Implementation of Dimension Types in C++0x

Dr Leonid A. Timochouk

FICC Core Libraries, UBS Investment Bank
Europastrasse 1, 8152 Opfikon, Switzerland

L.Timochouk@gmail.com

2 July 2011

1 Introduction

a FIG

The idea of dimension types is nothing new (see Section 3 for a historical overview). Informally speaking, *dimension types* is a feature of a type system of a compiled programming language which allows the user to express the physical dimensions and units of quantities represented by entities of that language, and to enforce, by means of static type-checking performed at compile-time, the correct structure of dimensioned expressions.

For example, if the constant `c` represents the speed of light in the vacuum, then the C/C++ definition like

```
double const c = 299792458.0;
```

only tells the compiler (and the programmer) that the machine representation of c is `double` and that c has the constant value given above. However, what we would really like to express is that c is a physical quantity, its dimension is that of the velocity (i. e., lt^{-1}, where l is length and t is time), and its physical unit is $km\ sec^{-1}$.

Capturing dimension and unit properties of physical quantities in a programming language is indeed important because this would preclude erroneous operations with such dimensioned quantities. For example, if we define another quantity to represent the astronomical unit (roughly speaking, this is the mean distance in km between Earth and Sun),

```
double const AU = 149597870.6996262;
```

then expressions like `AU / c` and even `AU * c` are formally valid. The former represents the time t required for light to travel the distance of $1AU$, and the latter has no straightforward physical interpretation (its dimension is $l^2\ t^{-1}$ and the unit is $km^2\ sec^{-1}$) but it is nevertheless well-formed.

On the other hand, the expression `AU + c` is a valid C/C++ expression but it is clearly ill-formed from the physical point of view (contains addition of quantities of different physical dimensions). Furthermore, if `AU` is expressed in meters rather than in kilometers:

```
double const AUm = 149597870699.6262;
```

then expressions `AUm / c` and `AUm * c` also become ill-formed because their operands are expressed in incompatible units.

Generally speaking, any *monomial* expressions over dimensioned quantities (multiplication, division, raising to a rational power) are well-formed provided that the operands' units are the same. However, *polynomial* expressions (sums and differences) are well-formed if and only if all their monomial terms have same physical dimensions and units. Furthermore, any non-polynomial functions (e. g. `exp`, `sin`, ...) can only be applied to *dimension-less* quantities, because otherwise the Taylor power series of such functions, viewed as polynomials of infinite degree, would be ill-formed. For example, the expression $\exp c$ where c is the speed of light defined above is ill-formed, whereas $\exp\left(\frac{c}{v}\right)$ where v is another velocity expressed in $km\ sec^{-1}$ is well-formed because $\frac{c}{v}$ is dimension-less.

However, the standard type systems of all mainstream programming languages (C, C++, Java, FORTRAN, Ada, OCaml, ...) do not allow us to express the dimensions and units of physical quantities directly. As a result, it would not be possible to capture errors originating from invalid dimensions and/or units at compile-time (or indeed at run-time, before such an error manifests itself as a software failure).

This is indeed an important software engineering problem which particularly affects scientific and engineering computing and real-time control systems. For example, the loss of Mars Climate Orbiter spacecraft in 1999 was caused by a software error which was due to mix-up of metric and imperial units across the ground and on-board software components [9]. The failure was attributed by the NASA investigation board to various management shortcomings in the software development and testing cycle. However, the underlying cause of this high-profile mishap was the inability of the programming language used (apparently FORTRAN) to capture dimension / unit errors through its type system.

2 Requirements for a dimensioned type system

It is desirable that a system of dimension types implemented for a programming language X would possess the following properties.

First, X itself should be an accepted mainstream programming language used sufficiently widely for scientific, engineering and real-time computations, as these application areas would benefit most from the advent of dimension types.

Second, dimension types should preferably be implemented in terms of the standard type system of language X without introducing any new syntactic constructs or annotations. Consequently, type-checking of dimensioned expressions should be performed by the X compiler as part of normal type-checking process at compile-time, without the recourse to any external tools and without the need to modify the compiler itself.

Third, dimension types should provide zero run-time overhead, either temporal or spatial, compared to representing the corresponding quantities solely in terms of underlying elementary data types (such `double` in C/C++). In other words, the whole complexity of dimensioned type system must be absorbed at compilation time only.

Fourth, the system of dimension types must allow for monomial expressions over dimensioned quantities containing arbitrary *rational* (not just integral) powers. For example, a monomial expression like $c^{\frac{1}{2}} AU^{\frac{3}{4}}$ (see Section 1) should be accepted.

Fifth, it should be possible to express dimensioned quantities in multiple units, to convert them automatically between different units and to check the consistency of units used as part of the over-all type-checking process.

For example, for the dimension l (length) the units could be meters, kilometers and so on. Continuing with the above example, the expression $c^{\frac{1}{2}} AU^{\frac{3}{4}}$ has dimension $l^{\frac{5}{4}} t^{\frac{-1}{2}}$ where t is time. This expression is well-formed if only if the unit of length used in both c and AU is the same. If that unit was, for example, kilometer but we now want to change it to meter, than the value of any expression of dimension proportional to $l^{\frac{5}{4}}$ would be multiplied by the factor $1000^{\frac{5}{4}}$ and this should be performed automatically via a dimensioned type conversion operator. A similar operation would be applied when the unit of time t changes.

An alternative approach is to maintain just one fundamental unit for each dimension and to automatically convert all dimensioned quantities to the fundamental units at construction time. However, in our view, this approach has two disadvantages:

- it would enforce data representation which may be physically counter-intuitive (for example, the fundamental unit for length l is meter but in nuclear physics, microscopic units of length are used instead);
- it may also incur run-time overhead from unnecessary implicit data conversions.

In addition, there is a problem of configurability of the set of dimensions and units. In physics, there are 7 fundamental dimensions which are standardised in the SI system and considered to be irreducible with respect to each other [2]:

Dimension	Fundamental unit (SI)
Length (l)	meter (m)
Time (t)	second (sec)
Mass (m)	kilogram (kg)
Electrical current (I)	Ampere (A)
Thermodynamic temperature (T)	Kelvin (K)
Substance amount (n)	mole (mol)
Luminous intensity (I_v)	candela (cd)

Table 1: Fundamental physical dimensions and units.

It is therefore recommended that any implementation of dimension types should provide for *at least* the fundamental dimensions and units specified in Table 1. However, there are several issues here which should be noted.

First of all, the 7 fundamental dimensions are not entirely independent from each other; they are connected, for example, via the fundamental physical constants (such as c). Some implementations of dimension types try to take such connections into account as discussed in Section 3 below, though in our view, this is not strictly necessary and is more confusing than useful.

Similarly, one can argue that the substance amount n is not really a fundamental dimension but a dimension-less quantity because it is in fact proportional to the number of molecules of a given substance; nevertheless, it has been internationally standardised in the SI system as a fundamental dimension and the standard should be adhered to.

More importantly, the user of a dimensioned type system may want to introduce their own dimensions. For example, although the *angle* and *solid angle* (their fundamental units being radian and steradian, respectively) are considered in the SI system to be dimension-less quantities, it is often useful to make them into stand-alone dimensions, for example in order to be able to express them in other conventional units as well (such as degrees and square degrees), whereas dimension-less quantities have no units. In financial computations, it would be useful to have *money* as yet another dimension, e. g. with US$ being its fundamental unit; and so on.

It would therefore be advantageous if a system of dimension types allows the user to declare new dimensions apart from those listed in Table 1, and also to create arbitrary units of any dimensions. Unfortunately, it appears that this requirement would typically lead to significant complications in the implementation of dimension types and would conflict with other requirements presented above. For this reason, this feature is currently not supported by the existing implementations we are aware of.

Typically, a dimensioned type system would provide a pre-configured set of dimensions (though it may be larger than the 7 fundamental SI dimensions) and a pre-configured set of units for each dimension (or just one fundamental unit).

3 Overview of existing research

Dimensional analysis has been used extensively in theoretical physics (see for example [8, 7]) in order to provide approximate values for various physical quantities (especially in nuclear physics) or to reduce the dimensionality of partial differential equations of mathematical physics, possibly simplifying them down to ordinary differential equations (construction of self-similar solutions, e. g. in continuous media mechanics or in mathematical finance).

In computer science, the idea of augmenting the type systems of programming languages by information on physical dimensions of represented quantities was first developed in late 1970s – early 1980s. A systematic theory of dimension types (including formal type inference rules) was provided in [3, 4] and a software implementation of dimension types was constructed within the ML Kit compiler of Standard ML. However, there are three problems with this approach:

- only the dimensions of quantities were considered, not the units; more precisely, only one unit for each dimension was allowed, so the spaces of dimensions and units were isomorphic;
- implementation of dimension types in ML Kit required modifications to the compiler which is undesirable from the practical point of view;
- ML Kit and Standard ML are not widely used in scientific and engineering computations which hindered integration of dimension types into software engineering practices.

More recently, dimension types have been implemented in Haskell [1]. This implementation is based only on the native Haskell type system features (such as multi-parameter type classes, phantom types, functional dependencies etc) supported by Glasgow Haskell Compiler. However, it still has some limitations:

- arbitrary rational powers of dimensioned quantities are not supported; the only allowed denominators are 1 and 2;
- multiple units for each dimension are allowed, but all such units are obtained from one fundamental unit by applying various SI decimal prefixes.

Recent versions of computer algebra systems Maple [5] and Mathematica [6] are equipped with rather comprehensive modules for dimensional and unit analysis. However, Maple and Mathematica programming languages are interpreted and dynamically-typed, so their dimensional analysis tools are essentially outside the definition of dimension types as presented in Section 1. Furthermore, the primary purpose of these systems is symbolic and algebraic computations; they are less well optimised for heavy-duty numerical computations (scientific, engineering, real-time) than the mainstream compiled programming languages.

Of the general-purpose programming languages, the (arguably) most suitable one for supporting dimension types is C++. This is due to C++ template meta-programming mechanisms which provide a Turing-complete *compile-time* evaluation engine [10]. In particular, parameterisation of (templated) C++ types can be used to represent dimension and unit information. There exist several dimension types implementations in C++: [**?**].

References

[1] BUCKWALTER, B. Announce: `Dimensional` — statically checked physical dimensions. Tech. rep., haskell@haskell.org mailing list, December 2006. http://www.haskell.org/pipermail/haskell/2006-December/018993.html.

[2] INTERNATIONAL UNION OF PURE AND APPLIED CHEMISTRY. *Quantities, Units and Symbols in Physical Chemistry*, 2nd ed. Blackwell Science, Oxford, 1993.

[3] KENNEDY, A. J. Dimension types. In *Proceedings of the 5th European Symposium on Programming '94* (Berlin, 1994), vol. 788 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 348–362.

[4] KENNEDY, A. J. Rational parametricity and units of measure. In *Proceedings of the 24th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, 1997), ACM Press.

[5] MAPLE 9.5 APPLICATION PAPER. Units and dimensional management. Tech. rep., MapleSoft, August 2001. http://www.maplesoft.com/view.aspx?SF=4036/Units.pdf.

[6] MCLOONE, J. Automatic physical units in *Mathematica*. Tech. rep., Wolfram Blog, December 2010. http://blog.wolfram.com/2010/12/09/automatic-physical-units-in-mathematica.

[7] OVSYANNIKOV, L. V. *Group Analysis of Differential Equations*. Nauka, Moscow, 1978. In Russian.

[8] SEDOV, L. I. *Similarity and Dimensional Methods in Mechanics*, 10th ed. CRC Press, Boca Raton, FL, 1993.

[9] STEPHENSON, A. G., AND ET AL. Mars Climate Orbiter Mishap Investigation Board Phase I Report. Tech. rep., NASA, November 1999.

[10] VELDHUIZEN, T. L. C++ templates are Turing complete. Tech. rep., Indiana University of Computer Science, 2003. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.14.3670.