Learn You a Physics for Great Good!

>>> WORK IN PROGRESS <<<

Dimensions / Testing of Quantities

[src: Dimensions/Quantity/Test.lhs] Previous: Quantities Table of contents Next: Usage

Testing of Quantity

```
module Dimensions.Quantity.Test
( runTests
)
where
```

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

```
import Prelude hiding (length, div)
import Test.QuickCheck
```

```
import Dimensions.TypeLevel
import Dimensions.Quantity
```

`Quantity` consists of a numerical type such as `Double` and a value-level dimension. Let's assume Haskell's implementation of numbers abide the laws. We also know the value-level dimensions abide the laws since since we tested them in a previous section. So what's there to test now? Well, it's still useful to test the top-level construct. And error may arise when combing different parts.

So let's test *some* things. Due to the type-level dimensions it's impossible to easily and readably design test for all combinations of dimensions. Hence it'll have to suffice with only some combinations.

Generating arbitrary quantities

First we need an `Arbitrary` instance for `Quantity d val`. For `d` we'll mostly use `One` and for `val` we'll exclusively use `Double`.

```
type Q d = Quantity d Double
```

A generator for an arbitrary dimension.

```
genQuantity :: Quantity d Double -> Gen (Q d)
genQuantity quantity = do
  value <- arbitrary
  return (value # quantity)
```

And now we make `Arbitrary` instances of arbitrary selected dimensions in the `Quantity`s.

```
instance Arbitrary (Q One) where
 arbitrary = genQuantity one


instance Arbitrary (Q Length) where
  arbitrary = genQuantity length


instance Arbitrary (Q Mass) where
  arbitrary = genQuantity mass


instance Arbitrary (Q Time) where
  arbitrary = genQuantity time
```

Testing arithmetic properties

On regular numbers, and hence too on quantites with their dimensions, a bunch of properties should hold. The things we test here are

- Addition commutative
- Addition associative
- Zero is identity for addition
- Multiplication commutative
- Multiplication associative
- One is identity for multiplication
- Addition distributes over multiplication
- Subtraction and addition cancel each other out
- Division and multiplication cancel each other out
- Pythagoran trigonometric identity

Let's start!

We could write the type signatures in a general way like

```
prop_addCom :: Q d -> Q d -> Bool
```

But we won't do that since QuickCheck needs concrete types in order to work. So we would have to do a bunch of specialization anyway. And even if we begin with a general signature, we can't cover all cases since there are infinitly many dimensions.

Instead we'll pick some arbitrary dimensions that have an `Arbitrary` instance.

```
-- a + b = b + a
prop_addCom :: Q Length -> Q Length -> Bool
prop_addCom a b = (a +# b) ~= (b +# a)


-- a + (b + c) = (a + b) + c
prop_addAss :: Q Mass -> Q Mass -> Q Mass -> Bool
prop_addAss a b c = a +# (b +# c) ~= (a +# b) +# c


-- 0 + a = a
prop_addId :: Q Time -> Bool
prop_addId a = zero +# a ~= a
  where
    zero = 0 # a
```

```haskell
-- a * b = b * a
prop_mulCom :: Q Length -> Q Mass -> Bool
prop_mulCom a b = a *# b ~= b *# a


-- a * (b * c) = (a * b) * c
prop_mulAss :: Q Time -> Q Length -> Q Mass -> Bool
prop_mulAss a b c = a *# (b *# c) ~=
                    (a *# b) *# c


-- 1 * a = a
prop_mulId :: Q Time -> Bool
prop_mulId a = (1 # one) *# a ~= a


-- a * (b + c) = a * b + a * c
prop_addDistOverMul :: Q Length -> Q Mass -> Q Mass -> Bool
prop_addDistOverMul a b c = a *# (b +# c) ~=
                               a *# b +# a *# c


-- (a + b) - b = a
prop_addSubCancel :: Q Length -> Q Length -> Bool
prop_addSubCancel a b = (a +# b) -# b ~= a


-- (a * b) / b = a
prop_mulDivCancel :: Q Time -> Q Length -> Property
prop_mulDivCancel a b = not (isZero b) ==>
  (a *# b) /# b ~= a


-- sin a * sin a + cos a * cos a = 1
prop_pythagoranIdentity :: Q One -> Bool
prop_pythagoranIdentity a = sinq a *# sinq a +#
                            cosq a *# cosq a ~= (1 # one)
```