Learn You a Physics for Great Good!

>>> WORK IN PROGRESS <<<

Dimensions / Type-level dimensions

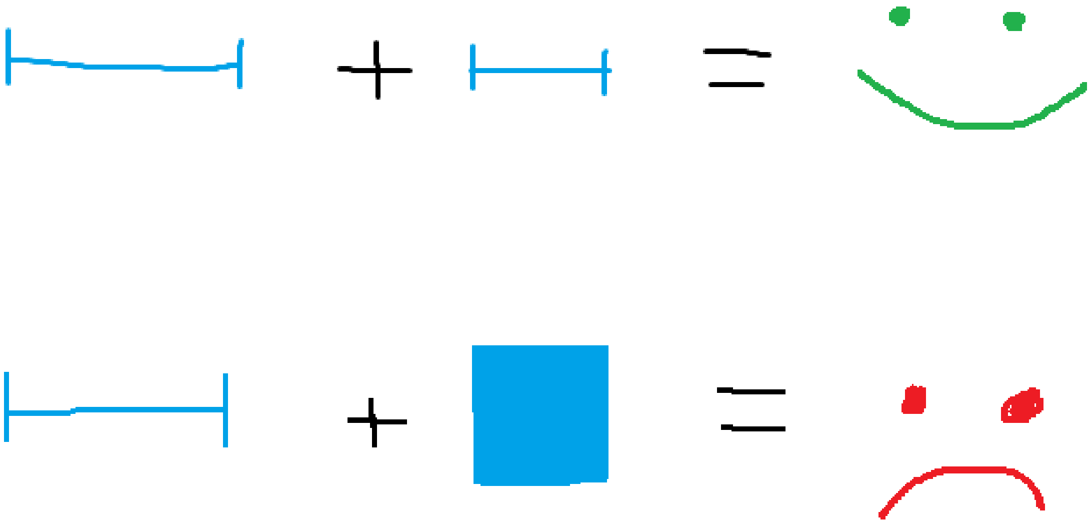
[src: Dimensions/TypeLevel.lhs] Previous: Testing of value-level dimensions Table of contents Next: Quantities

Type-level dimensions

```
module Dimensions.TypeLevel
    ( Dim(..)
    , Mul
    , Div
    , Length
    , Mass
    , Time
    , Current
    , Temperature
    , Substance
    , Luminosity
    , One
    ) where
```

We will now implement *type-level* dimensions. What is type-level? Programs (in Haskell) normally operatate on (e.g. add) values (e.g. `1` and `2`). This is on *value-level*. Now we'll do the same thing but on *type-level*, that is, perform operations on types.

What's the purpose of type-level dimensions? It's so we'll notice as soon as compile-time if we've written something incorrect. E.g. adding a length and an area is not allowed since they have different dimensions.



This implemention is very similar to the value-level one. It would be possible to only have one implementation by using `Data.Proxy`. But it would be trickier. This way is lengthier but easier to understand.

To be able to do type-level programming, we'll need a nice stash of GHC-extensions.

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

See the end of the next chapter to read what they do.

We'll need to be able to operate on integers on the type-level. Instead of implementing it ourselves, we will just import the machinery so we can focus on the physics-part.

```
import Numeric.NumType.DK.Integers
```

We make a *kind* for dimensions, just like we in the previous section made *type* for dimensions. On value-level we made a *type* with *values*. Now we make a *kind* with *types*. The meaning is exactly the same, except we have moved "one step up".

```haskell
data Dim = Dim TypeInt -- Length
               TypeInt -- Mass
               TypeInt -- Time
               TypeInt -- Current
               TypeInt -- Temperature
               TypeInt -- Substance
               TypeInt -- Luminosity
```

But `data Dim = ...` looks awfully similar to a regular data type! That's correct. But with the GHC-extension `DataKinds` this will, apart from creating a regular data type, **also** create a *kind*. Perhaps a less confusing syntax would've been `kind Dim = ...`. The above definition can be seen as the two following definitions.

```haskell
-- LHS: Type
-- RHS: Value
data Dim = Dim TypeInt
               TypeInt
               TypeInt
               TypeInt
               TypeInt
               TypeInt
               TypeInt


-- LHS: Kind
-- RHS: Type
kind Dim = Dim TypeInt
               TypeInt
               TypeInt
               TypeInt
               TypeInt
               TypeInt
               TypeInt
```

Thanks to the `Dim`-kind we can force certain types in functions to be of this kind.

This may sound confusing, but the point of this will become clear over time. Let's show some example *types* of the `Dim`-kind.

```haskell
type Length      = 'Dim Pos1 Zero Zero Zero Zero Zero Zero
type Mass        = 'Dim Zero Pos1 Zero Zero Zero Zero Zero
type Time        = 'Dim Zero Zero Pos1 Zero Zero Zero Zero
type Current     = 'Dim Zero Zero Zero Pos1 Zero Zero Zero
type Temperature = 'Dim Zero Zero Zero Zero Pos1 Zero Zero
```

```haskell
type Substance   = 'Dim Zero Zero Zero Zero Zero Pos1 Zero
type Luminosity  = 'Dim Zero Zero Zero Zero Zero Zero Pos1
```

`'Dim` is used to distinguish between the *type* `Dim` (left-hand-side of the `data Dim` definition) and the *type constructor* `Dim` (right-hand-side of the `data Dim` definition, with `DataKinds`-perspective). `'Dim` refers to the type constructor. Both are created when using `DataKinds`.

`Pos1`, `Neg1` and so on corresponds to `1` and `-1` in the imported package, which operates on type-level integers.

Exercise Create types for velocity, acceleration and the scalar.

▼ **Solution**

```haskell
type Velocity     = 'Dim Pos1 Zero Neg1 Zero Zero Zero Zero
type Acceleration = 'Dim Pos1 Zero Neg2 Zero Zero Zero Zero
```

```haskell
type One = 'Dim Zero Zero Zero Zero Zero Zero Zero
```

Multiplication and division

Let's implement multiplication and division on the type-level. After such an operation a new dimension is created. And from the previous section we already know what the dimension should look like. To translate to Haskell-language: "after such an operation a new *type* is created". How does one implement that? With `type family`! A `type family` can easiest be thought of as a function on the type-level.

```haskell
type family Mul (d1 :: Dim) (d2 :: Dim) where
  Mul ('Dim le1 ma1 ti1 cu1 te1 su1 lu1)
      ('Dim le2 ma2 ti2 cu2 te2 su2 lu2) =
      'Dim (le1+le2) (ma1+ma2) (ti1+ti2) (cu1+cu2)
        (te1+te2) (su1+su2) (lu1+lu2)
```

- `type family` means it's a function on type-level.
- `Mul` is the name of the function.
- `d1 :: Dim` is read as "the *type* `d1` has *kind* `Dim`".

Exercise As you would suspect, division is very similar, so why don't you try 'n implement it yourself?

▼ **Solution**

```
type family Div (d1 :: Dim) (d2 :: Dim) where
  Div ('Dim le1 ma1 ti1 cu1 te1 su1 lu1)
      ('Dim le2 ma2 ti2 cu2 te2 su2 lu2) =
      'Dim (le1-le2) (ma1-ma2) (ti1-ti2) (cu1-cu2)
        (te1-te2) (su1-su2) (lu1-lu2)
```

Exercise Implement a type-level function for raising a dimension to the power of some integer.

▼ **Solution**

```
type family Power (d :: Dim) (n :: TypeInt) where
  Power ('Dim le ma ti cu te su lu) n =
    'Dim (le*n) (ma*n) (ti*n) (cu*n) (te*n) (su*n) (lu*n)
```

Now types for dimensions can be created by combining exisiting types, much like we did for values in the previous chapter.

Exercise Create types for velocity, area, force and impulse.

▼ **Solution**

```
type Velocity' = Length `Div` Time
type Area      = Length `Mul` Length
type Force     = Mass   `Mul` Length
type Impulse   = Force  `Mul` Time
```

Perhaps not very exiting so far. But just wait 'til we create a data type for quantities. Then the strenghts of type-level dimensions will be clearer.