: (Ord v) => Quantity d v ->
                              Quantity d v -> Ordering
quantityCompare (Quantity _ v1) (Quantity _ v2) =
  compare v1 v2


instance (Ord v) => Ord (Quantity d v) where
  compare = quantityCompare
```

We often use `Double` as the value holding type. Doing exact comparsions isn't always possible to due rounding errors. Therefore, we'll create a ~= function for testing if two quantities are almost equal.

```
infixl 4 ~=
(~=) :: Quantity d Double -> Quantity d Double -> Bool
(Quantity _ v1) ~= (Quantity _ v2) = abs (v1-v2) < 0.001
```

Testing if a quantity is zero is something which might be a common operation. So we define it here.

```
isZero :: (Num v, Eq v) => Quantity d v -> Bool
isZero (Quantity _ v) =  v == 0
```

Arithmetic on quantities

Let's implement the arithmetic operations on `Quantity`. Basically it's all about creating functions with the correct type-level dimensions.

```
infixl 6 +#
(+#) :: (Num v) => Quantity d v -> Quantity d v ->
                   Quantity d v
(+#) = quantityAdd


infixl 6 -#
(-#) :: (Num v) => Quantity d v -> Quantity d v ->
                   Quantity d v
(Quantity d v1) -# (Quantity _ v2) = Quantity d (v1-v2)


infixl 7 *#
(*#) :: (Num v) => Quantity d1 v -> Quantity d2 v ->
```

```
                    Quantity (d1 `Mul` d2) v
(*#) = quantityMul


infixl 7 /#
(/#) :: (Fractional  v) => Quantity d1 v ->
                           Quantity d2 v ->
                           Quantity (d1 `Div` d2) v
(Quantity d1 v1) /# (Quantity d2 v2) =
  Quantity (d1 `V.div` d2) (v1 / v2)
```

For all operations on quantities, one does the operation on the value and, in the case of multiplication and division, on the dimensions separetly. For addition and subtraction the in-dimensions must be the same. Nothing then happens with the dimension.

How does one perform operations such as `sin` on a quantity with a *potential* dimension? The answer is that the quantity must be dimensionless, and with the dimension nothing happens.

Why is that, one might wonder. Every function can be written as a power series. For sin the series looks like

$$sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \dots$$

The dimension of *every* term must be the same. The only way for that to be case is if the input x is dimensionless. Then so will the output.

The other functions can be written as similar power series and we'll see on those too that the input and output must be dimensionless.

```
sinq :: (Floating v) => Quantity One v -> Quantity One v
sinq (Quantity dl v) = Quantity d1 (sin v)

cosq :: (Floating v) => Quantity One v -> Quantity One v
cosq (Quantity dl v) = Quantity d1 (cos v)
```

We quickly realize a pattern, so let's generalize a bit.

```
qmap :: (a -> b) -> Quantity One a -> Quantity One b
qmap f (Quantity d1 v) = Quantity d1 (f v)


sinq, cosq, asinq, acosq, atanq, expq, logq :: (Floating v) =>
  Quantity One v -> Quantity One v
```

```
sinq  = qmap sin
cosq  = qmap cos
asinq = qmap asin
acosq = qmap acos
atanq = qmap atan
expq  = qmap exp
logq  = qmap log
```

Why not make `Quantity` an instance of `Num`, `Fractional`, `Floating` och `Functor`? The reason is that the functions of those type classes have the following type

```
(*) :: (Num a) => a -> a -> a
```

which isn't compatible with `Quantity` since multiplication with `Quantity` has the following type

```
(*#) :: (Num v) => Quantity d1 v -> Quantity d2 v ->
                   Quantity (d1 `Mul` d2) v
```

The input here may actually be of *different* types, and the output has a type depending on the types of the input. However, the *kind* of the inputs and output are the same, namely `Quantity`. We'll just have to live with not being able to make `Quantity` a `Num`-instance.

However, operations with only scalars (type `One`) has types compatible with `Num`.

Exercise. `Quantity One` has compatible types. Make it an instance of `Num`, `Fractional`, `Floating` and `Functor`.

Solution.

```
instance (Num v) => Num (Quantity One v) where
  (+) = (+#)
  (-) = (-#)
  (*) = (*#)
  abs = qmap abs
  signum = qmap signum
  fromInteger n = Quantity V.one (fromInteger n)


instance (Fractional v) => Fractional (Quantity One v) where
  (/) = (/#)
  fromRational r = Quantity V.one (fromRational r)
```

```
instance (Floating v) => Floating (Quantity One v) where
  pi    = Quantity V.one pi
  exp   = expq
  log   = logq
  sin   = sinq
  cos   = cosq
  asin  = asinq
  acos  = acosq
  atan  = atanq
  sinh  = qmap sinh
  cosh  = qmap cosh
  asinh = qmap asinh
  acosh = qmap acosh
  atanh = qmap atanh


instance Functor (Quantity One) where
  fmap = qmap
```

Syntactic sugar

In order to create a value representing a certain distance (5 metres, for example) one does the following

```
distance :: Quantity T.Length Double
distance = Quantity V.length 5.0
```

Writing that way each time is very clumsy. You can also do “dumb” things such as

```
distance :: Quantity T.Length Double
distance = Quantity V.time 5.0
```

with different dimensions on value-level and type-level.

To solve these two problems we will introduce some syntactic sugar. First some pre-made values for the 7 base dimensions and the scalar.

TODO: Please use “1” instead of “0” in all the dimensions. That signifies an amount of the “unit” (SI-units => always 1).

```
length :: (Num v) => Quantity Length v
length = Quantity V.length 0
```

```
mass :: (Num v) => Quantity Mass v
mass = Quantity V.mass 0


time :: (Num v) => Quantity Time v
time = Quantity V.time 0
```

Exercise. Do the rest.

Solution.

```
current :: (Num v) => Quantity Current v
current = Quantity V.current 0


temperature :: (Num v) => Quantity Temperature v
temperature = Quantity V.temperature 0


substance :: (Num v) => Quantity Substance v
substance = Quantity V.substance 0


luminosity :: (Num v) => Quantity Luminosity v
luminosity = Quantity V.luminosity 0


one :: (Num v) => Quantity One v
one = Quantity V.one 0
```

And now the sugar.

TODO: Please multiply the values instead of throwing them away. I'm pretty sure (#) should behave as "scale" of a vector space. So that (x#(y#z)) == ((x*y)#z).

```
(#) :: (Num v) => v -> Quantity d v -> Quantity d v
v # (Quantity d _) = Quantity d v
```

The intended usage of the function is the following

```
ghci> let myDistance = 5 # length
ghci> :t myDistance
t :: Num v => Quantity Length v
ghci> myDistance
5 m
```

To create a `Quantity` with a certain value (here 5) and a certain dimension (here `length`), you write as above and get the correct dimension on both value-level and type-level.

`length`, `mass` and so on are just dummy-values with the correct dimension (on both value-level and type-level) in order to easier create `Quantity` values. Any value with the correct dimension on both levels can be used as the right hand side argument.

TODO: (above): please change impl. so that it matters. Then you basically get units on the value level and dimensions at the type level. (you may also want to rename “length” to “meter”, etc. to match this intuition. And you can support “inch”, “foot”, “lightyear”, etc as well, of the same type)

```
ghci> let otherPersonsDistance = 10 # length
ghci> let myDistance = 5 # otherPersonsDistance
ghci> :t myDistance
t :: Num v => Quantity Length v
ghci> myDistance
5 m
```

Precisely the same result.

We want to maintain the invariant that the dimension on value-level and type-level always match. The pre-made values from above maintain it, and so do the arithmetic operations we previously created. Therefore, we only export those from this module! The user will have no choice but to use these constructs and hence the invariant will be maintained.

If the user needs other dimensions than the base dimensions (which it probably will), the follwing example shows how it’s done.

```
ghci> let myLength = 5 # length
ghci> let myTime = 2 # time
ghci> let myVelocity = myLength /# myTime
ghci> myVelocity
2.5 m/s
```

New dimensions are created “on demand”. Furthermore

```
ghci> let velocity = myVelocity
ghci> let otherVelocity = 188 # velocity
```

it’s possible to use the sugar from before on user-defined dimensions.

TODO: Dessa har alla Double som värdetyp. Hur förhindra det? Explicita typsignaturer löser det, men man vill inte att "användaren" ska behöva göra det varje gång.

Just for convenience sake we'll define a bunch of common composite dimensions. Erm, *you'll* define.

Exericse. Define pre-made values for velocity, acceleration, force, momentum and energy.

Solution.

```
velocity     = length   /# time
acceleration = velocity /# time
force        = mass     *# acceleration
momentum     = force    *# time
energy       = force    *# length
```

Alternative sugar with support for units

Exercise. Using the idea of the previous sugar, it's possible to add support for non-SI units (at least for input), by using multiplication on some pre-made values. Define an operator and pre-made values to do this. It should work as follows.

```
ghci> let distance = 5 `op` mile
ghci> :t distance
distance :: Quantity Length Double
ghci> distance
8046.72 m
```

Solution.

```
metre = 1 # length
kilometre = 1000 # length
second = 1 # time
hour = 3600 # time
metresPerSecond = metre /# second
kilometresPerHour = kilometre /# hour
```

Each of these pre-made quantites has as numerical value the corresponding value in the SI-unit.

The sugar function would then look like

```
(##) :: (Num v) => v -> Quantity d v -> Quantity d v
v ## (Quantity d bv) = Quantity d (v*bv)
```

Notice how the value is multiplied with the “base value” of the unit. This function is intended to be used as follows.

```
ghci> let velocity1 = 100 ## kilometresPerHour
ghci> let velocity2 = 33 ## metersPerSecond
ghci> velocity1 +# velocity2
67.77 m/s
```

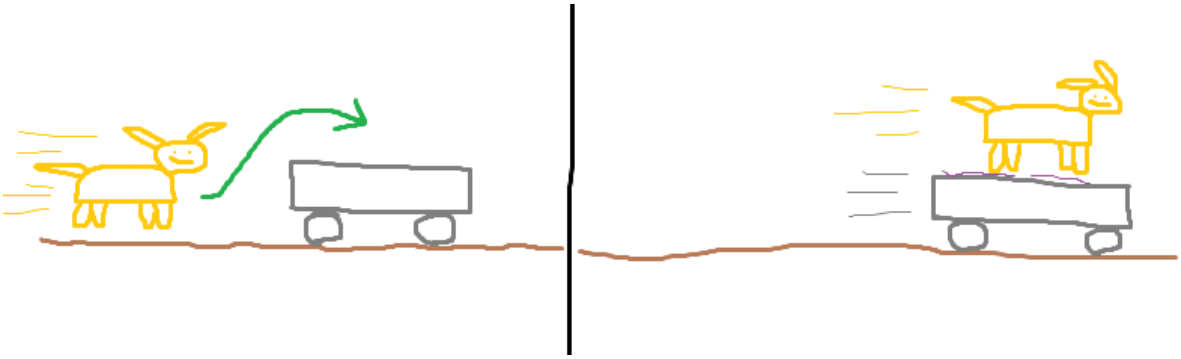
A physics example

To conclude this chapter, we show an example on how to code an exercise and its solution in this domain specific language we’ve created for quantities.

The code comments show what GHCi prints for that line.

The exercise is “A dog runs, jumps and lands sliding on a carriage. The dog weighs $40\,kg$ and runs in $8\,m/s$. The carriage weighs $190\,kg$. The coefficient of friction between the dog’s paws and the carriage is 0.7.”

This is illustrated in the painting below.



a. Calculate the (shared) final velocity of the dog and the carriage.

```
mDog      = 40 # mass     -- 40 kg
viDog     = 8 # velocity  -- 8 m/s
mCarriage = 190 # mass    -- 190 kg
u         = 0.7 # one     -- 0.7
```

No external forces are acting on the dog and the carriage. Hence the momentum of the system is preserved.

```
piSystem = mDog *# viDog -- 320 kg*m/s
pfSystem = piSystem      -- 320 kg*m/s
```

In the end the whole system has a shared velocity of its shared mass.

```
mSystem  = mDog +# mCarriage   -- 230 kg
vfSystem = pfSystem /# mSystem -- 1.39 m/s
```

b. Calculate the force of friction acting on the dog.

```
fFriction = u *# fNormal        -- 275 kg*m/s^2
fNormal   = mDog *# g           -- 393 kg*m/s^2
g         = 9.82 # acceleration -- 9.82 m/s^2
```

c. For how long time does the dog slide on the carriage?

```
aDog = fFriction /# mDog -- 6.87 m/s^2


vDelta = mDog -# vfSystem
```

```
* Couldn't match type 'Numeric.NumType.DK.Integers.Neg1
                  with 'Numeric.NumType.DK.Integers.Zero
...
```

Whoops! That’s not a good operation. Luckily the compiler caught it.

```
vDelta = viDog -# vfSystem -- 6.60 m/s
tSlide = vDelta /# aDog    -- 0.96 s
```

Conclusion

Okay, so you’ve read a whole lot of text by now. But in order to really learn anything, you need to practise with some additional exercises. Some suggestions are

- Implement first value-level dimensions, then type-level dimensions and last quantites by yourself, without looking too much at this text.
- Implement a power function.

- Implement a square-root function. The regular square-root function in Haskell uses `exp` and `log`, which only work on `Quantity One`. But taking the the square-root of e.g. an area should be possible.
- Extending the `Quantity` data type here by adding support for prefixes and non-SI-units.
- Ditching the separate implementations of value-level and type-level, and only use one with `Data.Proxy`.

After reading this text and doing some exercies, we hope you've learnt

- The relation between dimensions, quantities and units.
- How dimensions…
  - …restrict operations such as comparsion to quantities of the same dimension.
  - …change after operations such as multiplication.
  - …can be implemented on the type-level in Haskell to enforce the two above at compile-time.
- The basics about kinds and type-level programming in Haskell.

Further reading

The package Dimensional inspired a lot of what we did here. Our `Quantity` is like a "lite" version of Dimensional, which among other things, support different units and use `Data.Proxy`.

The type-level progamming in this text was done with the help of the tutorial Basic Type Level Programming in Haskell. If you want to know more about kinds and type-level programming, it's a very good starting point.