Learn You a Physics for Great Good!

>>> WORK IN PROGRESS <<<

Dimensions / Usage

[src: Dimensions/Usage.lhs] Previous: Testing of Quantities Table of contents Next: Single particle mechanics

Usage

```
module Dimensions.Usage
(
)
where
```

In this module we'll show how to use value-level dimensions, type-level dimensions and `Quantity` in an "actual" progam. Let's first use this fancy GHC-extension

```
{-# LANGUAGE TypeOperators #-}
```

and then import the things we have created.

```
import Dimensions.TypeLevel
import Dimensions.Quantity
import Prelude hiding (length)
```

We don't need to import `Dimensions.ValueLevel` since all its features are available by using the `Quantity` data type. The same thing could be said about `Dimensions.TypeLevel`. So why do we import it? It's so we can write explicit type signatures, the trick of trade in Haskell progamming.

Values and types

Let's create a length, time and mass, in the way hinted in the previous chapter.

```
myLength = 10.0 # length
myTime   = 20.0 # time
myMass   = 30.0 # mass
```

Okay, let's check out their type and value in GHCi.

```
ghci> myLength
10.0 m
ghci> :t myLength
myLength :: Quantity Length Double
ghci> myTime
20.0 s
ghci> :t myTime
myTime :: Quantity Time Double
```

The pretty printing works thanks to dimensions on *value-level* and the Haskell-types thanks to the *type-level* dimensions. Recall that the value-level dimensions just is a pretty simple "average" data type.

Let's write some type signatures on those values (of type `Quantity`).

```
myLength :: Quantity Length Double
myLength = 10.0 # length
myTime :: Quantity Time Double
myTime = 20.0 # time
myMass :: Quantity Mass Double
myMass = 30.0 # mass
```

Neat! `Length`, `Time` and `Mass` are type-level dimensions. Therefore, `Length` *is* a type, just like `String` is a type.

Remember that the sugary style of writing `10.0 # length` is a shorthand for doing

```
myLength :: Quantity Length Double
myLength = Quantity length' 10.0
```

where `length'` refers to `length` in `Dimensions.ValueLevel`.

The reasons for not allowing this “raw” way of creating the value is to maintain the invariant of having the same value-level and type-level dimension. With the `#` function, the are *forced* by it to be the same. The following try to work around it is not possible.

```
myWeird :: Quantity Length Double
myWeird = 10.0 # time
```

```
* Couldn't match type 'Numeric.NumType.DK.Integers.Zero
                  with 'Numeric.NumType.DK.Integers.Pos1
  Expected type: Quantity Length Double
    Actual type: Quantity Time Double
* In the expression: 10.0 # time
  In an equation for `myWeird': myWeird = 10.0 # time
```

Custom values and types

Creating values of a dimensions *not* exported is also possible. Let’s take energy as an example.

```
energy = force *# length
```

`energy` can now be used just as `length` and so on.

```
energyToBoilMyGlassOfWater = 12 # energy
```

In order to write explicit type signatures, we can do like this

```
type Velocity = Length `Div` Time
type Acceleration = Velocity `Div` Time
type Force = Acceleration `Mul` Mass
type Energy = Force `Mul` Length
```

Note we had to introduce some helper types since, unlike the pre-made `Quantity` values, not as many were created for types.

Now `energy` can be defined and used like

```
energy :: Quantity Energy Double
energy = force *# length
```

```
energyToBoilMyGlassOfWater :: Quantity Energy Double
energyToBoilMyGlassOfWater = 12 # energy
```

Some functions

Let's create a function operating on quantities. It should calculate the area of a rectangle given the lengths of its sides. The type should hence look like

```
areaOfRectangle :: Quantity Length Double -> Quantity Length Double ->
                   Quantity (Length `Mul` Length) Double
```

And the actual function like

```
areaOfRectangle width height = width *# height
```

The values (`width` and `height`) can't be pattern matched on. This is so we can maintain the invariant we keep nagging about. The side effect is that only the comparative and artithmetic operations exported by `Dimensions.Quantity` can be used.

To wrap up, let's create a function to calculate the kinectic energy à la

$$E_k = \frac{m * v^2}{2}$$

```
kinecticEnergy :: Quantity Mass Double ->
                  Quantity Velocity Double ->
                  Quantity Energy Double
kinecticEnergy m v = m *# v *# v /# two
  where
    two = 2.0 # one
```