## Enhancement request: Dimensionality of Types

**A modest (and completely backwards-compatible) extension to the type system could dramatically increase the power of type validation.**

**Summary**: Golang does not allow mathematical operations between operands of different types, even if they have compatible underlying types. This makes sense for addition and subtraction and is helpful in avoiding a certain class of logical errors. For example, it is almost certainly a logical error to attempt to calculate speed + elapsedTime, assuming their types have been appropriately defined. However, this is too restrictive for multiplication and division: distance = speed \* elapsedTime is a perfectly reasonable calculation. The request is to allow the latter while still forbidding the former, without sacrificing type checking by reducing everything to float64 or other underlying type.

The minimalist implementation of this enhancement would allow mixing as described here with the calculation yielding a result whose type is the underlying type of the operands, e.g. float64. A more powerful implementation would extend the type system to allow the creation of types that are “compounds” of other types formed by multiplication or division. The potential power of this extension is indicated by the fact that it parallels the way dimensions and dimensional analysis operate in science and engineering.

**Full Description**

Imagine you lost a $125M Mars Lander because of a trivial confusion about units of measurement. That’s exactly what happened to NASA over two decades ago:

A NASA review board found that the problem was in the software controlling the orbiter's thrusters. The software calculated the force the thrusters needed to exert in pounds of force. A separate piece of software took in the data assuming it was in the metric unit: newtons.
([Nov. 10, 1999: Metric Math Mistake Muffed Mars Meteorology Mission | WIRED](https://www.wired.com/2010/11/1110mars-climate-observer-report/))

Golang has a powerful feature in its type system that could have prevented this error. If the two routines in question had declared explicit types for the parameter in question, the compiler would have forbidden assignments or parameter passing that confused them:

type forceMetric float64

type forceCustomary float64

However, this feature has limitations in the amount of protection it can offer to the coder. It only protects the simplest of arithmetical cases and assignments, while preventing some more powerful, valuable protections. **A modest (and completely backwards-compatible) extension to the type system could dramatically increase the power of type validation** while retaining the current protections offered and remaining type-safe.

I refer to this extension as **dimensionality**. It mirrors the concept of dimension universally used in science and engineering, and allows the compiler to aid the programmer by automatically performing [dimensional analysis](https://en.wikipedia.org/wiki/Dimensional_analysis).

As an example, consider the following code fragment that attempts to calculate a velocity from the division of distance by time.

type distance float64

type time float64

type velocity float64

var v velocity

var d distance = 100

var t time = 2

v = d / t //will not compile!

The compiler will reject the assignment because d and t have different types, although they have the same underlying types. We could make it compilable by the following explicit type conversions:

v = velocity(float64(d) / float64(t))

However, now we have sacrificed the type checking that we had hoped for by creating types for distance and time; d and t could mistakenly be variables of any type whose underlying type is float64.

The proposal here is to make the original assignment v = d / t legal and safe while retaining the value of declaring different types for different units, through a small addition to the syntax of types. We do so by retaining the existing restrictions on mixing types in addition, subtraction, and assignment, while extending the rules to **allow** mixing types in multiplication and division, in a way that mirrors dimensionality in science and engineering.

Continuing our example from above, we would like velocity to have units (dimensions) of distance / time. In this proposed enhancement, the following type declaration would be legal:

type velocity (distance / time)

We can then declare our variables as before and make a legal, validated assignment:

var v velocity // v has dimension (distance / time)

var d distance = 100

var t time = 2

v = d / t //Dimensional analysis matches, so compiles

v = t / d //Wrong dimensions! Will not compile

Dimensionality can be compounded to any reasonable degree as needed, for example:

type velocity (distance / time)

type acceleration (velocity / time)

type force (mass \* acceleration)

type energy (mass \* velocity \* velocity)

The compiler will be responsible for analyzing such compound dimensions down to their component dimensions and ensuring that the types match.

This proposal is backwards-compatible since all existing correct code will continue to compile and behave as before.